Algorithms and Data Structures for Logic Synthesis and Verification Using Boolean Satisfiability

A DISSERTATION
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

John D. Backes

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
Doctor of Philosophy

Advisor: Marc D. Riedel

March, 2013

Acknowledgements

At the time of writing this dissertation, I have spent nearly one third of my life studying at the University of Minnesota. Throughout this time I have met many brilliant and kind people that have not only shaped me academically, but characteristically into the person that I am today. I had the pleasure of working, learning, and becoming friends with Weikang Qian, Mustafa Altun, Hua Jiang, Sasha Karam, Phil Senum, Saket Gupta, Denis Kune, and many other students that are far more brilliant than I am. I would also like to thank Brian Isle, Tom Haigh, Jason Sonnek, Steve Harp, Todd Carpenter, and everyone at Adventium labs for giving me the opportunity to collaborate with them. I had an amazing experience working with Neha Rungta, Oksana Tkachuk, and Suzette Person at the NASA Ames Research Center. I feel obliged to mention my time spent in the Bay Area would not have been the same without Evan Long, Atil Işçen, Will Curran, Carrie Rebhuhn, and Josie Hunter.

I am lucky to have had such a great and positive committee for my dissertation. Sachin Sapatnekar taught my first course in graduate school: VLSI Design Automation. His teaching helped me tremendously when I began doing research in EDA. Kia Bazargan has given me great feedback on my research, and his mentorship during "Teaching Experiences in ECE" taught me how to best articulate my ideas. When it came time for me to ask professors to be on my committee, I wanted to find someone in the computer science department that had experience with Formal Methods. I cannot articulate accurately how grateful I am that I found Mike Whalen. Dr. Whalen's advice for my research has been extremely helpful, and I will never be able to adequately repay him for the doors of opportunity that he has opened for me in my professional life.

During my third semester of undergraduate study, I was frustrated to find that a new professor who was supposed to teach the recitation for my class on "Introduction to Digital System Design" did not show up to the first discussion. Apparently he had confused the names of the buildings "Akerman Hall" and "Amundson Hall" and was over 15 minutes late. To make it up to his students, he offered extra offices hours so we could get caught up on the material that was supposed to be taught during the discussion. While Professor Riedel's complete lack of temporal organization was sometimes frustrating in the future, the disorganization inadvertently caused the first of what would be many meetings over the next six years of my life. Marc's gleeful excitement when discussing technical concepts and unending positive feedback was what convinced me to go to graduate school. He routinely encouraged me to study whatever topics I found interesting. I am thoroughly convinced that I would not have been successful studying under the guidance of anyone else other than him.

Finally, I must thank my brilliant and beautiful girlfriend Caitlyn. She gave me relentless encouragment and had an endless amount of patience with me over the past three and a half years. She was truly my greatest discovery during graduate school.

Dedication

To my parents,

who have always been able to find ways to satisfy all of my constraints...

Abstract

Boolean satisfiability (SAT) was the first problem to be proven to be NP-Complete. The proof, provided by Stephen Cook in 1971, demonstrated that inputs accepted by a non-deterministic Turing machine can be described by satisfying assignments of a Boolean formula. The reduction to SAT feels natural for a wealth of decision problems; this has motivated an immense amount of research into heuristics for solving SAT instances quickly. Over the past decade the performance of SAT solvers has improved tremendously, and as a consequence, real-world problems that were once thought to be intractable are now feasible in many cases.

In this thesis we discuss how some problems in logic synthesis and verification can be solved with Boolean satisfiability. The dissertation begins by discussing Cyclic Combinational Circuits. Cyclic Combinational Circuits are logic circuits that contain feedback, but exhibit no state behavior. Many functions can be implemented with fewer gates using a cyclic topology rather than an acyclic topology. A pivotal step in synthesizing these circuits is proving whether or not the resulting structure is actually combinational, and if not, how to modify the circuit to behave properly. This analysis can be elegantly cast as an instance of SAT. Furthermore, this thesis demonstrates how modern SAT-Based synthesis techniques can be used to generate cyclic structures, rather than just analyze them.

These SAT-Based synthesis techniques rely on augmenting proofs of unsatisfiability to generate circuit structures. These structures, called Craig Interpolants, and the proofs they are generated from are the focus of the second portion of this dissertation. Techniques are proposed for reducing the size of these interpolants, and then the use of proofs of unsatisfiability as an underlying data structure for synthesis is advocated.

Finally, the last portion of this thesis discusses some improvements to a new model checking algorithm known as Property Directed Reachability (PDR). This algorithm iteratively solves SAT instances representing discrete time frames of a sequential circuit in order to demonstrate that a state invariant exists.

Contents

List of Tables

List of Figures

Chapter 1

Overview

This first chapter serves as an introduction to concepts and terms that are common to all the topics presented in this dissertation. We begin by discussing the organization and how the subjects are related.

1.1 Organization and Contributions

This thesis encompasses many of the topics that I researched during my graduate education. Each of these topics naturally evolved out of the previous while maintaining the common theme of Boolean Satisfiability. The topics in this thesis are presented in the order in which preserves this natural evolution. Figure 1.1 shows how the topics relate to each other. The first topic discusses how to analyze and synthesize cyclic combinational circuits with Boolean Satisfiability. The synthesis method relies on generating Craig Interpolants to generate functional dependencies. The second topic, Reduction of Interpolants for Logic Synthesis, discusses a method for reducing the size of these interpolants. This method modifies the structure of resolution proofs to reduce the complexity of the interpolant's structure. The third topic discusses how resolution proofs can be used as an underlying data structure in which to perform synthesis. Resolution proofs and Craig Interpolants are common constructs used in formal verification. The final topic of this thesis discusses some improvements to a new SAT-based algorithm for formally verifying safety properties in a finite-state transition system.

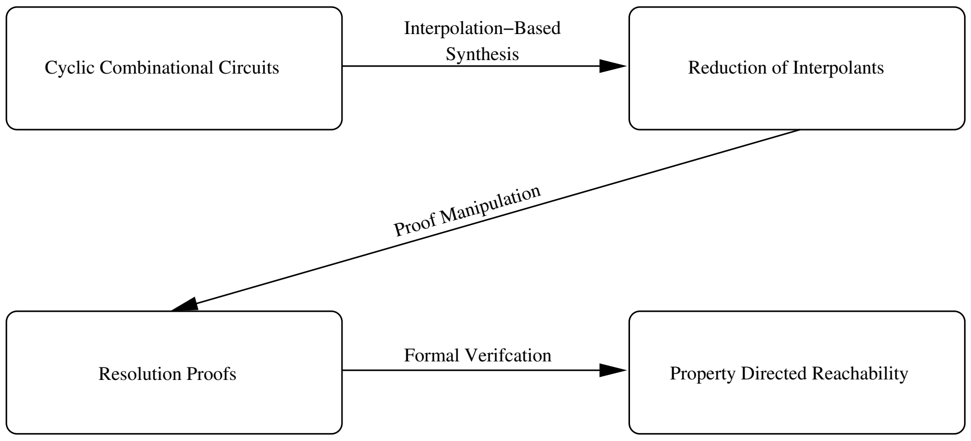

Cyclic Combinational Circuits
Interpolation−Based Synthesis
Reduction of Interpolants
Proof Manipulation
Resolution Proofs
Formal Verifcation
Property Directed Reachability

Figure 1.1: A diagram of how the thesis topics relate to each other.

Explicitly stated, this thesis makes the following contributions:

- We propose a SAT-Based algorithm to analyze and synthesize cyclic combinational circuits on a network level. This method synthesizes cyclic circuits while trying to minimize the support set size of the circuit's intermediate functions.

- We propose a SAT-Based algorithm for analyzing and mapping cyclic circuits with gate-level implementations. There are subtle challenges associated with translating cyclic circuits from a network-level to a gate-level implementation that we address.

- We propose a method for reducing the size of Craig Interpolants used to synthesize functional dependencies.

- We discuss the use of resolution proofs as a general data structure for logic synthesis.

- We discuss some improvements to a new model checking algorithm known as Property Directed Reachability.

In the following section we describe concepts and notation that are common to most of the concepts in this thesis.

1.2 Definitions, Notation, and Common Concepts

1.2.1 Boolean Satisfiability

Boolean satisfiability is the problem of determining whether there exists some truth assignment to the variables of a Boolean formula such that the formula is satisfied (is true for that assignment). Many combinatorial problems can be described simply by a set of Boolean constraints. Thus many combinatorial problems are easily translated into the Boolean Satisfiability problem. For example, consider the problem of three Ph.D. students (Hua, John, and Mustafa) deciding where to go to happy hour. This problem consists of a number of variables and constraints. The variables and their meanings are listed in Figure 1.2. The constraints and their meanings are listed in Figure 1.3.

Variable	Meaning
Joh_Att	*John will attend*
Mus_Att	*Mustafa will attend*
Hua_Att	*Hua will attend*
Hua_Dea	*Hua has a paper deadline he needs to meet*
Mus_Per	*Mustafa has permission to go from his wife Banu*
Gro_Mee	*There is a group meeting scheduled for the evening*
War_Wea	*The weather is warm*
Bur_Loc	*The students will go to the bar: "Burrito Loco"*
Stu_Her	*The students will go to the bar: "Stub & Herbs"*
Sal	*The students will go to the bar: "Sally's"*

Figure 1.2: Variables for deciding what bar to go to for happy hour.

The truth assignments to the Boolean variables listed in Figure 1.2 that satisfy all of the constraints in Figure 1.3 represent the possible scenarios in which at least some

Constraint	Meaning
$Joh\_Att \rightarrow Hua\_Att \vee Mus\_Att$ $Hua\_Att \rightarrow Joh\_Att \vee Mus\_Att$ $Mus\_Att \rightarrow Hua\_Att \vee Joh\_Att$	*A student will only attend if at least one of the other students attends*
$Mus\_Att \rightarrow Mus\_per \wedge \neg Gro\_Mee$	*Mustafa will attend only if he has permission and there is no group meeting*
$Hua\_Att \rightarrow \neg Hua\_Dea \wedge \neg Gro\_Mee$	*Hua will attend only if he does not have a paper deadline and there is no group meeting*
$Bur\_Loc \rightarrow \neg Hua\_Att$	*Hua refuses to go to "Burrito Loco"*
$Stu\_Her \rightarrow \neg War\_Wea$	*The students will only go to "Stub and Herbs" if the weather is not warm*
$Sal \rightarrow War\_Wea \vee \neg Joh\_Att$	*John refuses to go to "Sally's" unless the weather is warm*
$Bur\_Loc \rightarrow \neg Sal \wedge \neg Stu\_Her$ $Stu\_Her \rightarrow \neg Sal \wedge \neg Bur\_Loc$ $Sal \rightarrow \neg Bur\_Loc \wedge \neg Stu\_Her$	*The students cannot attend more than one bar*
$Joh\_Att \vee Hua\_Att \vee Mus\_Att$	*At least one student will attend*

Figure 1.3: Constraints for deciding what bar to go to for happy hour

of the students will attend happy hour. For example, the truth assignment: $Joh\_Att = Hua\_Att = Mus\_Att = War\_Wea = Mus\_Per = Sal =$ **true**, $Hua\_Dea = Bur\_Loc = Stu\_Her = Gro\_Mee =$ **false** satisfies all of the constraints. This indicates a scenario where all of the students attend "Sally's". Another satisfying assignment: $Joh\_Att = Mus\_Att = Hua\_Dea = Mus\_Per = Stu\_Her =$ **true**, $Hua\_Att = Bur\_Loc = Sal = Gro\_Mee =$ **false** indicates a scenario where only John and Mustafa go to "Stub & Herbs" because Hua has a paper deadline. Notice that in this scenario the truth value of the variable War_Wea does not affect the truth values of the constraints. There are several truth values which do not satisfy all of the constraints. For example, whenever there is a group meeting, neither Mustafa nor Hua will be able to go out for happy hour. Because neither Hua nor Mustafa will attend happy hour, John will also not attend (though he doesn't seem to care about the group meeting...). In fact, it can be shown through repeated use of the inference rule of resolution that the constraint: $Gro\_Mee \rightarrow \neg(Joh\_Att \vee Hua\_Att \vee Mus\_Att)$ can be implied by the conjunction of constraints in Figure 1.3. This means that the final constraint: "At least one student will attend." cannot be satisfied when there is a group meeting scheduled.

For the majority of this thesis, we use the "Electrical Engineering" notation: addition $(x + y)$ denotes disjunction, multiplication (xy), denotes conjunction, a "plus with a circle" $(x \oplus y)$ denotes inequivalence (exclusive OR), and an overbar $(\bar{x})$ or a "dash with a tail" $(\neg x)$ denotes negation. However, in the example above, in the final chapter, and in some pseudocode, we use the standard mathematical notation: a "$\vee$" denotes disjunction, a "$\wedge$" denotes conjunction, and a "$\rightarrow$" denotes implication. We often use the number 1 to indicate the constant **true** and the number 0 to indicate the constant **false**; although we sometimes explicitly say "true" or "false". We use the term "Boolean formula" and "Boolean function" interchangbly. If we are discussing a function that is not Boolean (either its arguments are not Boolean or its result is not Boolean) it will be made clear by context.

An appearance of a variable in a Boolean formula, either negated or non-negated, is refereed to as a **literal**. A **clause** or a **sum** is a disjunction (OR) of literals. A conjunction (AND) of literals is referred to as a **cube** or a **product**. A Boolean formula is in conjunctive normal form (CNF) if it is a conjunction (AND) of clauses. We will sometimes refer to a CNF formula as a set of clauses, when it is clear by context. The vast majority of SAT solvers only operate on CNF formulas. Though there are some that operate on circuit structures, or a combination of circuit structures and CNF formulas [2]. Any representation of a Boolean formula can be transformed into a CNF formula that preserves the satisfiability of the original in linear time by adding additional variables and constraints. The most common type of transformation, introduced by Tseitin in [3], is described in subsection 1.2.2.

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 logic circuit with a single primary output as a SAT instance; the satisfiability of the primary output can be represented as a CNF formula.

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).

A **partial assignment** of a function's support variables is a valuation of that function over a subset of its support variables; the result of a partial assignment is either 0, 1, or $\perp$ (the definition of $\perp$ is described in Chapter 2). A product is said to **cover** a partial assignment if that product evaluates to 1 for that partial assignment. Similarly, a sum is said to cover a partial assignment if the sum evaluates to 0 for that partial assignment.

When we assert some implication or bi implication in the text, we are claiming that the implication or bi implication is a tautology. For example, if we say something like "the cube c is true because $f \rightarrow a$ and $a \leftrightarrow c$", then we are asserting that $f \rightarrow a$ and $a \leftrightarrow c$ are both tautologies.

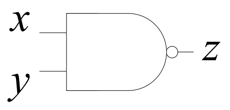
Given a product p and a function f, we say that "p is a product of f" if $p \rightarrow f$. Likewise, given a sum s and a function f, we say that "s is a sum of f" if $f \rightarrow s$.

A **prime implicant** p of some function f is a product such that $p \rightarrow f$ and p is not covered by any other product q of f ($p \not\rightarrow q$). A **prime implicate** s of some function f is a sum such that $f \rightarrow s$ and s is not covered by any other sum q of f ($q \not\rightarrow s$) [4].

1.2.2 Tsestin Transformation

In order for CNF-based SAT Solvers to reason about circuit structures, a translation from a combinational circuit to an equivalent CNF formula must take place. The most common, and arguably the most simple, type of translation was introduced by Tsetien [3].

The transformation generates a CNF formula such that the number of clauses and variables are linear in the size of the circuit. The translation works be converting each individual logic gate into a set of clauses. These clauses are in terms of variables assigned to each wire connecting to the logic gate. An example template for a NAND gate is shown in Figure 1.4

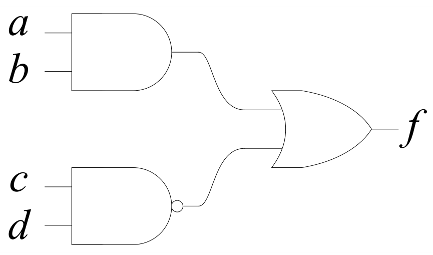


$$(x + z)(y + z)(\bar{x} + \bar{y} + \bar{z})$$

Figure 1.4: Nand Template for Tseitin transformation

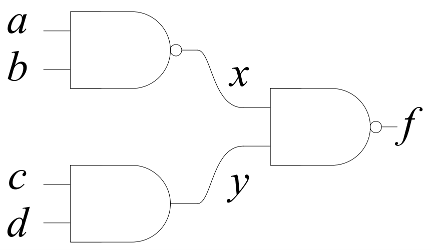
While the original circuit expresses a Boolean function in terms a set of primary input variables, the transformed CNF formula is expressed in terms of not only the primary input variables but also variables representing the state of each individual wire in the circuit. Assignments that satisfy the final formula correspond to **valid states** of the circuit. For example, consider the circuit and corresponding formula shown in Figure 1.5.

Using Demorgan's law, the OR gate can be converted into an AND gate with inverters on every input and output. Applying Demorgan's law, simplifying multiple inversions, and applying the template given in Figure 1.4 yields the circuit and corresponding Tseitin transformation shown in Figure 1.6



$f = ab + \overline{cd}$

Figure 1.5: A circuit implementing function $f = ab + \overline{cd}$.



$$(a + x)(b + x)(\bar{a} + \bar{b} + \bar{x})$$
$$(c + \bar{y})(d + \bar{y})(\bar{c} + \bar{d} + y)$$
$$(x + f)(y + f)(\bar{x} + \bar{y} + \bar{f})$$

Satisfiable Assignments			
a, b, c, d	$x = \overline{ab}$	$y = cd$	$f = \overline{xy}$
0 0 0 0	1	0	1
0 0 0 1	1	0	1
0 0 1 0	1	0	1
0 0 1 1	1	1	0
0 1 0 0	1	0	1
0 1 0 1	1	0	1
0 1 1 0	1	0	1
0 1 1 1	1	1	0
1 0 0 0	1	0	1
1 0 0 1	1	0	1
1 0 1 0	1	0	1
1 0 1 1	1	1	0
1 1 0 0	0	0	1
1 1 0 1	0	0	1
1 1 1 0	0	0	1
1 1 1 1	0	1	1

Figure 1.6: The circuit in Figure 1.5 expressed in terms of AND gates and inverters. The Tseitin transformation using the template from Figure 1.4 is shown beneath the circuit. The table on the right side of the Figure lists all the satisfying assignments of the SAT instance.

There are 16 valid states of the circuit in Figure 1.5. Hence there are only 16 assignments that satisfy the clauses in Figure 1.6. These assignments are explicitly listed on the right hand side of the Figure. It may be somewhat counterintuitive that the assignments that satisfy the SAT instance in Figure 1.6 do not neccesarily correspond to those that cause the primary output variable f to evaluate to 1. In order to check the satisfiability of the circuit in Figure 1.5, the clause containing the single variable f can be added to the SAT instance in Figure 1.6 to assert that f must be 1 in order to satisfying the formula.

Chapter 2

Cyclic Circuits and Boolean Satisfiability

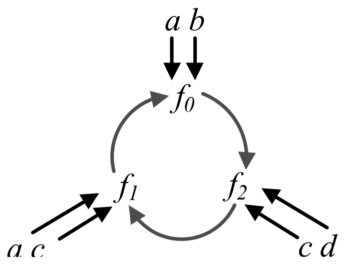
2.1 Introduction

2.1.1 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., [5], p. 14; [6], p. 193).

And yet, cyclic circuits can be combinational. Consider the truth table of values and the functions shown in Figure 2.1. The definition of these functions is cyclic. In spite of 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 2.2, 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 2.3, 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.

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



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

Figure 2.1: Example: A cyclic circuit with 4 primary inputs and 3 primary outputs.

Cyclic circuits can be analyzed on a network level, such as the example in Figure 2.1, or on the level of logic gates mapped to some technology. A cyclic combinational circuit mapped to two-input AND and OR gates is shown in Figure 2.4. This circuit is also combinational in the strictest sense: it produces the required output values *regardless*

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

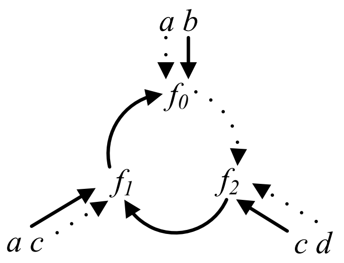


Figure 2.2: Network in Figure 2.1 with $a = 1, b = 0, c = 1, d = 0$.

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

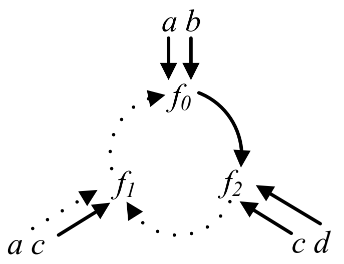


Figure 2.3: Network in Figure 2.1 with $a = 1, b = 1, c = 0, d = 0$.

of the prior values on the wires and for *any* choice of delay parameters. If $x = 0$ then g_1 produces an output of 0, because 0 is a controlling value for an AND gate. If $x = 1$ then g_4 produces a value of 1, because 1 is a controlling value for an OR gate. In both cases, the cycle is broken and the circuit produces definite outputs. Since x must assume one of these two values, we conclude that the circuit *always* produces definite outputs. In fact, it implements two functions that both depend on all five variables:

$$\begin{aligned} f_1 &= b(a + x(d + c)), \\ f_2 &= d + c(x + b\,a) \end{aligned} \tag{2.1}$$

(+) denotes OR, (·) denotes AND

Note that the computation of the two functions overlaps. If we were to implement these functions with an acyclic circuit, we would need eight two-input gates. There can be subtle differences between the behavior of a cyclic circuit defined on the network-level and it's gate-level implementation. We discuss these differences towards the end of this chapter.

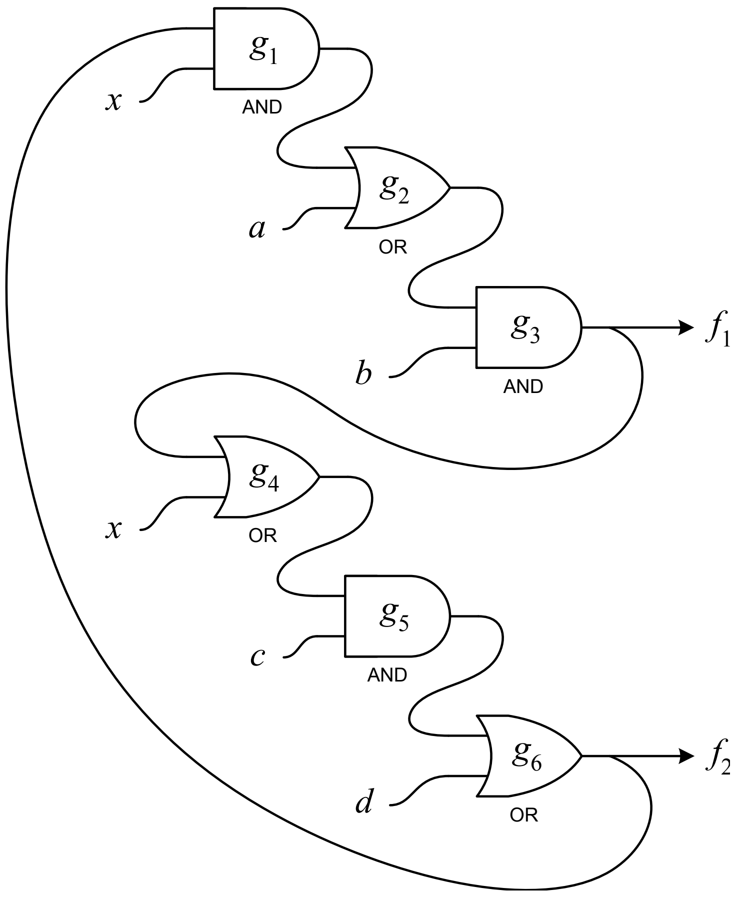


Figure 2.4: A cyclic combinational circuit.

The concept of cycles in combinational circuitry is conceptually similar to that of *false paths*. Khrapchenko was the first to recognize that *depth* and *delay* in a circuit are not equivalent concepts: the critical paths of a circuit may all be false, i.e., they might be blocked by off-path controlling values; as a consequence, the delay of the circuit might be less than its topological depth [7]. For a cyclic circuit, we can say that it is combinational if all of its cycles are false; no input assignment ever causes a cyclic path to be sensitized. 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 and so optimize the delay [8].

In previous work, it was shown that combinational circuits can be optimized significantly if cycles are introduced [9]. 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. The proposed methodology for synthesizing such circuits demonstrated that significant improvements in area and in delay could be. Cycles are introduced in the restructuring and minimization phases of logic synthesis at the level of functional dependencies.

2.1.2 Prior and Related Work

In an earlier era, theoreticians commented on the possibility of having cycles in combinational logic and conjectured that this might be a useful property [10], [11], [12]. Both McCaw and Rivest presented examples of cyclic circuits with provably fewer gates than is possible with equivalent acyclic circuits [13], [14].

Stok lamented that EDA tools were rejecting cyclic designs because there was no way to validate them [15]. In response, Malik discussed analysis techniques for cyclic combinational circuits [16]. 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 [17].

More recently, Neiroukh and Edwards discussed analysis strategies targeting cyclic circuits that are produced inadvertently during design [18, 19]. Following a strategy similar to Malik's, they proposed techniques for transforming valid cyclic circuits into functionally equivalent acyclic circuits [19]. 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.

Riedel was the first to suggest a method for synthesizing cyclic circuits [9]. The method was implemented in a package called CYCLIFY, built within the Berkeley SIS environment [20]. 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 [21]. The complexity of this sort of timing analysis is, in fact, the same for cyclic circuits as for acyclic circuits [22]. 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.

2.1.3 SAT-Based Synthesis

This chapter tackles the problem of synthesizing cyclic combinational circuits with SAT-based techniques. Specifically, we build off of a technique based on Craig interpolation for synthesizing functional dependencies [23]. 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 2.5. 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 look-up tables, at least one additional look-up 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 provides an implementation. Figure 2.6 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.

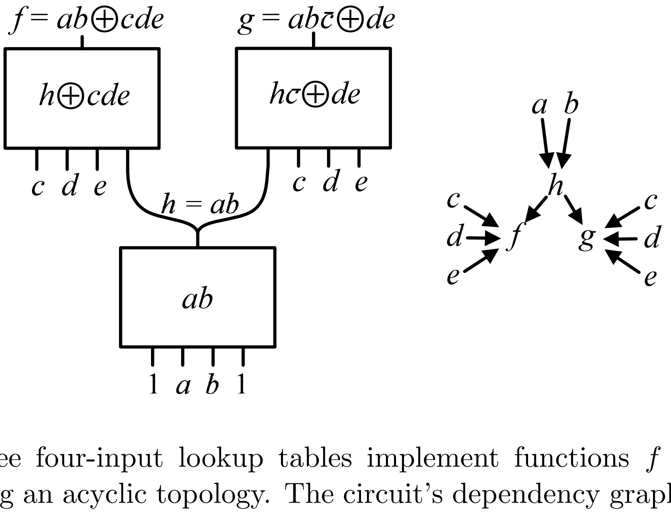


Figure 2.5: Three four-input lookup tables implement functions $f = ab \oplus cde$ and $g = ab\bar{c} \oplus de$ using an acyclic topology. The circuit's dependency graph is shown on the right.

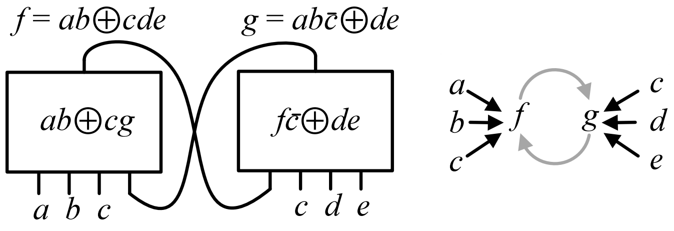


Figure 2.6: Two four-input lookup tables implement functions $f = ab \oplus cde$ and $g = ab\bar{c} \oplus de$ using a cyclic topology. The circuit's dependency graph is shown on the right.

2.2 Circuit and Network Model

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

We adopt a *ternary framework* for analysis. We assume that, at the outset, all wires in a circuit have *undefined* values, which we denote with the symbol $\perp$. Here $\perp$ captures both uncertainty as well as possible ambiguity: the signal might be 0 or 1 – but we do not know which; or it might not even have logical value, i.e., it could be a voltage value between logical 0 and logical 1. We say that a variable's value is *definite* or *known* if its value is 0 or 1 and that it is *indefinite* or *ambiguous* if it is $\perp$. The idea of three-valued logic for circuit analysis is well established. It was originally proposed for the analysis of *hazards* in combinational logic [24]. Bryant popularized its use for verification [25], and it has been widely adopted for the analysis of asynchronous circuits [26].

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

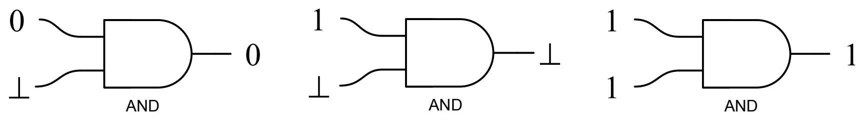


Figure 2.7: An AND gate with 0, 1, and $\perp$ inputs.

Consider the circuit fragment in Figure 2.8. One might be tempted to reason as follows: the output of the AND gate g_1 is fed in complemented and uncomplemented form into the OR gate g_2. Thus, one of the inputs to the OR gate must be 1, and so its output must be 1. And yet, by definition, $\perp$ designates an 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] .

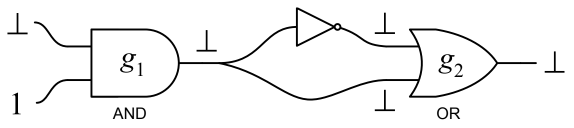


Figure 2.8: An illustration unknown/undefined values $\perp$.

In the analysis, we track the propagation of well-defined signal values. Once a definite value is assigned to an internal wire, this value persists for the duration (so long as the input values are held constant). For any input assignment, a circuit reaches a so-called *fixed point* in the ternary framework: a state where no further updates of controlling values are possible. This fixed point is unique [26]. We adopt the following definition.

[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 2.8 allows us to invalidate circuits where this could be a concern

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 1 *Consider the circuit shown in Figure 2.9, 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 2.10. The outcome for all eight cases is shown in Figure 2.11. We conclude that the circuit is not combinational.*

2.2.1 Gate Level vs. Functional Level Analysis

The algorithms and concepts presented in the begining of this chapter 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 [27]

Figure 2.12 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 2.8). In the case were $b = \perp$, it is possible that b is some value between 1 and 0, and in this

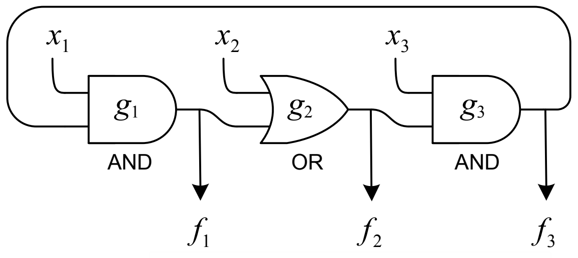


Figure 2.9: A cyclic circuit that is not combinational.

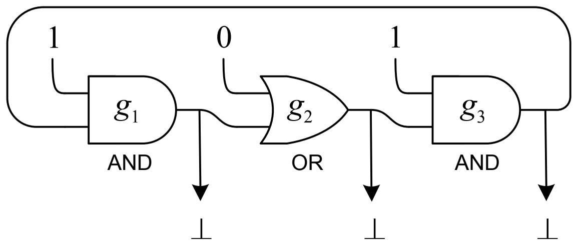


Figure 2.10: The circuit of Figure 2.9 with $x_1 = 1$, $x_2 = 0$ and $x_3 = 1$.

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

Figure 2.11: Analysis of the circuit in Figure 2.9.

case the mapped circuit shown in Figure 2.12 will evaluate to a different value than the unmapped function $f(a, b, c)$.

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 the beginning of this chapter, analysis and synthesis 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. Later, we explore methods of mapping and analyzing cyclic circuits at the level of gates. 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.

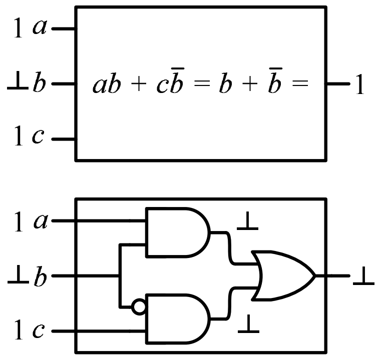


Figure 2.12: The function $ab + c\bar{b}$ and a gate level implementation.

2.3 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, $0 \leq i \leq n-1$. (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 [28]. An important step at the technology-independent stage is the task of structuring *functional dependencies.* (With SOP representations, this step was called *substitution* or *resubstitution.*) In this step, node functions are expressed or re-expressed in terms of other node functions as well as the primary inputs.

For each node function, different choices might be available as dependencies yielding alternative expressions of varying cost. Throughout this chapter, 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,

$$f_1 = bcx + bdx + ab \qquad (2.2)$$

$$f_2 = abc\bar{x} + cx + d. \qquad (2.3)$$

Figure 2.13 shows four different expressions for the functions and the corresponding dependency graphs. Figure 2.13.a shows expressions for f_1 and f_2, both in terms of the primary input variables only. With a support set of $\{a, b, c, d, x\}$, the cost of both of these expressions is 5, so the total cost is 10.

Figures 2.13.b and 2.13.c show alternate expressions, obtained by introducing functional dependencies. In Figure 2.13.b, f_1 is expressed in terms of f_2 and $\{a, b, x\}$. Accordingly, the total cost is 9. In Figure 2.13.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 2.13.b; the ordering of $f_1 \sqsubseteq f_2$ would produce the expressions in Figure 2.13.c. Insisting upon an ordering means that we would have to choose one of these two results.

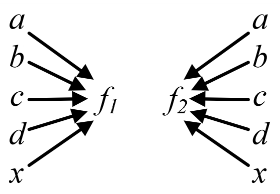
However, if we allow cyclic dependencies, we can find a better solution. Figure 2.13.d show expressions for f_1 and f_2 with support sets of $\{a, b, x, f_2\}$ and $\{c, d, x, f_1\}$, so a total cost 8. As the dependency graph in Figure 2.13.d illustrates, the functional dependencies are cyclic. Yet 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 2.14. 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

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

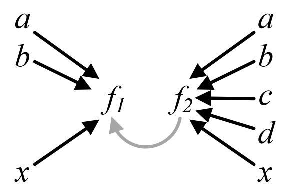
is evidently meaningless.

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



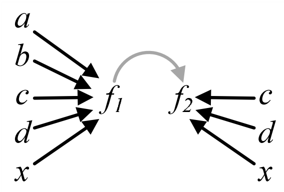
(a) $f_1(a, b, c, d, x)$ and $f_2(a, b, c, d, x)$

$$\begin{aligned} f_1 &= bxf_2 + ab \\ f_2 &= abc\bar{x} + cx + d \end{aligned}$$



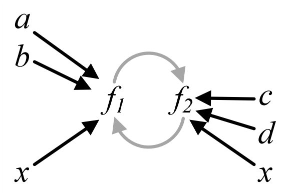
(b) $f_1(a, b, x, f_2)$ and $f_2(a, b, c, d, x)$

$$\begin{aligned} f_1 &= bcx + bdx + ab \\ f_2 &= cx + cf_1 + d \end{aligned}$$



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

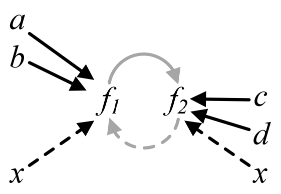
$$\begin{aligned} f_1 &= bxf_2 + ab \\ f_2 &= cx + cf_1 + d \end{aligned}$$



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

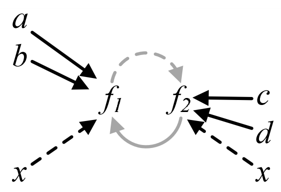
Figure 2.13: Four different implementations of two functions, f_1 and f_2, of five variables a, b, c, d, and x.

$$\begin{aligned} f_1 &= ab \\ f_2 &= cf_1 + d \end{aligned}$$



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

$$\begin{aligned} f_1 &= ab + bf_2 \\ f_2 &= c + d \end{aligned}$$



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

Figure 2.14: 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.

2.3.1 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 [1]. In [23], the method was applied to the problem of synthesizing functional dependencies. Broadly, the strategy is to formulate an instance of Boolean satisfiability (SAT) that asks whether or not a target function can be implemented with a certain support set. A proof of unsatisfiability, returned by a SAT solver, is converted into a circuit that computes the target function. We give a brief review of the method here, noting that in its current form, it is only applicable to acyclic orderings. In the next section, we generalize the method to cyclic orderings.

The method constructs a miter, as shown Figure 2.15. 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, $\ldots$, x_n (referred to as X) are the primary inputs to f_0 *Left*. The primary inputs x_0*, x_1*, $\ldots$, 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 [23], a method is proposed for finding the dependency function h using Craig interpolation. The underlying details of the approach to computing h are not important; it is only important that the reader understands that if the ON set of a function $f(f_0,f_1,\ldots,f_n)^1$ is disjoint from the OFF set $f(f_0,f_1,\ldots,f_n)^0$ then a function h can be computed by generating an interpolant from a SAT instance that is similar to that in Figure 2.15.

Using Craig interpolation to generate functional dependencies has proven to be 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.

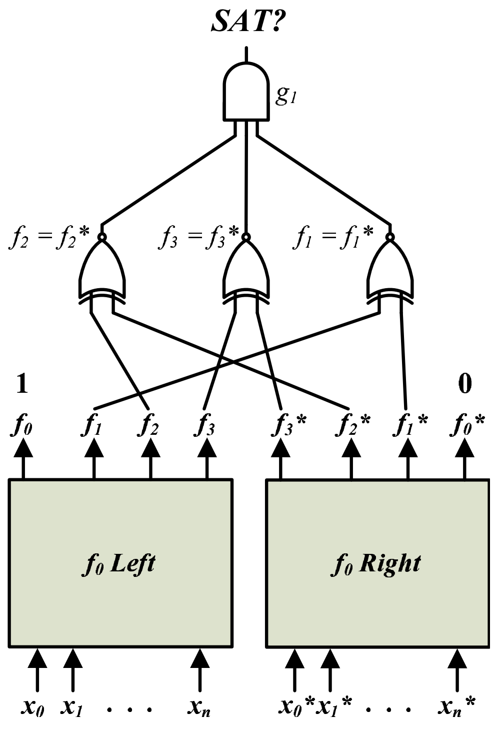


Figure 2.15: A miter that checks to see if f_0 can be specified in terms of f_1, f_2, and f_3.

2.3.2 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 5.3 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 2.17 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
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 1. In Proposition 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 5.3. 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 2.18 and to the example listed after the proof to help make sense of the constructs in Proposition 1.

Proposition 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.*

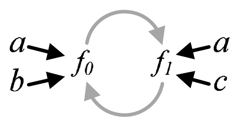
1. *Every row in R^1 and R^2 has the same values for its primary input variables.*

2. *Let R^1_i and R^2_i be the ith row in R^1 and R^2 respectively. $\forall i \in \{0, 1, \ldots, n-1\}$, If the value of $f_i(R^1_i)$ is the same as $f_i(R^2_i)$ 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 $f_i(R^1_i)$ differs from $f_i(R^2_i)$.*

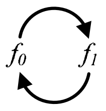
$a\,b\,f_1$	f_0
0 0 0	1
0 0 1	1
0 1 0	0
0 1 1	1
1 0 0	0
1 0 1	0
1 1 0	1
1 1 1	1

$a\,c\,f_0$	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



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

Figure 2.16: The truth tables for two functions. The cyclic dependency graph containing these two functions is not combinational.



$$f_0 = f_1 \tag{2.4}$$
$$f_1 = \bar{f}_0 \tag{2.5}$$

Figure 2.17: The dependency graph for the functions in Figure 5.3 for the assignment: $a = c = 0, b = 1$. The dependency graph is not combinational.

Proof 1 *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.

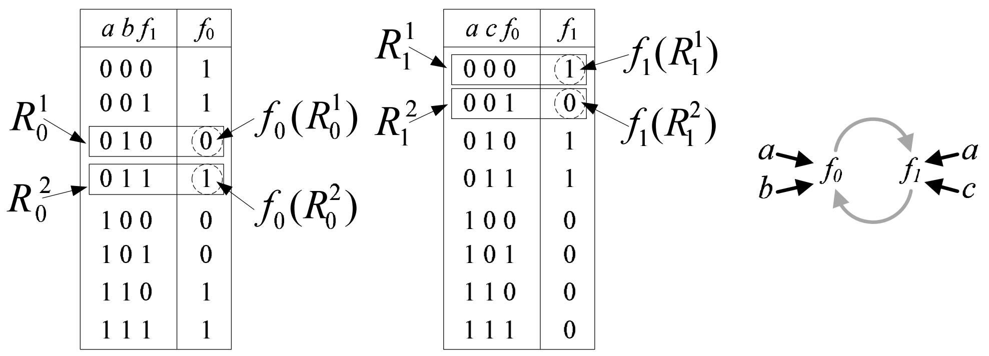
The second condition asserts that if the output value of some function f_i is the same between two rows R_i^1 and R_i^2, then the variable corresponding to this function in other rows of R^1 and R^2 must also be assigned this value. Essentially this condition guarantees that if the value of some function 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 2.2, 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.

$\square$



$a\ b\ f_1$	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

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

Figure 2.18: 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.

Because the example in Figure 5.3 is not combinational, there must be some choice of pairs of rows (R^1 and R^2) that satisfies the three conditions in Proposition 1. As stated above, 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 2.18 illustrates this specific selection of rows for the example in Figure 5.3.

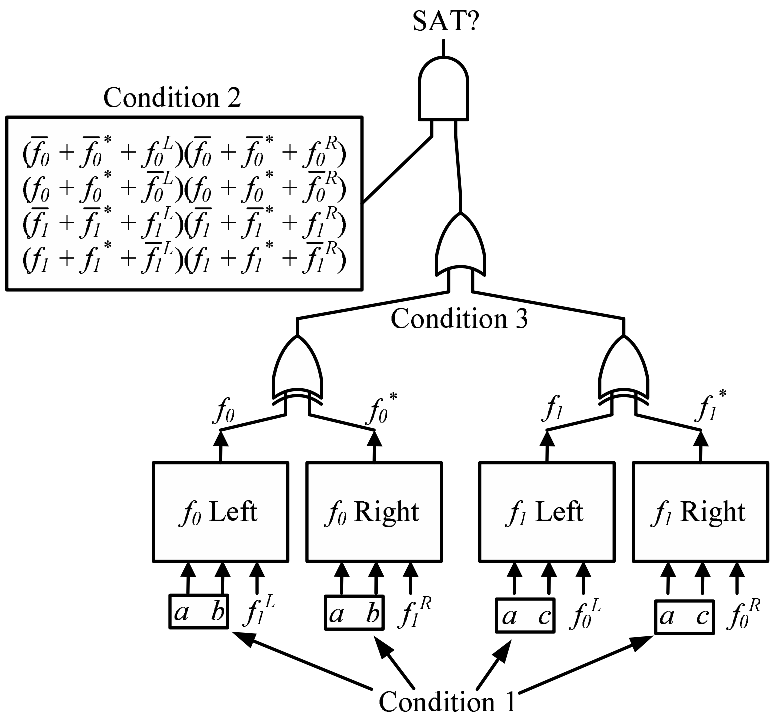

SAT?
Condition 2
$(\overline{f_0} + \overline{f_0}^* + f_0^L)(\overline{f_0} + \overline{f_0}^* + f_0^R)$
$(f_0 + f_0^* + \overline{f_0}^L)(f_0 + f_0^* + \overline{f_0}^R)$
$(\overline{f_1} + \overline{f_1}^* + f_1^L)(\overline{f_1} + \overline{f_1}^* + f_1^R)$
$(f_1 + f_1^* + \overline{f_1}^L)(f_1 + f_1^* + \overline{f_1}^R)$
Condition 3
f_0
f_0^*
f_1
f_1^*
f_0 Left
f_0 Right
f_1 Left
f_1 Right
a b f_1^L
a b f_1^R
a c f_0^L
a c f_0^R
Condition 1

Figure 2.19: A SAT instance that verifies whether or not the functions described in Figure 5.3 are combinational.

Craig interpolation provides an implementation for each target function in a dependency graph [23]. Given this implementation, a SAT instance can be formulated that is satisfiable if and only if the three conditions above are met. A circuit whose satisfiability indicates that these three conditions are met for the functions in Figure 5.3 is shown in Figure 2.19.

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 1). Additional logic is added that computes the *OR* of the *Exclusive OR* of each copy of each function (satisfying Condition 3 of Proposition 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 non-combinational.

2.3.3 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 2.3.1). 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

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.

$$c_1 = \prod_{i=0}^{n-1}(CNF_i^L(X, F^L) \leftrightarrow f_i)(CNF_i^R(X, F^R) \leftrightarrow f_i^*) \tag{2.6}$$

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

$$c_3 = \sum_{i=0}^{n-1}(f_i \oplus f_i^*) \tag{2.7}$$

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.

$$c_2 = \prod_{i=0}^{n-1}(\bar{f}_i + \bar{f}_i^* + f_i^L)(\bar{f}_i + \bar{f}_i^* + f_i^R)(f_i + f_i^* + \bar{f}_i^L)(f_i + f_i^* + \bar{f}_i^R) \tag{2.8}$$

Figure 2.20 shows a graphical representation of the general SAT instance for n functions of m variables[2] . Similarly to Figure 2.19, the conditions stated in Proposition 1 are shown in this figure as well.

Proposition 2 *Some choice of R^1 and R^2, for some set of functions satisfies the three conditions in Proposition 1 if and only if $(c_1)(c_2)(c_3)$ is satisfiable.*

Proof 2 *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 must be assigned the same value in every copy of every function. This corresponds to Condition*

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

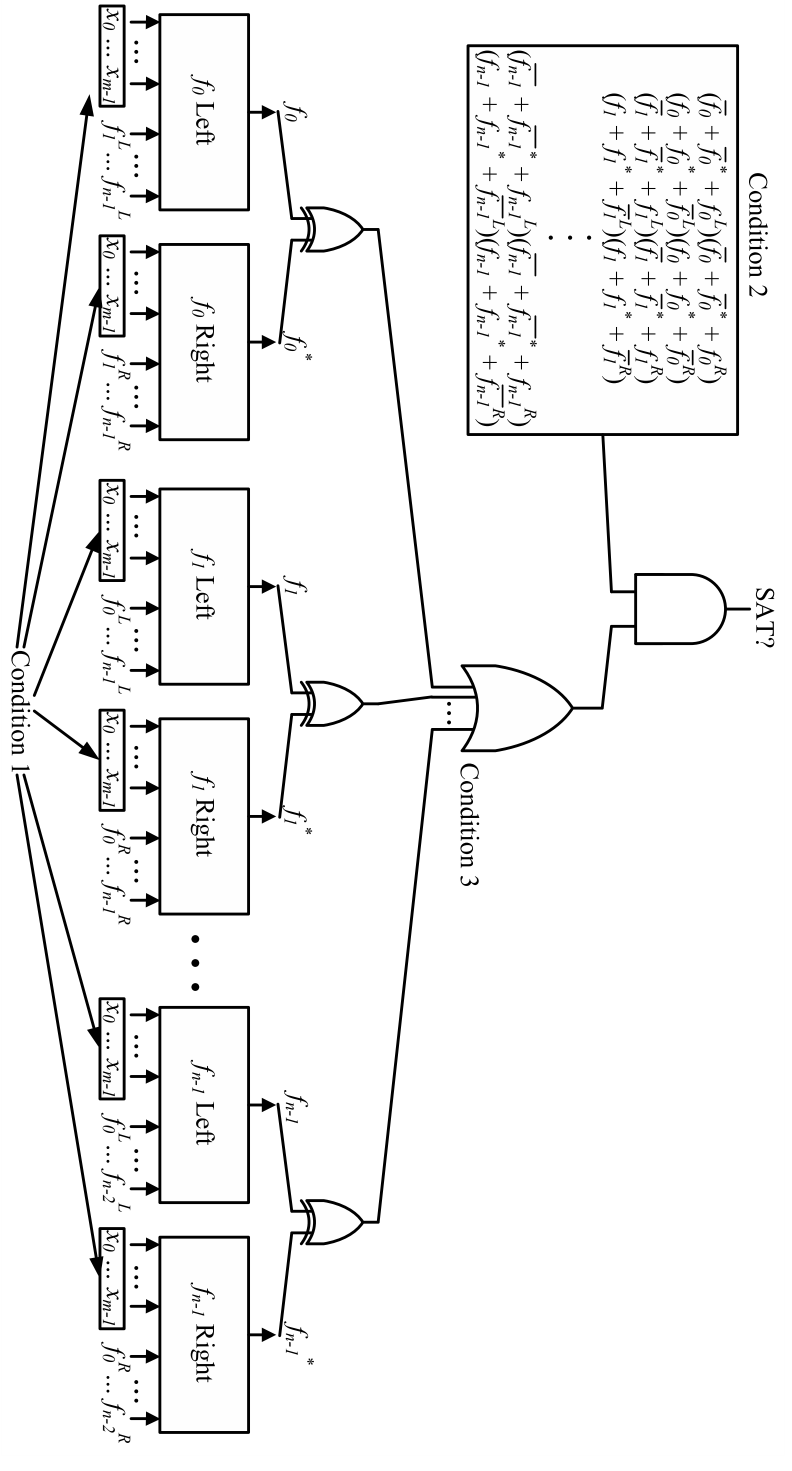


Figure 2.20: A SAT instance that checks if the set of functions $F = \{f_0, f_1, \ldots, f_{n-1}\}$ of variables $X = \{x_0, x_1, \ldots, x_{m-1}$ is combinational.

1 in Proposition 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 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 1. If the SAT instance $(c_1)(c_2)(c_3)$ is satisfiable, then all the conditions of 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 1 can never be simultaneously satisfied, and the network is deemed combinational.

$\square$

2.4 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 2.21. The routine "Synthesis" receives a set of Boolean functions as arguments. It first constructs a list of possible support sets for each function. Initially, it chooses a dependency graph containing the smallest possible support set for each function. This solution, as well as the list of possible support sets for each function, is sent to the "BreakDown" routine.

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

A visual illustration of the synthesis algorithm is shown in Figure 2.22. 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.

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

```
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 $BreakDown(Functions, DepGraph, SupportSetsList)$
```

Figure 2.21: 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 $\perp$.

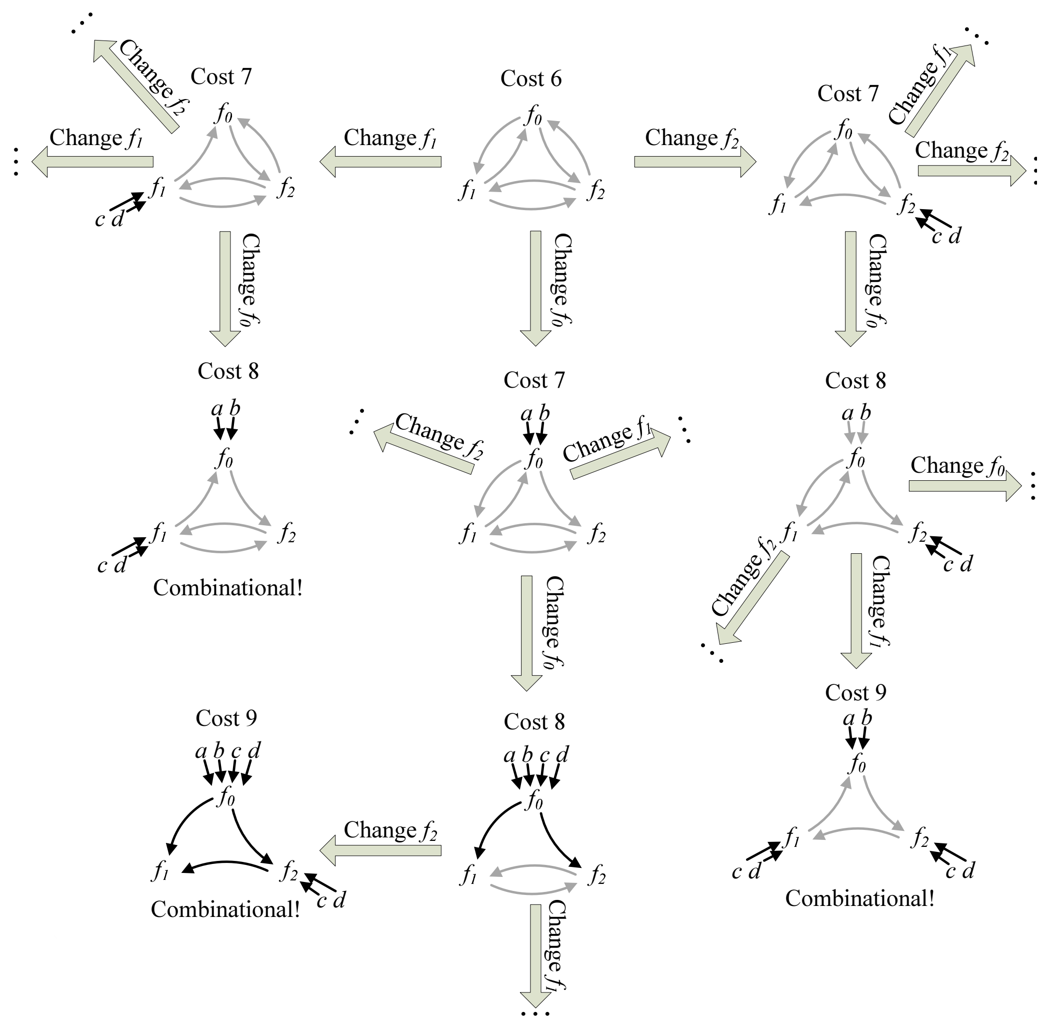


Figure 2.22: An illustration of the synthesis algorithm on an example consisting of 3 functions and 4 primary input variables. The thin gray arrows indicate cyclic dependencies in the dependency graphs. Some branches are omitted for clarity, as indicated by "…".

2.4.1 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 [29, 30]. 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 2.23. 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 removed, the resulting support set could not be used to represent the target function. In the experiments run in Table 2.5 of Section 2.10 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.

2.5 Implementation and Results

We present two sets of synthesis results on standard benchmarks [31]. 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 [32]. CYCLIFY is based on an earlier tool, Berkeley SIS [20], 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 2.4. (Instead of support set size, it uses literal counts as its cost function.) For Table I, we selected benchmarks where CYCLIFY produced cyclic solutions. Before reading these circuits into ABC, dummy primary inputs were introduced at the feedback locations (implicitly removing the cycles). The circuits were then run through 10 iterations of `compress2`,

```
**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 $SupSets$
```

Figure 2.23: 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 2.3.1 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.

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 2.1: Results of circuits synthesized with CYCLIFY and then optimized with ABC.

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 [8], based on symbolic event propagation, to compute the delay. For the acyclic circuits, we compute the delay as the longest path from the primary outputs to the primary inputs in the AIG. As Table I demonstrates, introducing cyclic dependencies yields significant reductions in area as well as delay.[4] 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 2.21 was implemented in Berkeley ABC [32]. The SAT solver used was MiniSAT [33]. 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.

[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 [8].

Synthesis Results							
Benchmark	PIs	POs	Orig AIG Size	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

Table II lists benchmarks that were run through the synthesis routine described in Section 2.4. The algorithm generated support sets for each of the benchmarks with primary output functions expressed in terms of other primary output functions and primary inputs. (For benchmarks that had less than 40 primary outputs, additional primary outputs were added to intermediate nodes until the benchmark contained exactly 40. This was done to increase the number of possible dependency graphs.) We ran the *BreakDown* procedure described in Section 2.4 until either 40 combinational solutions were found, or until a total of 200 dependency graphs were explored and none of these were deemed to be combinational. Table II reports results for the smallest cyclic and acyclic representations that were found.

The columns “PIs” and “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 “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 vs. AIG size). Modern FPGA mapping algorithms have a similar aim as the synthesis methodology presented in this chapter; 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 chapter, increasing the space of possible solutions that these tools can produce.

2.6 Gate-Level Combinational Analysis Algorithm

2.6.1 Overview

As mentioned in Section 2.2.1, there are subtle differences between analyzing cyclic circuits on a network-level versus a gate-level. The remainder of this chapter addresses the problem of analyzing and mapping cyclic circuits to gate-level implementations. The algorithm, which we refer to as the *analysis algorithm* from here on out, is described in detail in Section 2.6. In rough outline, the steps are:

- We find a *feedback arc set*, that is to say, wires that we can cut to make the circuit acyclic.
- We introduce new *dummy variables* at these cut locations.

- We encode the entire computation of the circuit in terms of ternary-valued logic: zeros, ones and "undefined" values. These ternary values are encoded with "dual-rail" binary values: zero is encoded as $[0, 0]$, one as $[1, 1]$, and "undefined" as either $[1, 0]$ or $[0, 1]$.

- We set up an acyclic circuit that answers the question: given undefined values for the dummy variables (in the ternary encoding) is there any input assignment that produces undefined values (again in the ternary encoding) at the output? This circuit forms the SAT question.

In the case where the circuit in question is indeed combinational, the SAT solver returns an answer of *UNSAT*. If some assignment of the circuit's primary inputs result in non-combinational behavior, the solver returns an answer of *SAT* and it also provides a satisfying assignment. As we discuss in the next section, we can make use of this satisfying assignment. The flow of this analysis algorithm is illustrated in Figure 2.24.

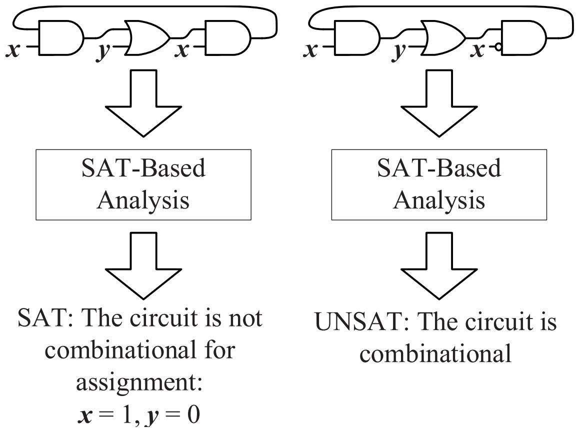

x
y
x
x
y
x
SAT-Based Analysis
SAT-Based Analysis
SAT: The circuit is not combinational for assignment: $x = 1, y = 0$
UNSAT: The circuit is combinational

Figure 2.24: An illustration of how SAT-based analysis works. If the circuit is combinational, the SAT solver returns an answer of *UNSAT*. If the circuit is not combinational, it returns an answer of *SAT* and it provides a satisfying assignment.

The complexity of the analysis is dependent on the runtime of the SAT solver. Setting up the circuit for the SAT instance is comparatively trivial: it entails but a single pass through the circuit to compute a feedback arc set. The circuit for the SAT question is larger than the original circuit: for every gate in the original circuit, approximately six gates are needed to formulate the ternary-valued encoding. Given the efficiency of SAT solvers, this is a winning strategy in spite of the increase in the circuit's size. In 2.10, we compare runtimes on benchmark circuits for this method compared to binary decision diagram (BDD)-based methods.

2.6.2 Analysis Algorithm

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

1. The first step is to find wires that, if cut from the circuit, would break all the cycles. Such a set can be found through a simple depth-first search [34]. Finding the smallest set of wires is, of course, difficult: this is the *minimum feedback arc set* problem, known to be NP-hard. The fewer wires in the cut set, the fewer dummy variables that we introduce and hence the smaller the size of the SAT instance; however, given the efficiency of SAT solvers, spending time finding the very best cutsets may not be expeditious; the construction will work for any cut set.

2. The next step is to convert every gate in the circuit into a corresponding module that operates on the *dual-rail* encoded ternary logic. In this step, each wire in the original circuit is replaced by a pair of wires. The four possible values of these wires code for ternary values 0, 1, and $\perp$. We chose the encoding scheme given in Figure 2.25. In this scheme, two zeros on each wire codes for logic 0, two ones on each wire codes for logic 1, and the remaining two values code for $\perp$. Consider the encoding for an AND operation on ternary-valued inputs a and b. We use pairs of

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

Figure 2.25: Dual-rail encoding scheme for ternary values. The left two columns show the value for each wire in the dual-rail encoding, and the right most column shows the corresponding ternary value.

inputs for each value: a_0 and a_1 corresponding to a, and b_0 and b_1 corresponding to b. The outputs are encoded by the functions:

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

Other gates, such as OR, NAND, NOR, etc., can be implemented similarly. The NOT operation is particularly easy – we simply complement the bit on each rail.

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

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

5. For every pair of dummy variables, we set up an **equivalence checker**: this is a module that evaluates to 1 if and only if the value assigned to dummies agrees with the value computed by the circuit at the cut location. The circuit may be computing $\perp$, encoded as (01) or (10); in this case, the equivalence checker evaluates to 1 if the dummies have *different* values. Call the output of the equivalence

checker x_i for each cut location i. For dummy variables d_1 and d_2 and gate outputs f_1 and f_2, the logic for the equivalence checker is

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

6. For every pair of dummy variables, we set up a **$\perp$-checker**[5] : this is simply an exclusive-OR gate on the two dummies; it evaluates to 1 if and only if the dummies are assigned different values (encoding $\perp$). Call the output of the $\perp$-checker y_i for each cut location i.

   Note that rather than introducing dummy variables, equivalence checkers, and $\perp$-checkers into the SAT circuit, we could instead append the logically equivalent clauses to the circuit's CNF formula representation to produce the same results. By introducing dummy variables and equivalence gates into the SAT circuit, we are implicitly adding these clauses to the CNF formula. Many modern SAT techniques take advantage of circuit structure alongside the circuit's CNF representation in order to find a result faster [35]. The latter method would not make use of the structural information that dummy variables, equivalence checkers and $\perp$-checker add to the circuit.

7. Finally, as illustrated in Figure 2.26, the output of the circuit is the AND of the AND of the x_i's and the OR of the y_i's.

Consider the circuit in Figure 2.27, consisting of four NAND gates. Note that there are four cycles. By inserting dummy variables d and e, we obtain the circuit in Figure 2.28 (This circuit is acyclic). Next, we replace each gate with a dual-rail

[5] As discussed in Section 2.2, we are using the stringent definition of combinationality here: all gates, not only the outputs, must eventually produce definite values. For the less stringent definition, **$\perp$-checkers** only need to be included at the primary outputs of the circuit.

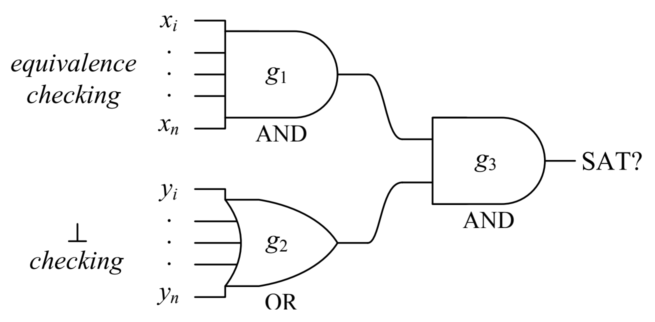


Figure 2.26: Constructing the SAT instance.

version; we feed in pairs of dummy variables, d_0, d_1, and e_0, e_1, corresponding to each of the previous dummy variables; we double the primary inputs a and b; we add two equivalence-checkers, producing x_0 and x_1; we add two $\perp$-checkers (i.e., exclusive-OR gates) producing y_0 and y_1; and we add three logic gates g_1, g_2, and g_3 to form the final output.

This circuit, shown in Figure 2.29, forms the SAT instance with six primary input variables: a, b, d_0, d_1, e_0, and e_1. We see that for a primary input assignment of $a = b = 1$, $d_0 = \bar{d}_1$, and $e_0 = \bar{e}_1$, $\perp$ values remain on each pair of rails on the inputs of the equivalence checkers, indicating that the inputs to each are equivalent; so x_0 and x_1 produce outputs of 1; y_0 and y_1 produce outputs of 1 as well; so the final output is 1. Therefore, the SAT instance is *satisfiable* and the circuit is *invalid.* Indeed, $a = b = 1$ are non-controlling values for the NAND gates, so this is the outcome that we expect.

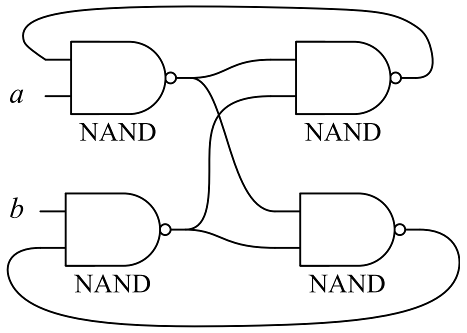


Figure 2.27: A cyclic circuit

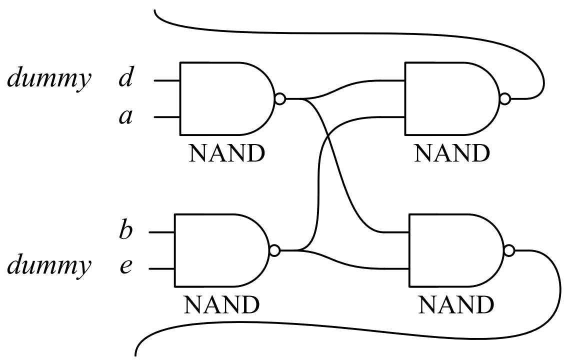


Figure 2.28: The circuit in Figure 2.27 with cycles broken.

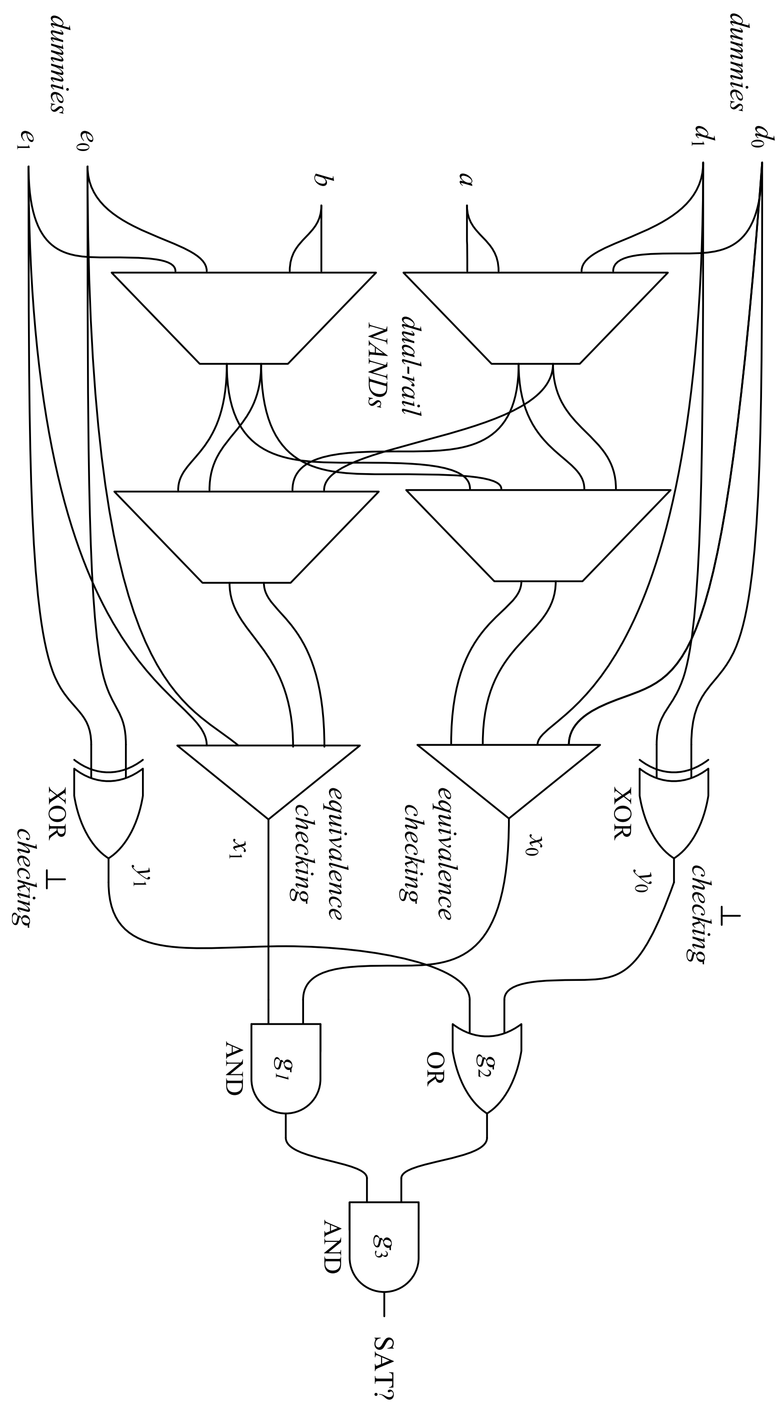


Figure 2.29: The SAT circuit corresponding to the cyclic circuit in Figure 2.27.

2.7 Proof of Correctness of Analysis

Here we prove the correctness of our SAT-Based analysis algorithm for gate-level cyclic circuits. We first prove the forward direction: Our algorithm always correctly identifies cyclic circuits which are combinational. We then prove the backwards direction: Our algorithm always correctly identifies cyclic circuits which are not combinational.

Proposition 3 *A SAT circuit that evaluates to 1 never corresponds to a* valid *cyclic circuit.*

Proof 3 *Indeed, if a SAT circuit evaluates to 1, then both the gates g_1 and g_2 are at 1. If g_1 is at 1, then the corresponding values in the cyclic circuit are at a* fixed point*; however, if g_2 is at 1, then some of the values in the fixed point are $\perp$. By definition, the cyclic circuit is invalid.* □

Proposition 4 *Every invalid cyclic circuit translates into a SAT circuit that evaluates to 1 for a specific input assignment.*

Proof 4 *Indeed, if the circuit is invalid then it has a fixed point with $\perp$ values on some of the wires of the cut set. (A fixed point that contains $\perp$ values* somewhere *must also have these on the cut set.) In the SAT circuit, consider such an input assignment: assign the dummy values that correspond to the values from the fixed point; this ensures that g_1 is at 1. Because some of these values are $\perp$, g_2 is also at 1 and so the SAT circuit evaluates to 1.* □

2.8 Mapping Cyclic Combinational Circuits

2.8.1 Overview

In our synthesis flow, we introduce cycles at the level of functional dependencies in a Boolean network. These designs are then mapped to gates from a library. Cyclic designs must be validated both at the level of functional dependencies and then again after mapping. This is necessary because mapping sometimes breaks the validity: designs that are combinational at the functional level get mapped onto designs that are not combinational at the gate level. This was first observed in [27].

Consider the functions in Figure 2.30. The three functions form a cycle: f depends on h, h depends on g, and g depends on f. The reader can verify that for all assignments of the primary inputs a and b, the functions f, g, and h evaluate to definite Boolean values, so we consider this specification to be combinational. Figure 2.31 shows gate-level mappings for the three functions. Since the functional-level specification is combinational, one might assume that one can simply wire these gate-level mappings together, as shown in Figure 2.32. But this doesn't work: trying input combinations, we see that the assignment $a = b = 1$ does not result in definite values for the outputs f, g, and h. The individual gate mappings for the functions are correct, but the resulting circuit is not combinational.

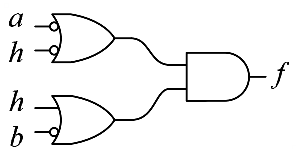
The problem arises with the mapping for f. At the functional level, input values of $a = b = 1$ result in $f = (h)(\bar{h}) = 0$. However, at the gate level, the initial values on internal wires are not only unknown but possibly undefined. (These could have voltage values that are not unequivocally 0 or 1 but possibly some value in between.) Here the value of h is undefined, so the value of f is undefined. As we explain in Section 2.2, the validity of a circuit can be established with ternary-valued simulation.

This chapter presents a technique for modifying the mapping of cyclic circuits to ensure that they are combinational, based on the results of SAT analysis. The circuit in Figure 2.32 can be fixed by adding additional logic, as shown in Figure 2.33.

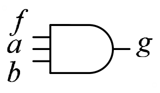
This additional logic can be generated from a set of input assignments that results in non-combinational behavior. Our SAT-based analysis provides exactly such satisfying assignments. For the circuit in Figure 2.32, SAT-based analysis returns the satisfying assignment $a = b = 1$. This assignment is used to generate the additional logic in Figure 2.33. The reader can verify that the circuit in this figure is combinational.

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

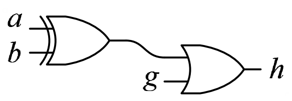
Figure 2.30: A cyclic specification of three Boolean functions, f, g and h. These evaluate to definite Boolean values for all assignments of the inputs a and b.



$$f = (\bar{a} + \bar{h})(\bar{b} + h)$$

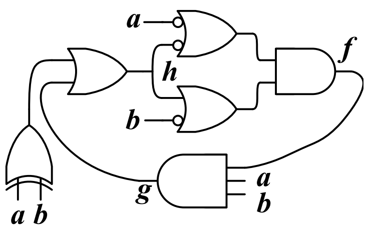


$$g = abf$$



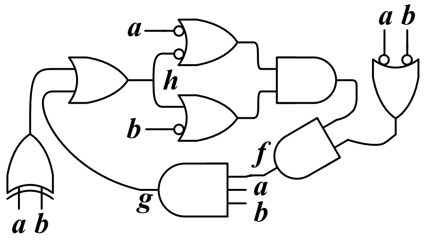
$$h = a \oplus b + g$$

Figure 2.31: Individual gate mappings for the functions in Figure 2.30.



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

Figure 2.32: The circuit obtained by assembling the mappings in Figure 2.31 together. It is *not* combinational.



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

Figure 2.33: The circuit in Figure 2.32 with additional logic. It is combinational.

2.8.2 Mapping Algorithm

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

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

Our method for synthesizing this additional logic is as follows.

1. Consider the functions at the cut locations in the unmapped circuit. For each such function f, create an empty list of products and an empty list of sums. Map the circuit to gates. Perform SAT-based analysis to determine if the mapped circuit is combinational.

2. If the SAT solver returns $UNSAT$, skip to Step 4. If the SAT solver returns SAT, proceed to Step 3.

3. For each function f at a cut location, set the variables in f's support set to the corresponding values in the satisfying assignment. Then:

   - Let P be a product with literals corresponding to variables with definite values in f's support set. If f evaluates to 1 in the unmapped circuit, add P to f's list of products.
   - Let S be a sum with literals corresponding to the negation of the variables with definite values in f's support set. If f evaluates to 0 in the unmapped circuit, add S to f's list of sums.

   Add the following clause to the SAT instance created in Step 1: a clause that evaluates to 0 for the definite values among the variables in f's support set. Solve the SAT instance again and go back to Step 2.

4. For every function f at a cut location, minimize f's list of products and f's list of sums. In the minimization of the products, select a cover of all the partial assignments that evaluate to 1; in the minimization of the sums, select a cover of all the partial assignments that evaluate to 0.

5. After performing this minimization:

   - For each product P in f's list of products, replace the output of f by $f + P$ in the mapped circuit.
   - For each sum S in f's list of sums, replace the output of f by $(f)(S)$ in the mapped circuit.

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

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

The goal of this mapping algorithm is simply to try to fix circuits that are "close to correct" by adding a minimal amount of extra logic. Note that there might not be any unmapped functions that evaluate to a definite value in Step 3. In this case, there is no additional logic to add in Steps 4 and 5. Here the conclusion is that the mapping cannot be fixed; the explanation is that the functional-level specification was not combinational to begin with.

Example 2 *Consider again the circuit in Figure 2.32. If SAT-based analysis is performed on this circuit, the solver will return the satisfying assignment:* $a = b = 1$, $f = g = h = \perp$. *Apply this assignment to the unmapped circuit consisting of* f, g, *and* h. *Observe that, for this assignment,* f *in the unmapped circuit evaluates to* 0. *In the mapped circuit, attach an* AND *gate to the output of* f *that evaluates to 0 for the assignment* $a = b = 1$. *This fixes the mapping. The resulting circuit is shown in Figure 2.33.*

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

Example 3 *Consider a cyclic circuit that has been mapped to gates. Suppose that the support set of a function* f *in the circuit is* $\{a, b, c, d\}$. *Suppose that, after analyzing the circuit, it is found that the value* f *computed by the mapped circuit is* $\perp$ *for the following assignments. Suppose that, for each of these assignments,* f *evaluates to 1 in the unmapped circuit:*

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

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

In our experience, relatively few satisfying assignments are ever found for a circuit that needs its mapping fixed. Accordingly, exact methods such as Quine-McCluskey are a viable option [36]. Of course, heuristic methods or multi-level minimization could be used [28, 37]. Note, however, that the minimization in Step 4 is not traditional minimization in a *binary* context. Rather, the requirement is that terms in the sum cover the satisfying assignments in a *ternary* context. This is illustrated in the following example.

Example 4 *Consider a cyclic circuit that has been mapped to gates. Suppose that the support set of a function* f *in the circuit is* $\{a, b, c, d, e\}$. *Suppose that, after analyzing the circuit, it is found that the value of* f *in the mapped circuit is* $\perp$ *for the following assignments. Suppose that, for each of these assignments,* f *evaluates to 1 in the unmapped circuit:*

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

Here the variables a, b, and d, could be primary inputs or they could be other functions. Clearly, c and e are functions; primary inputs are never assigned $\perp$ values. As in the previous example, these assignments can be minimized to a smaller set. One might assume that assignments of c and e that are $\perp$ can be treated as if they were either 0 or 1 (i.e., treated as don't cares). Assuming this, the set would be minimized to $\{a\bar{c}d, bcd\}$. This would result in the additional logic shown in Figure 2.36. However, in Figure 2.36, we see that when $a = b = d = 1$ and $c = e = \perp$, the output of f is still $\perp$. So the fix did not work!

In Step 5 of the algorithm, a list of products is OR-ed with the output of a mapped function f. This set of products is meant to cover the partial assignments provided by the SAT solver, that cause f to evaluate $\perp$ when f should evaluate to 1. Call this set of partial assignments $\boldsymbol{A}$. Since the list of products is minimized via two level logic minimization, it only contains prime implicants [36]. The following proposition intends to show that minimizing this set of products is necessary for the correctness of the mapping algorithm.

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

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

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

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

Example 5 *Consider the set of assignments from Example 4. Applying Proposition 1 during two-level minimization, we obtain the set of products $\{a\bar{c}d, bcd, abd\}$. Unlike the previous example, the product abd is contained in the minimum representation for the partial assignments. Figure 2.37 shows a mapping when product abd is not removed from the two level minimization. For all the assignments listed in the table in the previous example, the newly mapped function behaves correctly in Figure 2.37.*

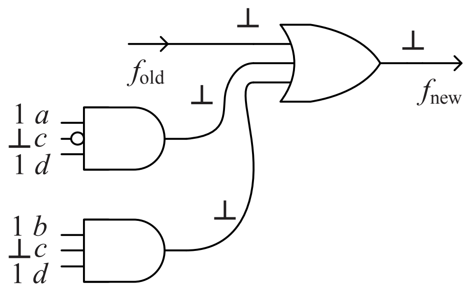


Figure 2.34: Example 4: A mapping fix without a product covering assignment $a = b = d = 1$, $c = e = \perp$.

Example 6 *Consider again the the function f from Example 3. Suppose that variable c is not a primary input but rather a function with the support set: f, g, and h. Where f is the function described in Example 3 and g and h are primary input variables. This circuit then implements two functions:*

$$f(c, a, b)$$

$$c(f, g, h)$$

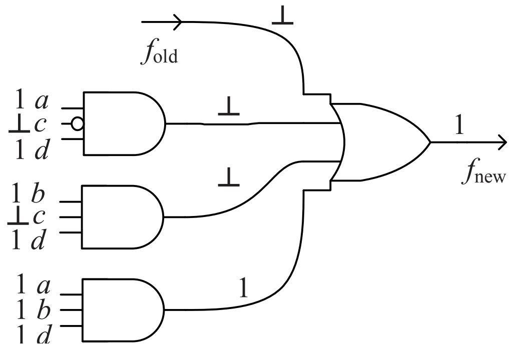


Figure 2.35: Example 5: A mapping fix that works.

There exists a cycle between functions f and c. It is possible that, for some assignment of the variables g and h, c may not evaluate to a definite value until the value of f is definite. If this is the case, then including c in the additional logic will not correct the circuit's behavior: the value of c will still be $\perp$ on the input of one of the additional gates. Because the value of c is not definite, the additional logic will not control the output of f for this assignment of a, b, and c.

Again, if all satisfying assignments from the analysis algorithm are considered and then optimized, then the values of c and b need not be considered when adding the additional logic to fix f, as was shown in the previous example. The output of f has to be OR*-ed with the product $\bar{a}$ to fix this circuit.* OR*-ing the output of f with $\bar{a}c$ and then* OR*-ing the output again with $\bar{a}\bar{c}$ will* ***not*** *correct the behavior of this circuit. The additional logic is shown in Figures 2.36 and 2.37. In Figure 2.36, the products $\bar{a}c$ and $\bar{a}\bar{c}$ are added to the output of f. The circuit does not behave combinationally: when c is $\perp$, f does not evaluate to a definite value. In Figure 2.37, the product $\bar{a}$ has been added. When a is 0, the input of the* OR *gate assumes a controlling value and so f evaluates to 1. Here the additional logic results in a combinational circuit.*

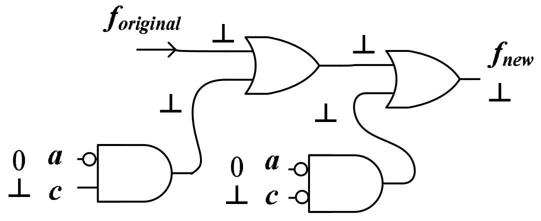


Figure 2.36: The incorrect fix in Example 4. The function f is not controlled for this assignment of a and c.

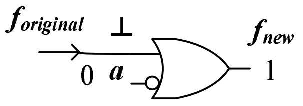


Figure 2.37: The correct fix Example 5. The function f assumes the correct value of 1 when a is 0.

2.9 Proof of Correctness for Mapping

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

Proposition 6 *(Does no harm with products.) Each product P that is* OR*-ed with f in Step 5 of the mapping algorithm never evaluates to $\perp$ when f evaluates to 0.*

Proof 6 *Each product P is a redundant product in the computation of f:* OR*-ing P with f does not expand the set of assignments that causes f to be 1. Consider a function f_{new}, where $f_{new} = f + P$. Because P is redundant, f_{new} must be equivalent to f. Therefore f_{new} cannot evaluate to $\perp$ while f evaluates to 0 (or else f_{new} and f would not be equivalent). This implies that P cannot be $\perp$ while f is 0.* □

Proposition 7 *(Does no harm with sums.) Each sum S that is* AND*-ed with f in Step 5 of the mapping algorithm never evaluates to $\perp$ when f evaluates to 1.*

Proof 7 *Each sum S is a redundant sum in the computation of f:* AND*-ing S with f does not expand the set of assignments that causes f to be 0. Consider a function f_{new}, where $f_{new} = (f)(S)$. Because S is redundant, f_{new} must be equivalent to f. Therefore f_{new} cannot evaluate to $\perp$ while f evaluates to 1 (or else f_{new} and f would not be equivalent). This implies that S cannot be $\perp$ while f is 1.* □

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

Proposition 8 *(Makes progress.) Each product (each sum) that is* OR*-ed (*AND*-ed) with the output of the mapped function f in Step 5 results in in a definite output for some assignment that otherwise produces $\perp$.*

Proof 8 *Each such product (sum) evaluates to 1 (0) for every partial assignment found in Step 3. Because each such product (sum) fans into the input of an OR (AND) gate that is attached to the output of f, the OR (AND) gate is forced to 1 (0) for every assignment found in Step 3.* □

Evidently, this algorithm must eventually halt because there are a finite number of input assignments. Of course, iterating through all the input assignments would entail an exponential number of steps. In practice, we have found that initial mappings are invariably "close to correct." We have not seen instances where there were more than 10 satisfying assignments that resulted in non-combinational behavior. (Recall that these are mappings that were produced from cyclic dependencies that were valid at the functional level.) Furthermore, we use incremental SAT for this step, so successive calls to the SAT solver return very quickly [38].

If the number of satisfying assignments in Step 3 becomes exceedingly large, then a heuristic choice can be made about when to terminate the mapping algorithm and discard the current circuit as "unfixable." Following, say a branch-and-bound approach, the synthesis routine would then try different cyclic configurations for functional dependencies and perform a new mapping [9].

2.10 Implementation and Results

We implemented the algorithms described in Sections 2.6 and 2.8 in the Berkeley ABC environment [32]. ABC invokes the "MiniSAT" SAT Solver [33]. We performed trials on cyclic circuits produced by our tool CYCLIFY on benchmark circuits in the IWLS collection [39]. (For circuits with latches, we extracted the combinational part.) We ran 10 iterations of the script `compress2` on both the cyclic versions produced by CYCLIFY as well as the original acyclic versions.

In the following tables, the size that is reported is the number of AND2 gates in an AND-inverter graph (AIG) representation. The runtimes for the new SAT-based analysis are compared to those of the previous BDD-based approach [8]. Trials were performed on an AMD Athlon 64 X2 6000+ Processor (@ 3Ghz) with 3.6GB of RAM running Linux. Only one core was utilized for the trials.

Table 2.2 lists benchmarks that mapped correctly. Table 2.3 lists benchmarks that needed additional logic to correct the mappings. The numbers reported in Table 2.2 include the time to:

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

(CYCLIFY provided a feedback arc set, so a depth-first search to find cut locations was not necessary.)

The "Gates" columns report the number of AND2 gates in the AIG. The "Size Ratio" column is calculated as "Gates Cyclic / Gates Acyclic."

The "Delay" columns report the delay for the cyclic and acyclic circuits. For the cyclic circuits, we use algorithm presented in [8], 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. 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."

The "Time Ratio" column is calculated as "Time SAT / Time BDD." As expected, we see that SAT-based analysis is considerably faster than BDD-based analysis – orders of magnitude faster for the larger circuits. We note that, for nearly every benchmark, the cyclic circuits have smaller area and smaller delay than their acyclic counterparts.

Table 2.3 lists benchmarks that needed to have their mappings corrected. The cyclic version of the circuit `table3` was initially larger than its smallest acyclic representation. For the circuit `dk16`, we ran both the acyclic and cyclic versions, obtained after remapping, through an additional 10 iterations of `compress2`. The remapped cyclic circuit still was slightly larger.

Unfortunately, the set of cyclic benchmarks we have to test is quite limited. All of the circuits were produced by CYCLIFY, implemented in the Berkeley SIS framework [9]. As we have noted, the size of benchmarks that CYCLIFY can tackle is limited by the underlying data structures (SOP and BDD representations). This chapter is part of our effort to develop more scalable techniques for synthesis.

For Table 2.4, we generated random Boolean functions to test our mapping algorithm. For each choice of different numbers of inputs and outputs, we randomly generated 300 circuits. The column "Cyclic Solutions" lists the number of circuits where a cyclic solution was found. The columns "Gates Cyclic" and "Gates Acyclic" list the average number of gates in the smallest cyclic and acyclic implementations of the circuit. The "Gates Cyclic" column includes the additional gates that were added to fix incorrect mappings. "Remapped" lists the number of circuits that needed to have their mappings fixed. "Vectors" and "Gates Added" list the average number of satisfying assignments returned from the SAT solver and the average number of gates added to the mapping. These two fields were only averaged over circuits whose initial mapping needed to be fixed. Finally, the last column gives the ratio of the average number of gates in the smallest cyclic and acyclic implementations that were found.

While few of the benchmark circuits required fixes to their mappings, between 10% and 30% of the randomly generated circuits required such fixes. The size reduction of the cyclic versions of the random circuits was, in general, not as significant as the size reduction of the benchmark circuits presented in Table 2.2. Perhaps this was because we were impatient: we set a relatively small timeout when synthesizing the circuits in Table 2.4 compared to the timeout when synthesizing the circuits in Tables 2.2 and 2.3.

2.11 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 [15]. 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

Runtimes (Mapping Initially Correct)									
Benchmark Name	Gates Cyclic	Gates Acyclic	Delay Cyclic	Delay Acyclic	Time BDD (s)	Time SAT (s)	Size Ratio	Delay Ratio	Time Ratio
bbsse	90	96	5	8	0.08	0.01	0.94	0.63	0.13
bw	110	183	9	9	0.02	< .01	0.6	1	-
clip	113	181	5	9	0.02	0.01	0.62	0.56	0.5
cse	128	152	6	9	0.23	0.01	0.84	0.67	0.04
duke2	309	301	11	11	0.49	0.06	1.03	1	0.12
ex1	205	210	14	8	0.26	0.03	0.98	1.75	0.12
ex6	61	116	8	7	< .01	0.01	0.53	1.14	-
inc	87	115	6	8	< .01	< .01	0.76	0.75	1
planet	381	419	7	9	0.25	0.06	0.91	0.78	0.24
planet1	377	433	7	9	0.13	0.05	0.87	0.78	0.38
pma	167	161	5	8	0.17	0.02	1.03	0.63	0.12
s1	254	339	6	11	4.92	0.05	0.75	0.55	0.01
s298	1806	1823	7	14	106.62	2.07	0.99	0.50	0.02
s386	91	102	5	7	0.02	0.01	0.89	0.71	0.5
s510	189	199	5	9	0.03	0.03	0.95	0.56	1
s526	129	135	9	9	0.01	0.02	0.96	1	2
s526n	130	117	8	10	< .01	0.02	1.11	0.80	-
s1488	431	500	9	9	0.34	0.07	0.86	1	0.21
sse	87	102	5	8	0.02	0.01	0.85	0.63	0.5
styr	344	380	8	10	0.59	0.06	0.91	0.80	0.1
table5	686	639	8	13	50.32	0.28	1.07	0.62	0.01

Table 2.2: Runtime comparison for circuits whose initial mapping was combinational

Runtimes (Mapping Fixed)						
Benchmark Name	Added Gates	Delay Cyclic	Delay Acyclic	Gates Cyclic	Gates Acyclic	Size Ratio
5xp1	6	8	8	92	97	.94
table3	26	10	13	833	771	1.06
dk16	17	5	9	208	199	1.04

Table 2.3: Runtime comparison for circuits whose initial mapping was not combinational.

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

Table 2.4: Results for randomly generated functions of five variables.

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.)

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. In the next chapter we propose 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.

Chapter 3

Reduction of Interpolants for Logic Synthesis

3.1 Introduction

As we discussed in the previous chapter, Craig Interpolation has been proposed for synthesizing functional dependencies in combinational logic [23]. For this application, a SAT instance is created to answer the question of whether a target function can be implemented with a specified support set or not. If the target function can be implemented in terms of the specified support set, then the SAT instance is unsatisfiable. The SAT instance is partitioned into two sets of clauses that only have variables in the specified support set in common. An interpolant is generated from the proof of unsatisfiability. The interpolant provides an implementation of the target function in terms of the support set.

While generating functional dependencies in this manner is quick and scalable, it generally does not yield optimal (or even very good) results. These methods can work effectively when mapping to a technology where the complexity of implementing a function is based on the size of the support set (like FPGAs). However, when mapping to

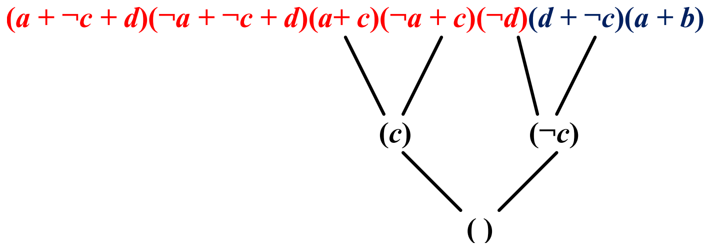

$(a + \neg c + d)(\neg a + \neg c + d)(a + c)(\neg a + c)(\neg d)(d + \neg c)(a + b)$
(c)
$(\neg c)$
$()$

Figure 3.1: A resolution proof from an unsatisfiable CNF formula. Clauses of $\boldsymbol{A}$ are shown in red while clauses of $\boldsymbol{B}$ are shown in blue

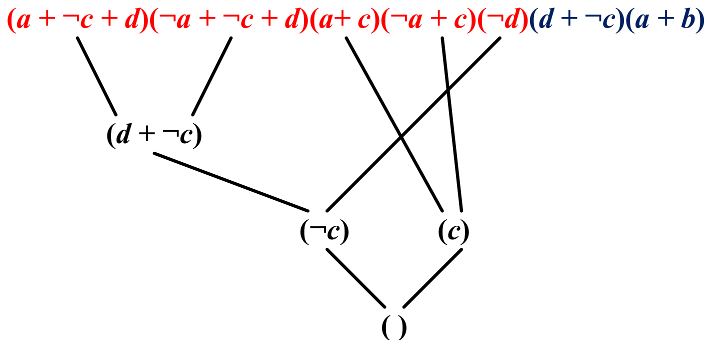

$(a + \neg c + d)(\neg a + \neg c + d)(a + c)(\neg a + c)(\neg d)(d + \neg c)(a + b)$
$(d + \neg c)$
$(\neg c)$
(c)
$()$

Figure 3.2: A different resolution proof from the same unsatisfiable CNF formula. Clauses of $\boldsymbol{A}$ are shown in red while clauses of $\boldsymbol{B}$ are shown in blue

primitive gates (ANDs, ORs, NOTs, etc...) the structure of the interpolants can be overly large and redundant.

The methods to derive interpolants from unsatisfiable SAT instances proposed in [1] and [40] follow fixed trajectories based on the resolution proof generated by a SAT solver. However, $\boldsymbol{I}$ is an over-approximation of the variable assignments that cause $\boldsymbol{A}$ to be true; there may be many different valid implementations for $\boldsymbol{I}$. Also, there may be many ways to prove that an unsatisfiable instance of SAT is indeed unsatisfiable. Consider the two proofs of unsatisfiability shown in Figures 3.1 and 3.2. Both proofs start with the same original leaf clauses and the same clause partitions $\boldsymbol{A}$ and $\boldsymbol{B}$. However, the proofs use a different series of resolutions to derive the empty clause. Different proofs may result in different interpolants. The size of the interpolant correlates with the size of the circuit implementation of the target function. Using the interpolant generation algorithm proposed in [1], the interpolants for the clause partition in Figures 3.1 and 3.2 result in implementations with 4 gates and 8 gates, respectively. Even with small problem sizes, poorly structured resolution proofs can result in overly complex interpolants. When synthesizing functional dependencies, large interpolants produce large circuit implementations.

In [41], the authors proposed a strategy to mitigate against poor-quality solutions. They did not attempt to reduce the size of interpolants; rather, they suggested repeated trials of Craig Interpolation with different support sets. They suggested iterating over different combinations of dependencies, applying interpolation to each, and picking the one that yields the smallest implementation. After the implementation is chosen, traditional combinational synthesis algorithms are applied to further optimize its structure. Our approach is orthogonal to this one.

In this chapter, we explore methods for modifying resolution proofs in order to obtain more compact interpolants. This, in turn, reduces the amount of logic that is generated for functional dependencies. We use the concept of Minimum Unsatisfiable Cores (MUCs) [42]. An MUC is a minimal set of constraints that need to be present in order to prove that a SAT instance is unsatisfiable. We apply incremental SAT techniques, so our approach is algorithmically efficient.

3.2 Background and Definitions

3.2.1 DPLL Algorithm

Most modern SAT solvers are based on the Davis-Putnam-Logemann-Loveland (DPLL) backtracking-based search algorithm [43]. On a high level, the algorithm works by making a decision about the truth value of an un-assigned variable and then propagating that decision to all literals in the formula corresponding to that variable. If a clause becomes *unit* (containing only one literal) then the value of the corresponding variable is assigned such that the unit clause is satisfied and the assignment is propagated. If at some point a *conflict* occurs (some clause evaluates to false) the search backtracks to the last decision. If during the search all clauses become satisfied, then the current state of the variables is a satisfiable assignment to the formula. Otherwise, if the search backtracks past the first variable decision, the formula is unsatisfiable. The algorithm is described with pseudocode in Figure 3.3.

```
DPLL($\sigma$):
    if every clause in $\sigma$ is true
        return TRUE
    if some clause in $\sigma$ is false
        return FALSE
    for each unit clause $l$
        $\sigma$ = propagate($l$, $\sigma$)
    $l$ = var-decision($\sigma$)
    return DPLL($\sigma \wedge l$) or DPLL($\sigma \wedge \bar{l}$)
```

Figure 3.3: The DPLL algorithm. The “propagate(l, σ)” function assigns “TRUE” to every instance of l in σ. The “var-decision(σ)” function decides a truth value for an unassigned variable and returns the corresponding literal.

Modern SAT solvers use DPLL for the basis for their solving method. Different heuristics are used for making variable decisions and efficient data structures are leveraged to efficiently propagate assignments and discover conflicts.

3.2.2 Clause Learning and Incremental SAT

While the DPLL algorithm described in Figure 3.3 is complete (given an infinite amount of time it will always terminate for any given formula σ) it lacks an important mechanism that gives real power to modern SAT Solvers: Clause Learning. Boolean resolution is an inference rule that states that given two clauses that contain only one literal differing in polarity (negated in one clause but not in the other) a new clause is implied. The variable that differs in polarity is known as the *pivot variable* and the resolved clause is called the *resolvent*. The resolvent contains the union of all the literals in the two clauses except for the pivot variable. The rule is formally stated in Equation 3.1.

$$\frac{(x_1 + \ldots + x_j + \ldots + x_n), (y_1 + \ldots + y_k + \ldots + y_m), (x_j = \bar{y_k})}{(x_1 + \ldots + x_{j-1} + x_{j+1} + \ldots + x_n + y_1 + \ldots + y_{k-1} + y_{k+1} + \cdots + y_m)} \tag{3.1}$$

If a conflict occurs during DPLL search this implies that the conflicting clause, the clause containing all `FALSE` literals, contains only one variable of different polarity than the last clause where propagation occurred. This indicates that a new clause, the resolvent of the conflicting clause and the clause where propagation last occurred, is implied. To illustrate this, consider the SAT instance shown in Figure 3.4.

$$(x + y + z)(x + \bar{w} + z)(\bar{x} + y + z)(\bar{y} + z)(\bar{w} + \bar{z})$$

Figure 3.4: A CNF formula of variables w, x, y, and z.

Assume that the following variable decisions have been made: $y = z = 0$. Under these variable decisions, the first, third, fourth, and fifth clauses become unit clauses. When the value of the first unit clause (x) is propagated, a conflict occurs with the third clause $(\bar{x})$. This indicates that the only variables that differ in polarity between the first and third clauses is the variable x. The resolvent clause $(y + z)$ can then be added to the sat instance. After backtracking and changing the variable decision $z = 1$ a new clause (z) which is the resolvent of $(y + z)$ and $(\bar{y} + z)$ is learned. This new SAT instances with learned clauses is shown in Figure 3.5.

$$(x + y + z)(x + \bar{w} + z)(\bar{x} + y + z)(\bar{y} + z)(\bar{w} + \bar{z})(y + z)(z)$$

Figure 3.5: The CNF formula shown in Figure 3.4 with additional learned clauses $(y+z)$ and (z)

These additional clauses serve as a means to restrict the search space of decision variables. Adding the clause (z) implicitly prevents the search from making any decisions where $z = 0$. Without the use of learned clauses the DPLL algorithm may run into the same conflict where $z = 0$ an exponential number of times.

Learned clauses may also increase the performance of solvers tremendously when instances are solved *incrementally*. In many applications, several *similar* SAT instances need to be solved repeatedly. The formulas may be similar in the sense that they contain many of the same variables and/or clauses. Problems can be cast with a set of *assumption* variables. These are extra variables added to the CNF formula that can implicitly *add or remove* clauses based on the assumed values of the variables. An example of the the CNF formula from Figure 3.4 with added assumption variables is shown in Figure 3.6. The values of variables a_1, a_2, a_3, a_4, and a_5 can be chosen a priori to essentially eliminate certain clauses from the SAT instance. The DPLL procedure can be modified such that the assumption variables are decided during an initialization and if the search ever backtracks past an assumption variable, then the instance is deemed to be "unsatisfiable under the assumptions".

$$(a_1 + x + y + z)(a_2 + x + \bar{w} + z)(a_3 + \bar{x} + y + z)(a_4 + \bar{y} + z)(a_5 + \bar{w} + \bar{z})$$

Figure 3.6: A CNF formula from Figure 3.4 with assumption variables a_1, a_2, a_3, a_4, a_5

If the instance was solved with $a_1 = a_2 = a_3 = a_4 = a_5 = 0$ The additional clauses shown in Figure 3.7 may be learned. Now if the instance is solved again with different assumptions, say $(a_1 = a_3 = a_4 = 0,\ a_2 = a_5 = 1)$, then the solver will not explore the space of assignments restricted by the learned clauses generated from the previously solved instance.

$$\begin{aligned}(a_1 + x + y + z)(a_2 + x + \bar{w} + z)(a_3 + \bar{x} + y + z)(a_4 + \bar{y} + z)(a_5 + \bar{w} + \bar{z})\\ (a_1 + a_3 + y + z)(a_1 + a_3 + a_4 + z)\end{aligned}$$

Figure 3.7: A CNF formula from Figure 3.6 with two additional learned clauses

Learned clauses can dramatically improve solver performance, but as more and more are generated there can be diminishing returns. Modern solvers leverage heuristics for dropping learned clauses if they are deemed to be not "useful" [44].

3.2.3 Resolution Proofs and Craig Interpolation

Learning not only provides serious performance improvements on top of DPLL, but it also provides a mechanism for verifying the correctness of SAT solvers. With some minor book-keeping during the DPLL search the learned clauses can be used to produce a *resolution proof* of unsatisfiability [45]. A set of clauses can be proved to be unsatisfiable through a series of resolutions that lead to an empty clause. This results in a directed acyclic graph (DAG): the *roots* are the original clauses, the *intermediate* nodes are clauses proved by resolution, and the single *leaf* is the empty clause. We will sometimes use the words "node" and "clause" interchangeably when we are discussing resolution proofs.

When two clauses c_1 and c_2 resolve a clause c_3, c_1 and c_2 are said to be the *parents* of c_3; c_3 is said to be a *child* of c_1 and c_2. Clauses that are used to resolve c_1 or c_2 are said to be *ancestors* of c_3. When we say that a node is towards the *beginning* of a proof, we are declaring that there are few resolution steps taken from the leaves of the proof to reach this node. When we say that a node is towards the *end* of a proof, we are declaring that there are few resolution steps that need to be taken to reach the empty clause from this node. An example of resolution proof for the unsatisfiability of a SAT instance similar to the formula in Figure 3.4 is shown in Figure 3.8.

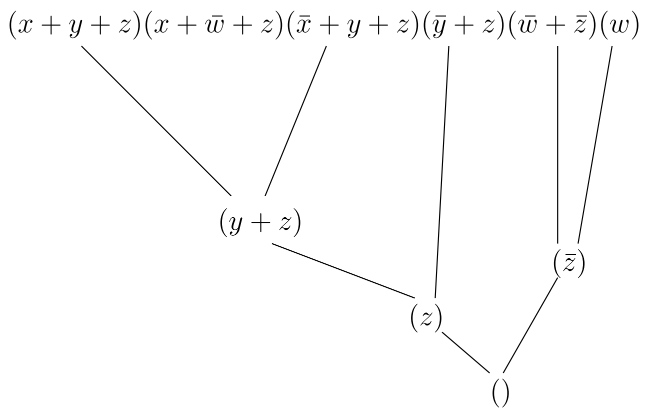

$(x + y + z)(x + \bar{w} + z)(\bar{x} + y + z)(\bar{y} + z)(\bar{w} + \bar{z})(w)$
$(y + z)$
$(\bar{z})$
(z)
$()$

Figure 3.8: A resolution proof of unsatisfiability

Resolution proofs are not only useful for verifying the results of a SAT solver, but also for examining certain properties of the underlying formula. Given an unsatisfiable instance of SAT and a bi-partition of its clauses, set $\boldsymbol{A}$ and set $\boldsymbol{B}$, Craig's Interpolation theorem states that there exists an intermediate formula $\boldsymbol{I}$, called an *interpolant*, such that $\boldsymbol{A} \rightarrow \boldsymbol{I}$ and $\boldsymbol{I} \rightarrow \overline{\boldsymbol{B}}$. A variable in the SAT instance is said to be a *global variable* if it is present in both clause sets $\boldsymbol{A}$ and $\boldsymbol{B}$. Likewise, a variable is said to be *local* to a clause partition if it is only present in that clause partition. An interpolant only contains variables that are global to $\boldsymbol{A}$ and $\boldsymbol{B}$. We say that a set of clauses is *satisfied* for some assignment of the set's variables if every clause in the set evaluates to true.

The algorithm in Figure 3.9, presented in [1], is a procedure for generating a circuit that implements an interpolant from a resolution proof and a clause partition. It was adapted from a procedure presented in [40] to find the Boolean value for an interpolant given a variable assignment. A proof of correctness for the procedure in Figure 3.9 is provided in [4].

```
p($c$):
    if $c$ is a leaf clause
        if $c$ is in $A$
            return g($c$)
        else
            return 1
    else let v be the pivot variable
        if v is local to $A$
            return p($c_1$) $\vee$ p($c_2$)
        else
            return p($c_1$) $\wedge$ p($c_2$)
```

Figure 3.9: The algorithm proposed in [1] to produce a circuit that implements an interpolant of a given clause partition, via a proof of unsatisfiability.

The procedure g(c) has a single clause c as its argument. The procedure returns clause c with only its global literals present. Let c_1 and c_2 be c's parent clauses. Procedure p(c) is defined in Figure 3.9. Calling p(c) on the empty clause of a resolution proof will return a DAG whose nodes represent Boolean functions. In this DAG, the node with no fanout, corresponding to the empty clause in the resolution proof, computes a Boolean function in terms of the global variables of $\boldsymbol{A}$ and $\boldsymbol{B}$. This Boolean function is an interpolant of the given clause partition. When we refer to the *size* of an interpolant, we mean the number of gates that are needed to represent it. It should be clear that the size of the interpolant is bounded by the number of nodes in the resolution proof. Figure 3.10 shows the results of applying this procedure on the resolution proofs in Figures 3.1 and 3.2.

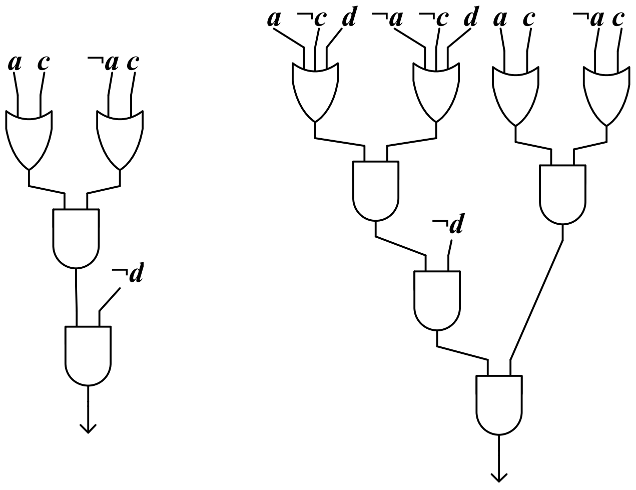


Figure 3.10: Two interpolants produced by calling p(c) on the empty clause of a resolution proof. The circuits on the left and right are generated from the proofs in Figures 3.1 and 3.2, respectively

3.3 Proposed Methodology

Whether or not a function is a valid interpolant for a clause partition depends on the space of Boolean assignments that each clause partition covers. A CNF formula is a non-canonical representation for a Boolean function so there are many different valid sets of clauses that cover the same space of Boolean assignments.

Proposition 9 *Given a resolution proof of unsatisfiability of clause set* ***A*** *and* ***B****, we can move all nodes that were resolved from only nodes of* ***A*** *and all nodes that were resolved from only nodes of* ***B*** *into the set of leaves of* ***A*** *and* ***B****, respectively. This action will still preserve the space of Boolean assignments covered by* ***A*** *and* ***B****. The* *interpolant generated from this new resolution proof will be a valid interpolant of the original clause sets* $\boldsymbol{A}$ *and* $\boldsymbol{B}$*.*

Proof 9 *Consider two leaves* n_1 *and* n_2 *in* $\boldsymbol{A}$ *(*$\boldsymbol{B}$*) and a new node* n_3 *that is the resolvent of* n_1 *and* n_2*. Since* n_3 *is never false for a variable assignment that causes* n_1 *and* n_2 *to be true,* n_3 *can be added to the set of leaves of* $\boldsymbol{A}$ *(*$\boldsymbol{B}$*) without reducing or expanding the space of Boolean assignments that satisfy* $\boldsymbol{A}$ *(*$\boldsymbol{B}$*). Since the space of Boolean assignments representing* $\boldsymbol{A}$ *and* $\boldsymbol{B}$ *does not change – only the clause representation changes – an interpolant generated by the procedure in Figure 3.9 while considering* n_3 *to be a leaf clause will still be a valid interpolant of the original clause partition.*

$\square$

We can mark nodes resolved from only nodes of $\boldsymbol{A}$ or $\boldsymbol{B}$ as leaves of $\boldsymbol{A}$ or $\boldsymbol{B}$, respectively. In theory this optimization should yield a smaller interpolant for a proof. The intuition behind this is that we are essentially generating an interpolant on a proof that has fewer resolutions (since many of the internal nodes can considered as leaves). However, for most applications this optimization by itself doesn't yield a significant improvement in interpolant size.[1]

Example 7 *Let us see how this optimization affects the interpolants of the proofs in Figures 3.1 and 3.2. In Figure 3.1, we notice that resolvent clause* (c) *was resolved from two leaf clauses of* $\boldsymbol{A}$ *(clauses* $(a + c)$ *and* $(\bar{a} + c)$*). This means that we can consider clause* (c) *to be a leaf of* $\boldsymbol{A}$*. Calling p(c) on this proof with clause (c) marked as a leaf node will yield the interpolant* $(c)(\bar{d})$*.*

[1] Often, CNF formulas are generated from a Tstein Decomposition of a logic circuit [3]. During the course of generating this representation, many variables are created. When a clause partition is made, there are usually very few variables that are common to both partitions (indeed this is the case for the applications in [23] and [1]). When a proof of unsatisfiability is generated, many of the resolutions involve variables that are only present in one partition. Performing an optimization that considers internal nodes as leaf clauses will do very little to improve the overall size of the interpolant with this kind of proof. This is because p(c), for the most part, will be creating redundant gates. If we are generating an interpolant using this method on a data structure that doesn't allow the creation of redundant logic – such as a structurally hashed AIG [46] – this optimization will likely yield little benefit to the interpolant size.

In the proof shown in Figure 3.2, we can see that the empty clause is derived from implications that only occur from partition $\boldsymbol{A}$. This means that the empty clause can be considered as a leaf of $\boldsymbol{A}$. Calling p(c) on this proof with the empty clause marked as a leaf of $\boldsymbol{A}$ will yield an interpolant of constant 0 *(since the OR of the global literals of an empty clause is* 0*).*

This optimization allows us to reduce the number of gates needed to implement the interpolant in Figure 3.1 to 1 gate and the number of gates to implement the proof in Figure 3.2 to 0 gates.

What may become clear from this observation is that proofs that tend to have few resolutions between a clause resolved from $\boldsymbol{A}$ and a clause resolved from $\boldsymbol{B}$ will tend to have smaller interpolants. This is because more of the internal nodes can be considered as leaves and therefore fewer gates will be created by the p(c) procedure. Abusing English a little, we will refer to a proof of this kind of structure as being more *disjoint* than a proof that has more resolutions that occur between a clause of $\boldsymbol{A}$ and a clause of $\boldsymbol{B}$ (e.g., the proof in Figure 3.2 is more disjoint then the proof in Figure 3.1).

In [42], a SAT-Based methodology is proposed that attempts to change the order of resolutions in a proof of unsatisfiability in order to yield a smaller set of leaves that are involved in the proof. In this work, we attempt to use the same type of methodology in order to yield a proof that will generate a smaller interpolant using the algorithm in Figure 3.9.

Consider some node c in a resolution proof of unsatisfiability for a SAT instance. If we follow the series of resolutions that took place in order to resolve c backwards (towards the leaves), we eventually arrive at a set of leaves that were used to derive c. Let $\boldsymbol{R}$ be this set of leaves.

Proposition 10 *For all variable assignments that satisfy the set of clauses $\boldsymbol{R}$, c must also be satisfied.*

Proof 10 *let c_1 and c_2 be the clauses that resolved c (c's parent nodes). Every variable assignment that satisfies both c_1 and c_2 must also satisfy c (because $(c_1)(c_2) \rightarrow c$). Likewise, every variable assignment that satisfies both of c_1's parents also satisfies c_1 and every variable assignment that satisfies both of c_2's parents also satisfies c_2 (and so on with c_1 and c_2's parents). Therefore every variable assignment that satisfies the ancestor nodes of c must also satisfy c.*

□

Since $\boldsymbol{R} \rightarrow c$, we know that the SAT instance $(\boldsymbol{R})(\bar{c})$ must be unsatisfiable. In [42], the authors exploit this fact to determine whether or not a clause c can be derived through resolution from a set of clauses $\boldsymbol{R}$. They propose a SAT-Based algorithm that iteratively checks intermediate nodes to see if they can be implied by a smaller set of leaves. The goal of the algorithm is to find a smaller set of leaf clauses that are needed to prove that the CNF formula is unsatisfiable.

We propose using this type of SAT instance to check to see whether or not a clause can be resolved by only leaves of set $\boldsymbol{A}$ or $\boldsymbol{B}$. We can verify if clause c can be implied from only clauses of $\boldsymbol{A}$ by checking the satisfiability of $(\boldsymbol{A})(\bar{c})$. Likewise we can check to see if c can be resolved from only clauses of $\boldsymbol{B}$ by checking the satisfiability of $(\boldsymbol{B})(\bar{c})$. If both of these SAT instances are satisfiable, then we know that clauses of both $\boldsymbol{A}$ and $\boldsymbol{B}$ are required to resolve clause c. If $(\boldsymbol{A})(\bar{c})$ is unsatisfiable then we know that c can be considered to be a leaf of $\boldsymbol{A}$. If $(\boldsymbol{B})(\bar{c})$ is unsatisfiable then we know that c can be considered to be a leaf of $\boldsymbol{B}$.

We propose using this observation to prove that some nodes in a resolution proof can be implied by only nodes of $\boldsymbol{A}$ or only nodes of $\boldsymbol{B}$ and therefore can be considered as leaves of $\boldsymbol{A}$ or $\boldsymbol{B}$.

Example 8 *Consider the proofs in Figures 3.1 and 3.2 again. The proof in Figure 3.1 involves only one clause of $\boldsymbol{B}$: $(d + \bar{c})$. Here we see that clause $(\bar{c})$ is resolved from a clause in $\boldsymbol{A}$ and a clause in $\boldsymbol{B}$. In the other proof we can see that the clause $(\bar{c})$ can be* *derived from the resolution of clauses* $(a+\bar{c}+d)$,$(\bar{a}+\bar{c}+d)$*, and* $(\bar{d})$*. We can create a SAT instance to check whether or not* $(\bar{c})$ *can be derived from clauses of* $\boldsymbol{A}$*. This SAT instance would be:* $(a+\bar{c}+d)(\bar{a}+\bar{c}+d)(a+c)(\bar{a}+c)(\bar{d})(c)$*. We know this instance will be unsatisfiable based on the resolution shown in Figure 3.2. This tells us that we can consider clause* $(\bar{c})$ *to be a leaf of partition* $\boldsymbol{A}$*. As shown earlier, marking* $(\bar{c})$ *as a leaf of* $\boldsymbol{A}$ *allows us to generate an interpolant of constant* 0 *for this proof.*

In what follows, we propose a methodology using these observations to modify a resolution proof in order to yield a smaller interpolant.

1. Mark every node that was resolved from only nodes of $\boldsymbol{A}$ or $\boldsymbol{B}$ as leaves of $\boldsymbol{A}$ or $\boldsymbol{B}$ respectively. Mark every other node as unvisited.
2. Select a clause c that is not a leaf of $\boldsymbol{A}$ or $\boldsymbol{B}$ and is not marked as visited. Check the satisfiability of $(\boldsymbol{A})(\bar{c})$ and $(\boldsymbol{B})(\bar{c})$.
3. Solve the SAT instances created in the previous step. If $(\boldsymbol{A})(\bar{c})$ is unsatisfiable, mark c as a leaf of $\boldsymbol{A}$. Otherwise, If $(\boldsymbol{B})(\bar{c})$ is unsatisfiable, mark c as a leaf of $\boldsymbol{B}$. Mark c as visited.
4. If c is now marked as a leaf of either $\boldsymbol{A}$ or $\boldsymbol{B}$, check to see if its children can trivially be marked as leaves of $\boldsymbol{A}$ or $\boldsymbol{B}$ and check to see if some of c's ancestors can be marked as visited.
5. Repeat steps 2-4 until all nodes are marked as either visited or as leaves leaves of $\boldsymbol{A}$ or $\boldsymbol{B}$ or until a threshold number of calls to the SAT solver is reached.

The details of steps 2 and 4 are explained in the next section.

3.3.1 Optimizations

The goal of our method is to determine if we can find a more disjoint proof to generate the interpolant. We can create SAT instances that check to see if nodes that have ancestor nodes of both $\boldsymbol{A}$ and $\boldsymbol{B}$ can be labeled as leaves of either $\boldsymbol{A}$ or $\boldsymbol{B}$. We will refer to such nodes as *mixed nodes.* Since the complexity of the method is dominated by calls to the SAT solver, we aim to reduce the number of SAT instances that need to be solved.

Rather than checking every mixed node to see if it can be considered as a leaf, we can limit ourselves to a fixed or variable number of nodes that we consider based on the overall size of the proof. For large proofs generated from large CNF formulas, many nodes may be mixed. At the same time, the size of the SAT instance that needs to be solved in order to determine if a node can be marked as a leaf increases (because there will be many clauses in $\boldsymbol{A}$ and $\boldsymbol{B}$). This can lead to a very long runtime for large CNF formulas. However, we can halt at anytime and still reduce the size of the interpolant.

In most cases, the nodes toward the bottom of the resolution proof (close to the empty clause) will tend to be mixed nodes. If we check these nodes first and verify that a node towards the end of the proof can be considered as a leaf node, then we might not need to check some of the node's ancestors.

Proposition 11 *Consider some mixed node n which we prove to be a leaf of either $\boldsymbol{A}$ or $\boldsymbol{B}$ by solving an instance of satisfiability. If an ancestor of n is only involved in resolutions that lead to n, then we do not need to check whether this node is a leaf node.*

Proof 11 *The procedure in Figure 3.9 terminates on leaves. If n is marked as a leaf node then the procedure will not be called on its parents and therefore will never be called on any of n's ancestors who are only involved in resolutions leading to n.*

$\square$

Example 9 *To better illustrate this point, consider the resolution proof shown in Figure 3.11. let us say that nodes 1–5 are mixed nodes in this proof. If we prove that node 1 can be considered to be a leaf of $\boldsymbol{A}$ then we will not have to check node 3 (because node 3's only resolvent is node 1). However, node 4 may still need to be checked because* *it is involved in the resolution of node 2. If we prove that both nodes 1 and 2 can be considered leaves of **A** (or **B**) first, then we do not need to check node 4 (since node 4 will not be reached by p(c) when p(c) is called on the empty clause).*

This is the condition that we are considering in Step 4 of our methodology when we say that we should check to see if ancestor nodes can be marked as visited. In this example, we would mark node 3 as visited, and we would not solve a SAT instance to see if it can be a leaf node. Showing that a mixed node can be considered leaf may allow us to check fewer of the node's ancestors, but it does not imply that its ancestors can be considered leaf.[2]

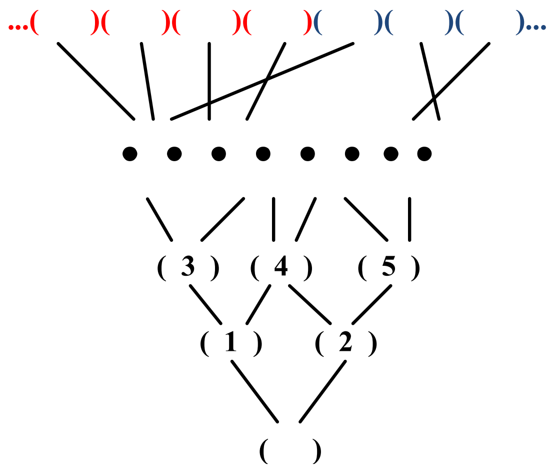


Figure 3.11: A resolution proof from an unsatisfiable CNF formula. Clauses of ***A*** are shown in red while clauses of ***B*** are shown in blue. Nodes 1, 2, 3, 4, and 5 are nodes somewhere in the proof

Proposition 12 *If we prove that two parents of a mixed node can be considered leaves of the same clause partition, then this implies their child clause can be considered as a leaf of this partition.*

[2] Clearly the series of implications that showed the node to be mixed in the original resolution proof used resolutions that occurred from nodes from both ***A*** and ***B***. Unless the leaves involved in the proof from one partition can be implied from the other, at least one of the ancestor nodes in the proof must remain mixed.

Proof 12 *This is basically the same condition as Proposition 1. If nodes n_1 and n_2 are marked as leaves of* $\boldsymbol{A}$ *(*$\boldsymbol{B}$*), then their resolvent node n_3 can be added to the same set of clauses as n_1 and n_2 without reducing or expanding the set of variable assignments that satisfy* $\boldsymbol{A}$ *(*$\boldsymbol{B}$*).*

$\square$

Example 10 *Once again consider the proof shown in Figure 3.11. If we show that nodes 3 and 4 can be considered to be leaves of* $\boldsymbol{A}$ *(or* $\boldsymbol{B}$*), then we do not need to create a SAT instance to see if node 1 can be marked as a leaf of* $\boldsymbol{A}$ *(*$\boldsymbol{B}$*) because it can trivially be considered a leaf node of one partition since both of its parents are leaves of the same partition.*

This is the condition that we are considering in Step 4 of our methodology when check to see if a node's children can be trivially marked as leaves.

By initially checking nodes that are close to the leaves of the proof, we can avoid unnecessary calls to the SAT solver because we may be able to mark many children of these nodes as leaves. Also, these nodes are more likely to be converted to leaves because they are in a sense "closer" to the set of original leaves.

However, if we initially check nodes that are close to the end of the proof (near the empty clause), we can avoid unnecessary calls to the SAT solver by marking many ancestor nodes as visited. These nodes are less likely to be proven to be leaves because they are in a sense "further" from the set of original leaves.

We will refer to the method of checking nodes towards the end of the proof first as a *backward search*, and we will refer to the method of checking nodes towards the beginning of the proof first as a *forward search*. A forward versus a backward search changes the order in which we consider nodes in Step 2 of our methodology. Using both methods may allow us to modify a proof while solving fewer SAT instances. However, if we check every node (regardless of the order) both methods will yield the same modified resolution proof. We determine the ordering of nodes in a resolution proof by the order in which the clauses were resolved by the SAT solver that produced the resolution proof.

When a SAT solver produces a resolution proof from an unsatisfiable CNF formula, it also provides an ordering of how each clause is implied [45]. The backward search checks clauses that were resolved at the *end* of the SAT solver's trace first, while the forward search checks clauses that were resolved at the *beginning* of the SAT solver's trace first.

3.3.2 Incremental Techniques

Since the SAT instances we are solving are all similar, we can implement the SAT solving portion of Step 2 of our methodology using incremental SAT techniques [38]. To implement these techniques we simply add two new variables into the SAT instance. For every clause in $\boldsymbol{A}$ we will add literal a_{off} and for every clause in $\boldsymbol{B}$ we add b_{off}. When we want to determine if a node can be considered a leaf of $\boldsymbol{A}$, we set the variable assumptions to be $a_{\text{off}} = 0$ and $b_{\text{off}} = 1$. When we want to determine if a node can be considered a leaf of $\boldsymbol{B}$ we set the variable assumptions to be $a_{\text{off}} = 1$ and $b_{\text{off}} = 0$. Setting the assumptions in this way essentially tells the SAT solver to ignore the clauses of set $\boldsymbol{A}$ or $\boldsymbol{B}$. After setting a_{off} and b_{off}, we assume all the literals of the node under inspection to be zero.

Example 11 *Consider again the proof shown in Figure 3.1. Let us say we want to use incremental techniques to see if clause $(d + \bar{c})$ could be considered to be a clause of $\boldsymbol{A}$. To solve this we check the satisfiability of the following CNF formula:*

$$(a+\bar{c}+d+a_{\mathit{off}})(\bar{a}+\bar{c}+d+a_{\mathit{off}})(a+c+a_{\mathit{off}})(\bar{a}+c+a_{\mathit{off}})(\bar{d}+a_{\mathit{off}})(d+\bar{c}+b_{\mathit{off}})(a+b+b_{\mathit{off}})$$

When we solve this SAT instance we assume $a_{\mathit{off}} = 0$ and $b_{\mathit{off}} = 1$. We also assume $d = 0$ and $\bar{c} = 0$ (this is the same as assuming that clause $(d + \bar{c})$ is 0). Notice if we want to check any other mixed node in the resolution proof we can use the same SAT instance but just change the set of variable assumptions. Since the SAT instance is the *same for each call to the SAT solver, it can remember information about the state of previous instances and use this information to make later instances easier to solve [38].*

3.4 Results

To test our algorithm, we created different SAT instances that checked for valid functional dependencies in the benchmarks listed in Tables I through IV. The SAT instances were generated using the method described in [23]. The support sets that we considered were for the benchmark's primary outputs expressed in terms of other primary outputs and primary inputs. We iterated over many possible support sets searching for valid sets of a minimal size. Once we verified that certain support sets could be used to implement primary outputs, we created a resolution proof from the corresponding CNF formula. We then compared the forward and backward traversals of the resolution graph checking to see if mixed nodes could be considered to be leaves. We generated the interpolants from the resolution proofs using the algorithm in Figure 3.9. Here we compare the sizes of the interpolants generated form the algorithm in Figure 3.9 on modified and unmodified resolution proofs. We chose benchmarks from the LGSynth05 [31] benchmark suite that had many possible valid target functions. Tables I and II provide detailed results for a particular benchmark, `table3`. Tables III and IV summarized the results for other benchmarks. In Tables III and IV, the numbers in every column are the average value of the field among all the functional dependencies that were generated.

The experiments were run on an AMD Athlon 64 X2 6000+ CPU with 3 GB of RAM. Only one core was utilized by the algorithm. Our code was implemented in Berkeley ABC [32] using MiniSAT for SAT solving [33].

We limited the number of mixed nodes that we checked in each resolution proof to 2500. (This limit was reached for the larger resolution proofs.) In each check, we solve two SAT problems (to see if the node can be considered a leaf of $\boldsymbol{A}$ or $\boldsymbol{B}$). The number of mixed nodes that were checked for each resolution proof is indicated in the "# Nodes Checked" column. The "# Res Nodes" column indicates the number of nodes formed by resolution in the original proof. The "# Found" column is the number of mixed nodes that we found to be leaves by proving a SAT instance to be unsatisfiable. The "Orig. Size" column lists the number of AIG nodes in the interpolant before optimizing the resolution proof. The "New Size" column lists the number of AIG nodes in the interpolant after the resolution proof was modified. The "Time" column indicates the time that it took to search through the nodes of the resolution proof and to check the SAT instances for the mixed nodes.

After the interpolants were simplified, we ran the `compress2` script in ABC on the original interpolants and the interpolants generated after the forward and backward searches. The idea is that our method could be used initially to make vast changes to the overall structure of the interpolant, and then other logic minimization techniques could be applied to the resulting structure. To show that our algorithm achieves minimization beyond traditional synthesis techniques, we ran `compress2` on interpolants generated from modified resolution proofs and non-modified proofs and then compared their sizes. The percent reduction in size is listed in the "% Change Compress2" column in Tables III and IV. "% Change" was calculated by: (New Size - Old Size) / Old Size. The size of the original and new interpolants after running `compress2` is shown in Tables I and II under the "Orig. Comp2" and "New Comp2" columns.

We see that modifying the resolution proof often results in substantial improvements in the interpolant size. After the `compress2` script is run, the % change in size between interpolants is less significant, but on average is still better than running `compress2` without modifying the proof. Tables I and II show the results of 10 iterations of `compress2`. We have noticed that running multiple iterations doesn't yield significant differences in terms of % change in size between interpolants generated from modified and non modified resolution proofs. Accordingly, Tables III and IV show the results from just one iteration of `compress2`.

The time for constructing an interpolant on the modified resolution proof and running `compress2` was negligible compared to the time it takes to simplify the resolution proof. In general, the backward search method makes more substantial reductions in size with a smaller number of calls to the SAT solver compared to the forward search method. Increasing the maximum number of node checks would likely yield better results at the expense of longer runtimes – particularly for some of the larger benchmarks where the maximum number of node checks was frequently reached.

For a couple of functions presented in Table I the original and new interpolant sizes were the same, yet the sizes after running `compress2` were different (see functions 1 and 5). This is due to the fact that our implementation gave the AIG nodes different orderings between the original and new AIGs. This can sometimes change the results of running `compress2`.

In many cases, the primary outputs of benchmarks have very small support sets. For the benchmarks listed in Tables III and IV, we did not report the savings for primary output functions that contained less that 50 AIG nodes. Also, our techniques performed much better on dependency functions where the support set contained many primary output functions and few primary input functions. This is likely due to don't care conditions that exist in the circuit that our method implicitly takes advantage of. We will discuss this in Section 5.5.

`table3` Benchmark: Forward Search								
Function #	# Res Nodes	Orig Size	New Size	# Nodes Checked	# Found	Time (s)	Orig Comp2	New Comp2
0	32262	277	267	2500	61	80.85	105	93
1	128654	1254	1254	2500	0	281.31	328	329
2	95042	638	630	2500	283	218.25	248	226
3	71647	682	648	2500	423	157.66	273	215
4	57015	776	743	2500	432	126.26	380	364
5	47285	657	657	2500	0	106.23	251	233
6	43884	268	245	2500	578	94.67	91	104
7	26714	287	271	2500	335	64.37	144	126
8	31715	116	90	2500	48	79.40	55	34
9	13182	43	36	1090	65	17.25	22	18
10	70964	867	850	2500	576	146.85	413	397
11	31772	253	229	2500	67	80.12	86	107
12	45784	376	360	2500	404	98.61	172	184
13	29078	408	373	2500	757	64.73	130	55

Table 3.1: Results of the forward-search method on the `table3` benchmark. Each function is a PO expressed in terms of the PIs and other POs.

`table3` Benchmark: Backward Search								
Function #	# Res Nodes	Orig Size	New Size	# Nodes Checked	# Found	Time (s)	Orig Comp2	New Comp2
0	32262	277	129	2500	20	85.88	105	58
1	128654	1254	1238	2500	5	287.62	328	346
2	95042	638	574	2500	8	225.37	248	217
3	71647	682	469	2500	45	179.96	273	177
4	57015	776	490	2500	26	144.83	380	193
5	47285	657	611	2500	8	114.33	251	242
6	43884	268	224	2500	8	107.96	91	106
7	26714	287	87	2500	27	76.61	144	51
8	31715	116	76	2500	15	85.23	55	34
9	13182	43	36	1017	3	16.55	22	18
10	70964	867	349	2500	41	179.22	413	192
11	31772	253	191	2500	8	82.38	86	50
12	45784	376	203	2500	34	120.00	172	117
13	29078	408	112	2500	32	84.29	130	37

Table 3.2: Results of the backward-search method on the `table3` benchmark. Each function is a PO expressed in terms of the PIs and other POs.

Forward Search						
Benchmark	# Res Nodes	# Nodes Checked	# Found	% Change	% Change Compress2	Time (s)
`apex1`	28279	2413	30	-4.89%	-2.73%	69.48
`apex3`	68585	1494	21	-2.12%	-1.47%	140.99
`styr`	9373	2143	88	-8.71%	-5.71%	18.3
`s1488`	5748	824	29	-9.24%	-8.41%	7.62
`s1494`	10488	1266	21	-6.69%	-4.43%	15.51
`s641`	46416	1886	39	-26.67%	-2.33%	97.45
`s713`	42412	1910	89	-36.00%	-3.70%	89.16
`table5`	35373	2500	252	-13.83%	-4.08%	48.05
`vda`	12951	2011	120	-18.78%	-17.33%	27.34
`sbc`	13951	1094	8	-1.46%	-1.08%	19.09

Table 3.3: A table of the averaged results using the forward-search method among different benchmarks.

Backward Search						
Benchmark	# Res Nodes	# Nodes Checked	# Found	% Change	% Change Compress2	Time (s)
`apex1`	28279	2384	6	-8.95%	-5.84%	72.03
`apex3`	68585	1485	5	-8.41%	-5.24%	145.63
`styr`	9373	2124	10	-11.57%	-10.14%	19.36
`s1488`	5748	797	5	-9.92%	-9.59%	7.98
`s1494`	10488	1241	7	-6.93%	-5.19%	15.83
`s641`	46416	1820	14	-42.22%	-2.78%	95.37
`s713`	42412	1724	17	-43.90%	-6.20%	82.86
`table5`	35373	2358	7	-26.67%	-15.83%	81.16
`vda`	12951	1850	7	-21.72%	-19.72%	27.07
`sbc`	13951	1087	1	-1.46%	-0.92%	19.09

Table 3.4: A table of the averaged results using the backward-search method among different benchmarks.

3.5 Discussion

There is still some variability in this method that can be explored. When checking mixed nodes for eligibility as root nodes, some nodes were shown to be considered as a clause of either $\boldsymbol{A}$ or $\boldsymbol{B}$. Depending on which set the node belongs to, more or fewer nodes in the resolution graph may have to be checked to see if they can be considered a root node. This idea is illustrated in the following example.

Example 12 *Consider the resolution proof in Figure 3.11. Let us say that we prove that node 4 can be considered as either a root of* $\boldsymbol{A}$ *or* $\boldsymbol{B}$*. At the same time, let us say we have proved that nodes 3 and 5 are roots of* $\boldsymbol{B}$*. We may be better off choosing to label node 4 as a root of* $\boldsymbol{B}$ *because then we avoid having to check whether or not nodes 1 and 2 are root nodes (because their parents are roots of* $\boldsymbol{B}$ *so they are also roots of* $\boldsymbol{B}$*).*

Furthermore, rather than considering all the nodes of sets $\boldsymbol{A}$ or $\boldsymbol{B}$ we could heuristically restrict ourselves to nodes that are *near* the mixed node that we are considering (near in a topological sense). This will reduce the size of the SAT instances that are iteratively solved and this will likely reduce the overall runtime.

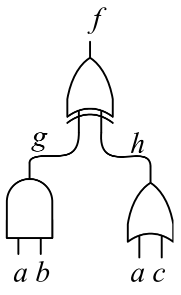


Figure 3.12: A circuit with an observability don't care of $g = 1$, $h = 0$

If interpolants are used as a starting point to generate a structure to perform traditional synthesis, these traditional techniques will perform better. When computing interpolants in the context of synthesizing functional dependencies, the function that is generated implicitly takes advantage of don't care conditions that exist in the circuit.

Consider the circuit shown in Figure 3.12. We see that function f can be expressed in terms of functions g and h. If we create a SAT instance to check whether or not f can be expressed in terms of variables g and h, the instance will be unsatisfiable. We can then use the resolution proof from this unsatisfiable instance to generate an interpolant whose function is the implementation of f in terms of g and h. However, we can see from Figure 3.12 that when g is logic 1, h must also be logic 1 (because of input a). After the interpolant that implements f is generated, f may be either logic 1 or logic 0 for the assignment $g = 1$ $h = 0$ (because the interpolant is an over-approximation of the on-set of f). If we pass the interpolant to traditional synthesis algorithms to optimize it, the algorithms will not know about the don't care condition on f and g.[3] However, if we optimize the resolution proof before generating the interpolant, then we can force the assignment of $g = 1$ $h = 0$ to cause f to evaluate to logic 0. Using this implementation of f as a starting point for traditional multilevel synthesis algorithms will yield better results.

We discussed the use of incremental SAT-based techniques to modify a resolution proof to yield a smaller interpolant. We positioned this method as a starting point for traditional synthesis algorithms. Perhaps this approach is more broadly applicable. In [23] it was shown that Craig Interpolation can be used to generate implementations for functions with a given support set. The choice of support set directly effects the clause partition in the SAT instance. If a larger support set is chosen, then a more constrained CNF formula is constructed. Using our approach, perhaps we could create a resolution proof from an unsatisfiable SAT instance (with a large support set) and perform optimizations on this proof to improve the *entire circuit.* Unlike modern synthesis algorithms that perform incremental operations on small portions of a network, working with a resolution proof might allows us to make incremental SAT calls that can

[3] Here we are assuming that the function $f(g, h)$ is passed to a synthesis algorithm in isolation from the rest of the circuit. If the interpolant is not considered in isolation, then traditional synthesis algorithms may take advantage of the don't care condition on f. The idea we are trying to present is that if g and h are many levels away from the primary inputs of the circuit, then local logic optimizations may not be able to detect this don't care condition.

make vast changes to a network's structure. In the next chapter we discuss thesse ideas in more detail.

Chapter 4

Resolution Proofs as a Data Structure For Logic Synthesis

4.1 Introduction

The previous chapter discussed methods for modifying resolution proofs with the aim of reducing the size of an interpolant generated from the proof. When interpolation is used to generate functional dependencies, often the goal is to generate dependencies for multiple target functions. In this case, target functions that are able to share gates in their transitive fanin are ideal. However, large differences in the resolution proofs can lead to little logic sharing between interpolants. Consider the example illustrated in Figures 4.1 and 4.2. Both resolution proofs have many of their root clauses in common. However, the resolution proofs in Figure 4.1 have few of their intermediate clauses in common. As a result, after the interpolants are generated for each proof, none of the gates compute the same Boolean function. In contrast, the resolution proofs in Figure 4.2 share many of the same intermediate clauses. The interpolants generated for these proofs share more logic compared to those in Figure 4.1.

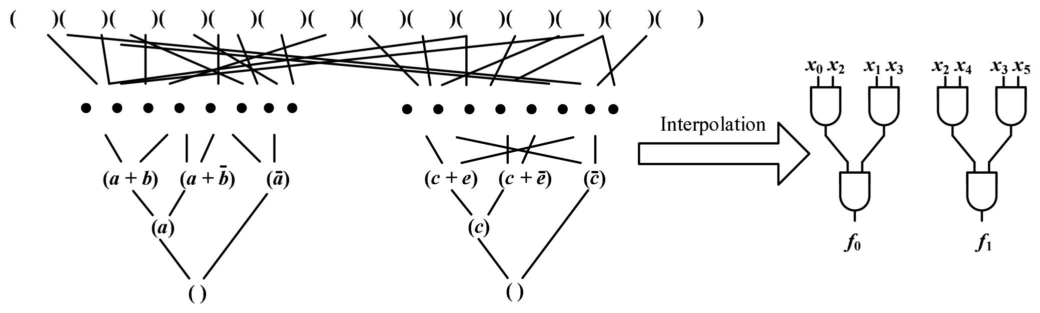


Figure 4.1: A conceptual example of two resolution proofs with very few shared clauses. The resulting interpolants do not contain any shared logic.

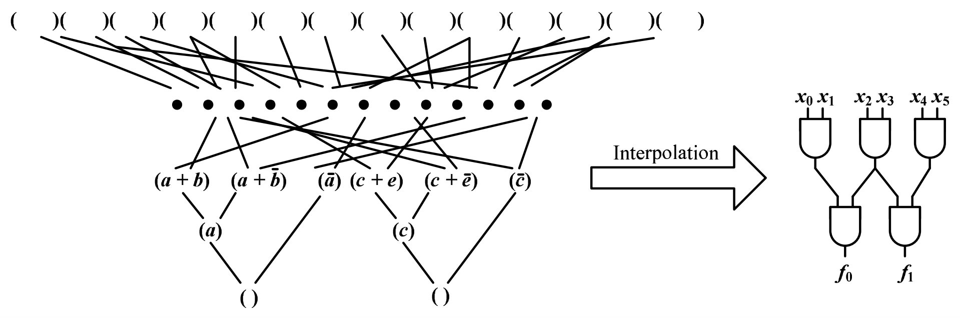


Figure 4.2: An conceptual example of two resolution proofs that share many of the same clauses. The resulting interpolants share some of same logic.

Many modern synthesis tools use And-Inverter Graphs (AIGs) as their underlying data structure. With AIGs, equivalence between nodes can be asserted with SAT-based algorithms, combined with structural hashing [47]. When two nodes are proved to be equivalent, one node can be substituted in place of the other, and any dangling logic can be removed from the netlist.

A related but more difficult problem is determining whether or not certain nodes can be expressed as a function of other nodes (sometimes called *substitution* or *resubstitution* [28]). For this task, the application of SAT-based methods is not so straightforward. In contrast, it can be easily determined if nodes in a resolution proof can be resolved from a subset of other nodes. Accordingly, Craig Interpolation provides a new approach to performing substitution in logical synthesis.

In this chapter, we propose an algorithm for merging resolution proofs and then restructuring them. The structure in the resulting interpolant is then less sparse. This interpolant can be further minimized with traditional minimization algorithms. Trials on benchmarks suggest that there is significant potential for restructuring in the proofs of target function sets encountered in practice.

4.2 Related Work and Context

In [23], a method for generating functional dependencies based on Craig Interpolation was proposed. This method was shown to scale much better with circuit size than previous methods based on binary decision diagrams (BDDs) [48]. While the process of finding and generating the dependencies with this method is efficient, in many cases, the resulting logic is poor. This is because the interpolant that implements the dependency function is often large and redundant. The structure of the interpolant is heavily dependent on the structure of the proof of unsatisfiability generated by the SAT solver. Generally, solvers strive for speed without regard to the proof structure. Some methods for reducing the size of interpolants in specific application domains have been proposed [41]. Methods for adjusting and reducing the size of resolution proofs have been discussed [42, 49]. Unfortunately, it is difficult to predict or measure how these algorithms affect the structure of the interpolants that are generated from the proofs.

In the previous chapter, an algorithm for augmenting a resolution proof with the goal of reducing the size of the proof's interpolant was proposed. We showed that changing the order in which clauses were resolved could greatly reduce the size of the corresponding interpolant. In this chapter, we expand on this idea by showing how multiple resolution proofs can be merged into a single, monolithic proof. We show how this monolithic resolution proof can be restructured in order to increase the amount of shared logic in the resulting interpolant.

4.3 General Method

Consider using the approach described in the previous section for generating functional dependencies in the case where many of functions contain the same support variables. Let functions $g_0, g_1, \ldots, g_n$ be the target functions and functions $x_0, x_1, \ldots, x_n$ be the variables in the potential support sets for these functions. Then the SAT instance to verify the existence of the ith functional dependency takes the following form:

$$\begin{aligned} A &= (g_i) \wedge (CNF_{\text{left}}) \\ B = (x_0 \equiv x_0^*) \wedge (x_1 \equiv x_1^*) \wedge \cdots \wedge (x_n \equiv x_n^*) \wedge (CNF_{\text{right}}) &\wedge (\bar{g}_i{}^*) \end{aligned}$$

Figure 4.3: A SAT instance that checks the existence of target function g_i with support set $x_0, x_1, \ldots, x_n$.

Here CNF_{left} and CNF_{right} represent the clauses for the circuit elements in the left and right halves of the circuit, respectively, as shown in Figure 2.15. The common variables between sets $\boldsymbol{A}$ and $\boldsymbol{B}$ $(x_0, x_1, \ldots, x_n)$ remain the same for each SAT instance. Also, the only clauses that differ among each of the individual SAT instances are the

(g_i) and $(\bar{g_i}*)$ terms. We refer to these terms as the "on" and "off" *assertion clauses*, respectively. Such large similarity between SAT instances can be leveraged to create similarities in the proofs of unsatisfiability. Structural similarities in the resolution proofs can then lead to structural similarities in the interpolants. Using these properties, we propose the following general method for creating a circuit structure that contains shared logic.

1. For each primary output, create a SAT instance verifying that the primary output can be expressed in terms of the circuit's primary inputs (Equation 4.3).
2. Generate a proof of unsatisfiability for each SAT instance created in the previous step.
3. For each node in each proof, color the node black if it has an assertion clause as an ancestor; otherwise color the node white.
4. Check to see if any black node in any proof can be resolved from any set of white nodes. If it can, color the node white.
5. Restructure the proofs so that the black nodes that were re-colored in step 4 are only resolved from white nodes.
6. Generate the interpolant for each proof.

Step 4 of the algorithm is illustrated graphically in Figures 4.4 and 4.5. Figure 4.4 contains two proofs whose interpolants are an implementation of functions g_0 and g_1, respectively. Figure 4.5 shows that two black nodes can be resolved from only white nodes present in both of the proofs.

4.4 Correctness

In this section, we argue the correctness of our method and elucidate it with examples. First we discuss our mechanism for deciding whether or not a node in a resolution

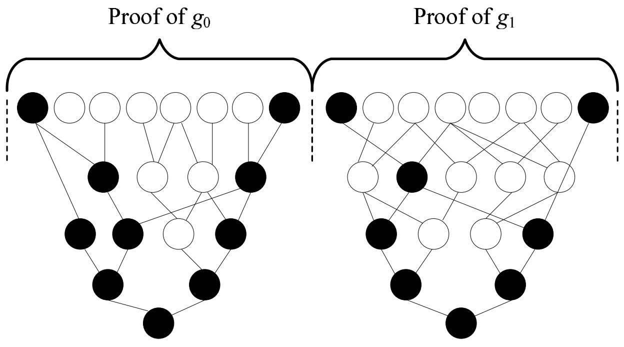


Figure 4.4: Two resolution proofs without any shared nodes.

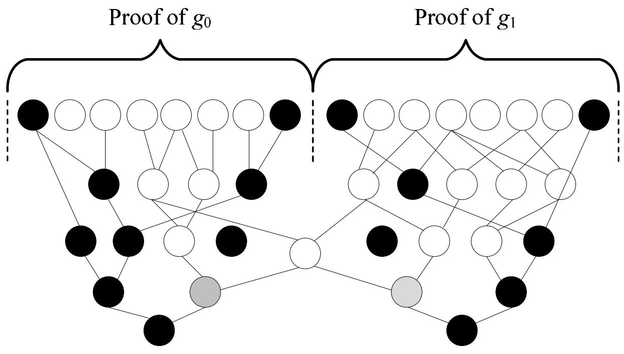


Figure 4.5: Two resolution proofs with a shared node. An additional white node is added to the proof, but two black nodes can then be removed. The gray nodes are nodes that were black but can now be colored white.

proof can be resolved from other nodes. This mechanism was discussed by Gershman in [42] and was used to reduce the size of interpolants in [50].

Proposition 13 *Let c be some clause and W be some set of clauses. Then c can be resolved from W if $W \wedge \bar{c}$ is unsatisfiable*

Proof 13 *The statement c can be resolved from W iff $W \rightarrow c$ is a tautology. Therefore, if there is no assignment of the variables present in W and c such that every clause in W evaluates to true and c evaluates to false, then c can be resolved from the clauses of W.* □

To perform Step 4 of our algorithm, we can simply repeatedly solve the SAT instance $W \wedge \bar{c}$, where W is the set of all the white clauses present in all of the resolution proofs and c is a black clause whose color we wish to change. If $W \wedge \bar{c}$ is unsatisfiable, a resolution proof of unsatisfiability will be returned by the SAT solver. This proof can then be modified by a procedure known as *bubble transformation*, described in [42], to show how c can be resolved by the clauses of W. The bubble transformation can then used to implement Step 5 of our algorithm.

Example 13 *Figure 4.6 shows a portion of a resolution proof. Let clauses $(a + b)$, $(a + b + \bar{c})$, and $(a + b + c)$ be colored black and the remaining clauses be colored white. In Step 3 of the algorithm we want to determine if clause $(a + b)$ can be colored white. Clause $(a + b)$ can be resolved from clauses $(a + \bar{e} + \bar{d})$, $(a + b + d)$ and $(a + b + \bar{d} + e)$ iff $(a + \bar{e} + \bar{d})(a + b + d)(a + b + \bar{d} + e) \rightarrow (a + b)$ is a tautology. Solving the SAT instance: $(a + \bar{e} + \bar{d})(a + b + d)(a + b + \bar{d} + e)(\bar{a})(\bar{b})$ verifies that $(a + b)$ can indeed be resolved from only white clauses.*

The bubble transformation method described in [42] can then be used to show the resolution steps needed to resolve $(a + b)$ from the white nodes. These resolution steps can then be used to restructure the proof as shown in Figure 4.7.

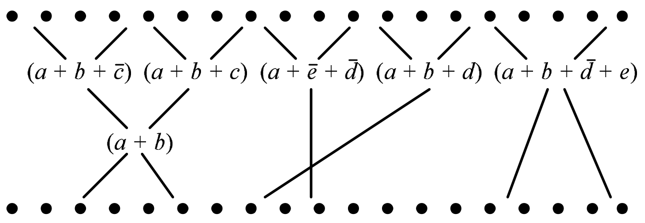


Figure 4.6: A portion of a resolution proof

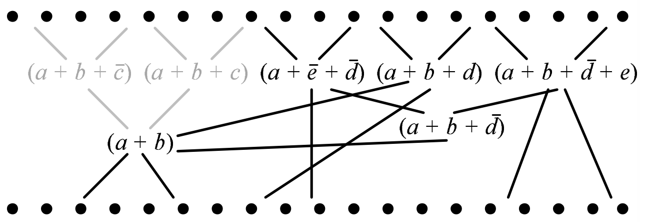


Figure 4.7: A portion of a resolution proof that has been restructured. Clause $(a + b)$ can be resolved from clauses $(a + \bar{e} + \bar{d})$, $(a + b + d)$ and $(a + b + \bar{d} + e)$. Restructuring the proof this way adds one clause and removes two others.

After the proofs are restructured, the interpolants are generated by calling the recursive procedure described in Figure 3.9 on each of the empty clauses present in the resolution proofs. We assert that these interpolants are *valid* in the sense that they still fulfill the same properties as interpolants generated from the non-modified resolution proofs.

Proposition 14 *The interpolants generated from the restructured proofs are still valid.*

Proof 14 *As described in the previous section, the only root clauses that differ among each resolution proof are the "on" and "off" assertion clauses. Therefore all the white root clauses are shared between the proofs. Step 5 of our algorithm only restructures the proof such that a previously black node is resolved from only white clauses. Therefore any white node present in the restructured proof is resolved from only root clauses that are common between each of the original proofs. Because the set of root clauses remains the same between each proof, the clauses present in* ***A*** *and* ***B*** *remain the same. Also,* *for each resolution proof, the common variables of **A** and **B** remain the same; therefore the interpolants generated in Step 6 are valid.*

□

4.5 Implementation and Results

We implemented steps 1 through 4 of our method in Berkeley ABC [32]. To test the potential savings of our algorithm, we performed trials on benchmarks in the IWLS benchmark collection [39]. Trials were run on a AMD Phenom^{TM} II X6 1090T machine with 3GB of RAM running 32-bit Linux. Only one core was utilized. In Table I, we present numbers on how many of the black nodes can be re-expressed in terms of only white nodes present in the proofs. These numbers allow us to gauge the potential for restructuring resolution proofs. The column "Original White" lists the number of white nodes originally present in the proofs. If a white node has an identical set of literals as another white node, these nodes are merged and counted as only one white node. The column "Original Black" lists the number of black nodes originally present in the proofs. The column "Checked" lists the number of black nodes that we checked to see if they could be considered white. The column "Fixable" lists the number of nodes that could be colored white out of the number of black nodes that we checked. The column "Percent Fixable" lists the percentage of the ratios of the "Fixable" to "Checked" columns. The "Time" column lists the time spent checking to see if black nodes could be colored white.

The time spent on each benchmark was limited to 200 seconds. In this experiment, all white nodes present in the proofs were considered when checking to see if a black node could be colored white (Step 4 of the algorithm). The more white nodes considered, the larger the SAT instance that needs to be solved in Step 4. Reducing this number to be a subset of the white nodes present in the proof will reduce the runtime; however this may also reduce the number of black nodes that can be colored white. This is a tradeoff that we plan to investigate in the future.

For most of the benchmarks, the percentage of black nodes that could be colored white was around 20–30%. The `table3` and `table5` benchmarks were two exceptions to this trend. Since trials on both of these benchmarks reached the timeout, only a subset of nodes were checked. The percentage was calculated as the number of fixable nodes divided by the number of nodes that were checked. It is likely that if a longer timeout had been used, then these percentages would be similar to the other benchmarks.

Perhaps the most salient result is that our algorithm scales well with the number of nodes in the resolution proof. Note that, unlike previous implementations, we are not applying interpolation to individual functions; rather we are applying it to each circuit in its entirety. For cases where we reach the computational limit, runtimes could be improved by applying the algorithm to smaller windows of logic within the circuit.

Number of Black Nodes That Can Be Colored White						
Benchmark	Original White	Original Black	Checked	Fixable	Percent Fixable	Time (s)
dk15	1743	581	581	175	30.12	0.04
5xp1	3203	1636	1636	275	16.81	0.18
sse	3848	2650	2650	563	21.25	0.28
ex6	4055	2731	2731	588	21.53	0.29
s641	6002	5148	5148	2269	44.08	0.46
s510	7851	5092	5092	1155	22.68	0.74
s832	15359	14826	14826	3358	22.65	3.67
planet	40516	43387	43387	10640	24.52	26.39
styr	44079	54128	54128	16578	30.63	33.88
s953	49642	46239	46239	12252	26.50	31.99
bcd	96385	109167	103514	34349	33.18	200.00
table5	137607	288461	69070	27848	40.32	200.00
table3	177410	283066	47279	24454	51.72	200.00

Table 4.1: Results of Steps 1 through 4 of our algorithm applied to a set of benchmarks. The timeout was set to 200 seconds.

4.6 Discussion

Given a resolution proof, determining whether or not a given node in the proof can be resolved from a set of other nodes is an easy task. Given an AIG, determining whether a given node in the graph can be expressed in terms of other nodes is difficult task. Accordingly, algorithms like AIG Rewriting and SAT-Sweeping can only make incremental improvements to small windows within an AIG [46, 47]. In contrast, the properties of resolution proofs allow for large transformations of the initial structure. The results presented in this chapter suggest that there is significant potential for clause sharing among resolution proofs for multiple target functions. Given an abundance of shared clauses, we expect interpolants with significant structural similarities. Even if these structures lead to logic that suboptimal, it will be much less *sparse* than traditional AIG representations; accordingly, such logic might be a promising starting point for the application of traditional logic synthesis.

Chapter 5

Using Cubes of Non-state Variables With Property Directed Reachability

5.1 Introduction

In the preceding chapters the topics mainly focused on problems in logic synthesis. However, in this final chapter we diverge slightly to consider a problem in formal verification. Specifically, we discuss a new symbolic model checking algorithm. Model checking refers the problem of automatically verifying that mathematical model of some real-world system exhibits a certain property. A system is generally modeled as a state transition system (a Kripke structure [51])[1] . A simple example is shown in Figure 5.1.

This transition system models a simple game of racquetball between two players. The game can start with either player one (P_1) or player two (P_2) serving. These initial states are indicated by the arrows entering the states with no origin states. The player who is serving is allowed to have one fault. If more than one fault occurs then the other

[1] Probabilistic model checking focuses on similar structures except transitions have probabilistic values and transitions can take either continuous our discrete amounts of time.

player begins serving. If the player has a successful serve (one or fewer faults) then the two players begin to volley. If a player makes an error (misses the return), then the opposite player begins serving.

This is a rather simple model for the game. There are no states to model the players score, and thus there are no terminating states. However, this simple model is useful for demonstrating some properties that might be useful to verify about the game. For example, one might want to verify that no player is allowed to return to the serve state directly after the fault state. Another important property one could verify is that no player is allowed to return the ball twice in a row.

These properties are expressed using some type of temporal logic such as LTL, CTL, or CTL* [52, 53, 54]. The properties expressed by these logics are generally categorized as either *safety* or *liveness* properties. Informally, a safety property asserts that *nothing bad happens* while a liveness property asserts that *something good will eventually happen.* Most symbolic model checking algorithms compute sets of forward and backward reachable states. These algorithms can be used to directly verify safety properties, and they can also be modified to check liveness properties [55].

This model only contains six states, and can easily be verified by explicitly walking the state graph. However, most *real world* systems may contain an intractable number of states. In order to verify properties of these systems, the state transitions are represented *symbolically.* Using symbolic techniques, properties have been verified on systems with over 10^{120} states [56]. Methods for symbolic model checking originally focused on the use of Binary Decision Diagrams (BDDs) [57, 58, 59]. These algorithms iteratively compute the image of a current set of states until a fixed point is reached, or the property is violated. While BDDs can be used to very quickly verify certain properties, often times the size of the BDD can explode during image computation. In the following subsections, we describe the most popular SAT-Based techniques for model checking.

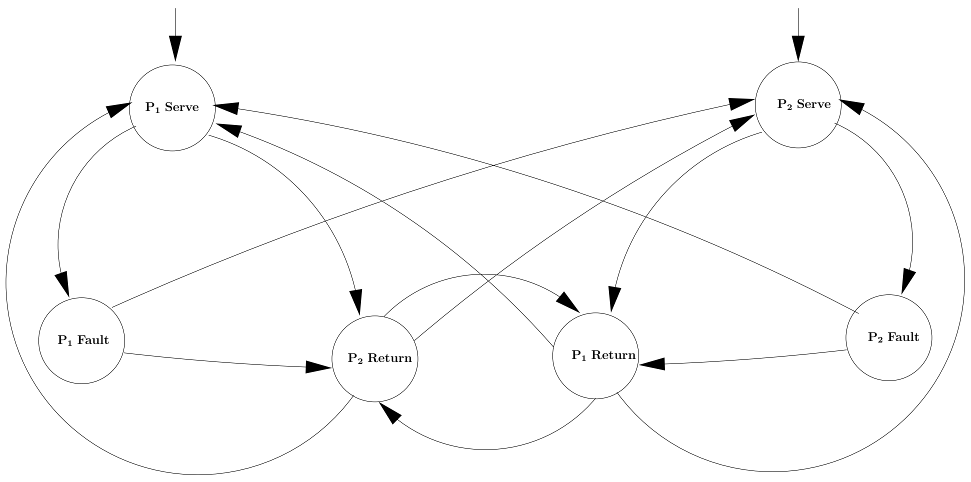


Figure 5.1: A state transition graph of a two player game of racquetball.

5.2 Background and Definitions

5.2.1 Definitions and Notation

We use magnitude symbols ($|c|$) to indicate the number of literals in a cube. A satisfiable assignment of a Boolean formula can be represented as a cube, and we will sometimes use the words cube and assignment interchangeably. We use the notation $c \models F$ if cube c is a satisfying assignment to F[2] .

A finite state transition system or finite state machine (FSM) $M = \{Z, X, I, T\}$ consists of a set of Boolean primary input variables Z, a set of Boolean state variables X, a set of initial states $I \subseteq \{0,1\}^X$, and a transition relation $T \subseteq \{0,1\}^X \times \{0,1\}^X$. The valuations of variables X ($s \in \{0,1\}^X$) represent possible *states* of the FSM; therefore we sometimes refer to assignments of X as states. We use an apostrophe (x') to indicate state variables in the next state of a transition relation. When we apply an apostrophe to a set (X') we are indicating that we are referring to the next state variables of the set. A set of states that can be reached in i transitions from the initial states are said to be among the states reachable in the ith *time frame* of the FSM. A superscript i

[2] Note that c may be a partial assignment of the variables of F. In this case, $F \wedge c$ is satisfiable for some assignment of the variables of F not in c.

(x^i) indicates that we are referring to state variable x in the ith time frame. When we apply a super script i to a set (X^i) we are indicating that we are referring to the state variables of the set X in the ith time frame.

An FSM's transition relation can be encoded as a circuit structure, which can then be translated into a CNF formula. This transformation can be done in linear time using the Teistin transformation [3]. After the circuit is transformed into a CNF formula, the formula contains current state variables $(x_0, x_1, \ldots, x_{n-1})$, next state variables $(x'_0, x'_1, \ldots, x'_{n-1})$, and variables representing the logic for the gates in T $(g_0, g_1 \ldots, g_{m-1})$. We will refer to $g_0, g_1 \ldots, g_{m-1}$ as *intermediate variables* or *gate variables*. This formula is satisfied only for assignments $x, x' \in \{0,1\}^X$ such that $(x, x') \in T$. We use the notation $T(X, X')$ to represent the set of clauses in this formula in terms of variables X and X'. Likewise, the set of initial states can be encoded as a CNF formula that is satisfied only for assignments $s \in \{0,1\}^X$ such that $s \in I$. We use the notation $I(X)$ to represent the set of clauses in this formula in terms of variables X. We will refer to a cube of only state variables as a *state cube* and a cube of gate variables (and possibly state variables as well) as a *gate cube*.

5.2.2 Bounded Model Checking

Bounded model checking (BMC) verifies properties up to a finite amount of transitions from an initial state. For example, given the state transition graph shown in Figure 5.1, bounded model checking could be used to verify that it is impossible to reach the P_2 *Fault* state within two transitions of the P_1 *Serve* state. These properties can be verified by converting the transition system's transition relation into a SAT instance and then unrolling it based on the number of transitions that the property should be verified over [60]. An instance of bounded model checking for FSM $M = \{Z, X, I, T\}$ is shown in Formula 5.1.

$$I(X^0) \wedge [\textstyle\bigwedge_{t=1}^{n} T(X^{t-1}, X^t)] \wedge [\textstyle\bigvee_{t=0}^{n} \neg P(X^t)] \tag{5.1}$$

The clauses of $P(X)$ are satisfied only for states that exhibit some sort of *property*. The SAT instance in Formula 5.1 is unsatisfiable if all of the states reachable within n transitions from I satisfy the property P. If it is possible to violate the property within n transitions, then Formula 5.1 is satisfiable. A satisfying assignment can be deciphered to determine a sequence of states that is sufficient to violate the property P.

Bounded model checking is arguably the most "straightforward" approach to verifying P, however it is not *complete* in the sense that P can only be verified up to n transitions. In order to verify that the property holds for all states, the number of transitions n must be at least as large as the longest shortest path from an initial state to an arbitrary state. A higher upper bound for n is the diameter of the state space of M[3] . This number is in the worst case exponential in the size of X, and is not usually known for most "real world" systems. This bound on the number of transitions can be improved for some models through the use of induction.

5.2.3 Induction

Mathematical induction is a common and powerful method used for mathematical proofs. Induction can be used in conjunction with a SAT solver to prove safety properties for transition systems. Unlike bounded model checking, induction can be used to prove properties hold through an infinite number of transitions without knowing the diameter of the transition system's state space.

Induction (sometimes called k-induction) is performed using a SAT solver through a number of subtly different algorithms [61]. These techniques unroll a circuit across multiple time frames and prove that a property holds over the longest loop-free path.

[3] The *diameter* of a graph is the "longest shortest path" between any two verticies. In the case of a state transition graph, the diameter is the longest shortest path between any two states.

Formula 5.2 is unsatisfiable if there is no loop free path within n transitions of FSM $M = \{Z, X, I, T\}$.

$$Loopfree(X, n) = [\bigwedge_{t=1}^{n} T(X^{t-1}, X^t)] \wedge [\bigwedge_{i<j\leq n}(X^i \neq X^j)] \tag{5.2}$$

The property P holds for M if there is no loop free path from an initial state that violates the property. This condition can be determined by increasing n until Formula 5.3 is unsatisfiable. Alternatively, the property P also holds if there is no loop free path longer than n states that reaches a state which violates the property. This condition can be determined by increasing n until Formula 5.4 is unsatisfiable.

$$I(X^0) \wedge Loopfree(X, n) \tag{5.3}$$

$$Loopfree(X, n) \wedge \neg P(X^n) \tag{5.4}$$

The basic induction algorithm works by iteratively solving Formulas 5.1, 5.3, and 5.4 while monotonically increasing n. If at any point Formula 5.1 becomes satisfiable then a counter example violating the property is extracted. However, if either Formula 5.3 or Formula 5.4 become unsatisfiable, then the property P holds from M.

The longest loop free path may be much longer than the diameter of the state space. Because the upper bound necessary to prove a property using BMC is indeed the diameter of the state transition graph, using this method of induction seems to be computational worse than just performing BMC. However, it is difficult to know what the diameter of the state space is for many transition systems, and the inductive method described above provides a means of indicating that indeed a large enough value of n has been chosen.

Furthermore, Formula 5.3 can be modified to verify that no initial state is visited more than once during the path[4] . To assert this condition, n can be increased until Formula 5.5 becomes a tautology[5] . Similarly, Formula 5.4 can be adjusted to assert that if the property holds for every state along the path, then it holds in the next state. To verify this property, n can be increased until Formula 5.6 becomes a tautology.

$$\neg I(X^0) \leftarrow [\textstyle\bigwedge_{i=1}^{n} \neg I(X^i)] \wedge Loopfree(X, n) \tag{5.5}$$

$$Loopfree(X, n) \wedge [\textstyle\bigwedge_{i=0}^{n-1} P(X^i)] \rightarrow P(X^n) \tag{5.6}$$

These modifications can improve the depth bounds for proving properties for some models, but the upper bound is still the longest loop free path in the model's state transition graph. An improvement that reduces this bound to the diameter of the state space is presented in [61], but this method requires quantifier elimination or the use of a QBF solver [62, 63].

5.2.4 Interpolation Based Model Checking

In the two preceding chapters we discussed how Craig Interpolation can be leveraged to generate functional dependencies. In 2003, McMillan proposed a SAT-Based method for model checking based on Craig's Interpolation Theorem [1]. In this algorithm, interpolants are used to compute an over approximation of the set of reachable states. Consider again the BMC SAT instance given by Formula 5.1. This formula can be split into two separate formulas referred to as a *prefix* and a *suffix*. The prefix of some FSM $M = \{Z, X, I, T\}$, shown in Formula 5.7, includes the constraints starting at the initial states and up to some transition i. The suffix, shown in Formula 5.8, includes the constraints starting at some transition i and up to some transition j. The suffix

[4] This optimization is useless for models that have only one initial state.
[5] The negation of a formula is unsatisfiable if and only if it is a tautology.

also includes the constraints asserting that the property is not satisfied for some state s_i such that $i \leq j$.

$$Pref(I,T,i) = (I(X^0) \wedge [\bigwedge_{t=1}^{i} T(X^{t-1}, X^t)] \tag{5.7}$$

$$Suff(T,P,i,j) = [\bigwedge_{t=i}^{j-1} T(X^t, X^{t+1})] \wedge [\bigvee_{t=i}^{j} \neg P(X^t)] \tag{5.8}$$

If for some $i < j \leq n$ Formula 5.9 is satisfiable, then a counter example can be extracted to show how P is violated. However, if Formula 5.9 is unsatisfiable, then an interpolant R can be created from the proof of unsatisfiability such that $Pref(I,T,i) \rightarrow R$ and $R \rightarrow \neg Suff(T,P,i,j)$. The only variables common to $Pref(I,T,i)$ and $Suff(T,P,i,j)$ are those representing the state variables X^i. Therefore R represents an over approximation of the states reachable in i transitions of $I(X)$ that are disjoint from states that may violate the property in $j - i$ transitions.

$$Pref(I,T,i) \wedge Suff(T,P,i,j) \tag{5.9}$$

Given this interpolant R, we can again check the satisfiability of Formula 5.9 with the exception that the set of initial states I is replaced by this Interpolant R. If this formula is satisfiable then the counter example can be verified by solving Formula 5.1. It is possible that the counter example is *spurious* because of the over approximation of the reachable states given by the interpolant R. However if the formula is unsatisfiable, then another interpolant R' can be extracted. If $R' \rightarrow R$ is a tautology, then the property P must hold because no additional states can be reached by transitioning from R, and every state in R satisfies the property P. This algorithm is described by psuedocode in Figure 5.2.

Interpolation can solve many instances faster than induction because the maximum number of transitions needed to prove a property is bounded by the diameter of the state

```
FiniteRun($I$, $T$, $P$, $k > 0$):
  *$I$, $T$, and $P$ are formulas
  *$I$ is the set of constraints representing the initial states
  *$T$ is the set of constraints representing the transition relation
  *$P$ is the set of constraints representing the property states
  ******************************************************
  if $I(X) \wedge P(X)$ is satisfiable then
    return $true$
  end if
  $R := I$
  while $true$ do
    $A := Pref(R, T, k)$
    $B := Suff(T, P, 0, k)$
    if $A \wedge B$ is satisfiable then
      if $R \equiv I$ then
        return $false$
      else
        return $maybe\ false$
      end if
    else
      * ($A \wedge B$ is unsatisfiable)
      $R' := interpolant(A, B)$
      if $R' \rightarrow R$ then
        return $true$
      end if
      $R := R \vee R'$
    end if
  end while
```

Figure 5.2: Pseudocode for the interpolation-based model checking algorithm given by McMillan [1]. Lines beginning with an asterisk (*) indicate comments. The function "$interpolant(A, B)$" returns an interpolant R' such that $A \rightarrow R'$ and $R' \rightarrow \neg B$. The algorithm returns $true$ if the property holds, $false$ if the property fails, and $maybe$ $false$ if the property might fail (the counter example might be spurious).

space. Furthermore, if the over approximation is a *good* in the sense that it accurately describes a set of reachable states, it is possible to prove some properties very quickly. However, interpolation has the drawback that if a counter example is discovered, it may be spurious. To block these spurious counter examples, interpolants may need to be generated from large unrollings of the circuit.

5.2.5 Property Directed Reachability

In 2011, Bradley proposed "Incremental Construction of Inductive Clauses for Indubitable Correctness" (IC3), a new SAT-Based method for symbolic model checking that does not require solving large unrollings, or generating messy abstractions of the reachable state space [64]. The technique works by iteratively solving a SAT-instance representing a single time frame of the underlying circuit. States that violate the property are recorded in the form of cubes of state variables. These cubes then must be blocked recursively by previous time frames. The process halts under two conditions; either a set of cubes extending from the initial state are found to reach a cube that violates the property, or the set of cubes blocked in one frame are shown to be blocked inductively in every future frame (proving the property to be invariant).

The IC3 algorithm works well when it is able to produce very small cubes (covering many states). Others have proposed an improved implementation of the algorithm referred to as "Property Directed Reachability" (PDR) [65]. One of the main improvements in PDR is the use of ternary valued simulation to reduce the size of state cubes. Using ternary valued simulation allows cubes to be shortened quickly without putting an unnecessary burden on the solver.

PDR embraces the benefits of both k-induction and interpolation-based model checking. The algorithm doesn't need to solve large unrollings, but instead solves different time frames in isolation from each other. Furthermore, the maximum number of time frames used for proving a property is bounded by the diameter of the model's state space. Often times properties can be proven with far fewer transitions than the state space's diameter.

From this point forward we will abuse some of the previously defined notation for the sake of readability. Specifically, we will stop explicitly listing the support variables of a set of constraints. We simply use I to implicitly represent the set of clauses $I(X)$ and T to represent the set of clauses $T(X, X')$. Similarly we use the notation P to represent the property constraints $P(X')$ for the next state transition. This notation is more convenient because the PDR algorithm only solves instances representing one transition at a time.

The algorithm operates on sets of clauses, denoted by F_i, called *frames*. The clauses in frame F_i symbolically encode an over approximation of the states that are reachable in the ith time frame of an FSM. In other words, if state s can be reached within i steps of the initial states, then $F_i \wedge s$ is satisfiable. The collection of these frames is referred to as the *trace*. The trace maintains the following properties.

1. The 0th frame only contains the initial states ($F_0 = I$)
2. Every assignment that satisfies the current frame also satisfies the next frame ($F_i \rightarrow F_{i+1}$)
3. Every state that can be reached in one transition from a state in the current frame, satisfies the next frame ($F_i \wedge T \rightarrow F_{i+1}$)
4. The property is satisfied in every frame except the last one ($F_i \rightarrow P$ for every F_i except F_n)

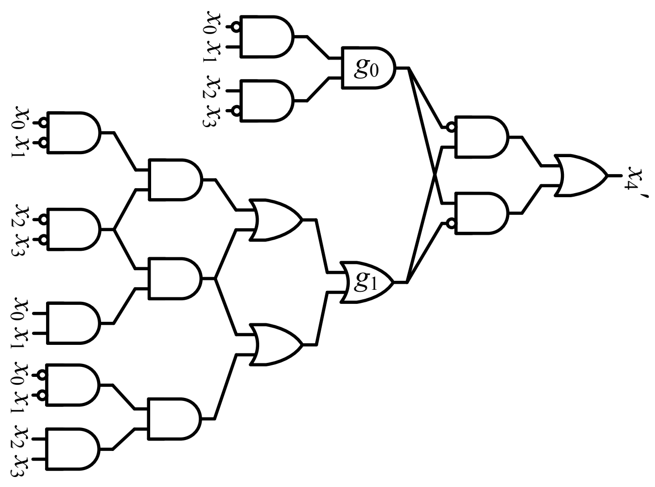
At the start of the algorithm, there exists just one frame $F_0 = I$. Each major iteration of the algorithm starts by checking to see if the property can be violated in one transition from the states in the highest frame. This is done by solving the SAT instance: $F_n \wedge T \wedge \bar{P}$. If this query is satisfiable, a satisfying state cube $s \models F_n \wedge T \wedge \bar{P}$ is extracted. The algorithm then proceeds to see if this cube can be blocked by the previous frame. This is done by solving the SAT instance: $F_{n-1} \wedge T \wedge s'$. If this query is satisfiable, a satisfying state cube is extracted from this SAT instance. The algorithm continues to try to block cubes in each previous frame. If, eventually, a cube cannot be blocked by the initial frame, F_0, a counter example is provided.

Whenever a cube is successfully blocked in some frame F_i, a clause blocking this cube is added to every frame F_k where $k \leq i$. This maintains property 2 of the trace. A clever improvement to this algorithm, given in [64], is to a priori add $\bar{s}$ to the query: $F_{n-1} \wedge \bar{s} \wedge T \wedge s'$. This improves the chances of blocking s in F_i and is sound because $F_0 \rightarrow \bar{s}$ and $F_i \rightarrow F_{i+1}$.

Once the query: $F_n \wedge T \wedge \bar{P}$ is unsatisfiable, a new frame F_{n+1} is created and the propagation phase begins. During this part of the algorithm, cubes learned in previous frames are attempted to be blocked in later frames. This is accomplished by repeatedly solving $F_i \wedge T \wedge \bar{c}'$ for each clause $c \in F_i$. If this formula is unsatisfiable, than c can be added to frame F_{i+1}. If at any point two frames become identical (contain the same clauses), then an invariant is proved. Because $F_i \equiv F_{i+1}$ and $F_i \rightarrow F_{i+1}$, any clause present in F_i can be added to all future frames, therefore the property will hold in all future frames because $F_i \rightarrow P$.

5.2.6 Property Directed Reachability Beyond State Variables

In this chapter, we extend the framework of PDR to allow for cubes containing *functions* of state variables. This idea is illustrated with an example in Figure 5.3. Figure 5.3 shows a truth table for an incompletely specified Boolean function for next state variable x_4' and a circuit level implementation of x_4'. Assume that x_4 must be blocked in the next frame. There are four cubes of state variables in the current frame that need to be blocked to make this so: $\bar{x_0} \wedge \bar{x_1} \wedge \bar{x_2} \wedge \bar{x_3}$, $\bar{x_0} \wedge \bar{x_1} \wedge x_2 \wedge x_3$, $\bar{x_0} \wedge x_1 \wedge x_2 \wedge \bar{x_3}$, and $x_0 \wedge x_1 \wedge \bar{x_2} \wedge \bar{x_3}$. In this example, none of these cubes can be simplified to smaller cubes. However, there are only two cubes in terms of variables g_0 and g_1 that must be blocked to prevent the justification of x_4 in the next frame: $\bar{g_0} \wedge g_1$ and $g_0 \wedge \bar{g_1}$. In this example, only half as many cubes in terms of intermediate logic variables need to be added to the current frame in order to block x_4'.



x_0, x_1, x_2, x_3	g_0, g_1	x_4'
0 0 0 0	0 1	1
0 0 1 1	0 1	1
0 1 1 0	1 0	1
1 1 0 0	0 1	1

Figure 5.3: An example netlist where fewer cubes in terms of intermediate variables need to be blocked than cubes in terms of state variables

In this work, we study the affect of extending PDR to allow cubes of intermediate logic variables. We then present the results of our implementation on the HWMCC '11 benchmarks [66].

5.3 Extending Cubes To Gate Variables

5.3.1 General Concept

In the previous implementations of the algorithm, states are symbolically encoded as cubes of state variables [65, 64, 67]. However, as Figure 5.3 demonstrates, there may be a significant advantage to allowing cubes in terms of intermediate variables. This extension does not change the conceptual flow of the algorithm greatly, but some care does need to be taken when choosing which intermediate variables to allow.

Figure 5.4 shows the transition relation that is used in the standard implementation of PDR. The logic in the transition relation can be partitioned into two types of logic: logic that only contains state variables in its transitive fanin (shown in grey) and logic that contains both state variables and primary input variables in its transitive fanin (shown in white). Because the grey logic only contains state variables in its transitive fanin, it is unaffected by valuations of the primary input variables. Consider the gates that are on the boundary between the grey and white logic. We use the set $G = \{g_0, g_1, \ldots, g_{k-1}\}$ to denote these gate variables. Note that some state variables may fanout directly into "white logic" shown in Figure 5.4. In this case, we consider these state variables to be among the set of variables G.

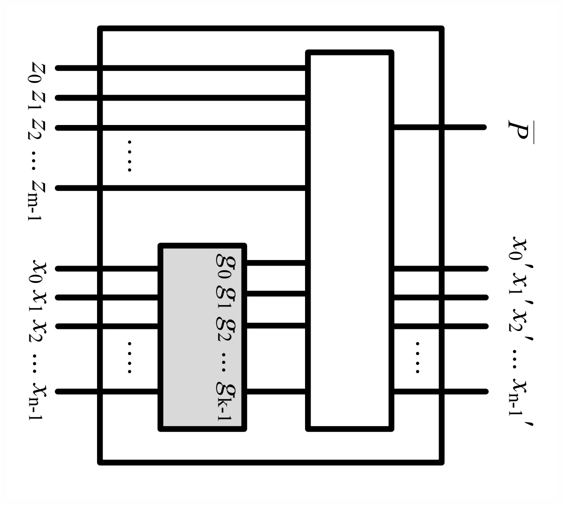

$z_0\ z_1\ z_2 \ldots z_{m-1}$
$x_0\ x_1\ x_2 \ldots x_{n-1}$
$g_0\ g_1\ g_2 \ldots g_{k-1}$
$\overline{P}$
$x_0'\ x_1'\ x_2' \ldots x_{n-1}'$

Figure 5.4: A transition relation that is used in a frame for the standard PDR implementation

Proposition 15 *Let S be the set of all cubes of state variables that can reach cube m in one transition from frame i. Formally: $S = \{s \in \{0,1\}^X : s \models F_i \wedge T \wedge m'\}$. Let W be the set of all cubes of the variables in G that can reach m in one transition from* *frame i. Formally: $W = \{w \in \{0,1\}^G : w \models F_i \wedge T \wedge m'\}$. Then cube m can be blocked in Frame $i+1$ if and only if all cubes of S are blocked in frame i or all cubes of W are blocked in frame i. Formally: $(m' \not\models (\bigwedge_{s \in S} \bar{s}) \wedge F_i \wedge T) \leftrightarrow (m' \not\models (\bigwedge_{w \in W} \bar{w}) \wedge F_i \wedge T)$.*

Proof 15 *Note that for every state variable $x \in X$ shown in Figure 5.4, every path from x to any $x' \in X'$ must pass through at least one gate variable in G. This means that there must be a surjection between the cubes in S and the cubes in W. i.e., every assignment in S maps to some assignment in W and every assignment in W maps to some subset of assignments in S. Since this surjection exists every assignment $a \models F_i \wedge T \wedge m'$ must satisfy at least one cube in W and S. Likewise, every cube in W and S must satisfy every assignment $a \models F_i \wedge T \wedge m'$.*

□

Proposition 15 allows us to pick a restricted set of gate variables to be used as state cubes. However, in order to block cubes of state variables, we must modify the transition relation shown in Figure 5.4 to be of the form shown in Figure 5.5. The transition relation in Figure 5.5 is a sort of *half unrolling* of the standard transition relation. The current state variables $(x_0, x_1, \ldots, x_{n-1})$ are replaced by outputs of the gate variables G in the current frame $(g_0, g_1, \ldots, g_{k-1})$. Also, the next state variables $(x'_0, x'_1, \ldots, x'_{n-1})$ extend to the outputs of the gate variables G in the next frame $(g'_0, g'_1, \ldots, g'_{k-1})$.

In the standard transition relation, cubes of variables of X are always extracted from the satisfying assignment. However, in our implementation we allow for cubes of variables of G. in order to block these cubes in the next time frame, the transition relation must be able to justify cubes of variables G'. As Figure 5.5 suggests, this does not increase the amount of logic in the transition relation, it only changes the placement of the logic.

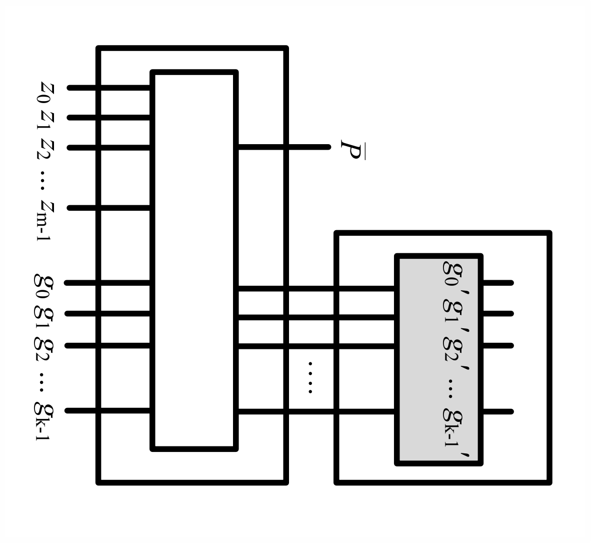


Figure 5.5: A transition relation that is used for blocking cubes of gate variables

5.3.2 Details for Ternary Valued Simulation

The use of ternary valued simulation to reduce state cubes increases the performance of the algorithm tremendously [65]. The reduction of these state cubes is accomplished by first finding a satisfying assignment $a \in \{0,1\}^{X \cup Z}$ to the query $F_i \wedge T \wedge m'$. The transition relation shown in Figure 5.4 is then simulated with the values of a. A cube s of state variables in the transitive fanin of the variables of m' is chosen from this assignment. Then, one by one, the value of each state variable $x \in s$ is replaced by the unknown value $\perp$. Each time the value is replaced, the transition relation is re-simulated. If the values of m' remain the same through this simulation of the transition relation, then x is removed from the cube s, and the value of x remains at $\perp$ in the transition relation. If some value in m' changes, then x is restored to its original value, and x remains in s.

Ternary valued simulation can also be applied to cubes containing gate variables, but some care must be taken. The order in which the cube variables are set to $\perp$ can greatly affect the number of variables removed from the cube. This is illustrated in

Figure 5.6. Consider the cube $g_0 \wedge g_1 \wedge g_2$ which must be blocked in order to prevent state $m'_0 \wedge m'_1 \wedge \bar{m}_2{}'$ from being reached. If g_0 is set to $\perp$, then the $\perp$ value reaches variables m'_0 and m'_2, so g_0 remains in the cube. Likewise if g_1 is set to $\perp$, then the $\perp$ value reaches variables m'_1 and m'_2. Gate g_2 also exhibits the same problem, a $\perp$ value on g_2 reaches variable in m'_2. Using this simulation order, one would conclude that all of the variables g_0, g_1 and g_2 must be kept in the cube in order to block $m'_0 \wedge m'_1 \wedge \bar{m}_2{}'$.

However, this is clearly not the case. The values of g_0 and g_1 are enough to resolve the value of g_2. As long as the value of a gate variable is uniquely determined by its inputs, then this variable can be removed from the cube. To solve this problem we propose the following ternary simulation algorithm.

1. Sort the variables in s in ascending order by the variable's logic level in the transition relation and set $i = 0$.

2. If $i = |s|$, then return s. Otherwise, for the ith variable v_i in s do the following: If v_i is a state variable, proceed to step 4. Otherwise, proceed to step 3.

3. If the value of v_i can be determined to be 1 or 0 (not $\perp$) by its fanins, then remove v_i from s and go back to step 2. Otherwise proceed to step 4.

4. Set v_i to $\perp$ and simulate the transition relation. If no variable in m' evaluates to $\perp$, then remove v_i from s and proceed to step 2. Otherwise, set v_i back to its original value, re-simulate the transition relation, increment i, and proceed to step 2.

Restricting the cubes to only contain state variables prevents the problem illustrated in Figure 5.6 because every variable is at logic level 0 in the transition relation. Therefore the value of one variable does not have any influence on the value of another. By enumerating the cube variables in order by logic level, we can eliminate variables whose value is clearly determined by previously simulated variables.

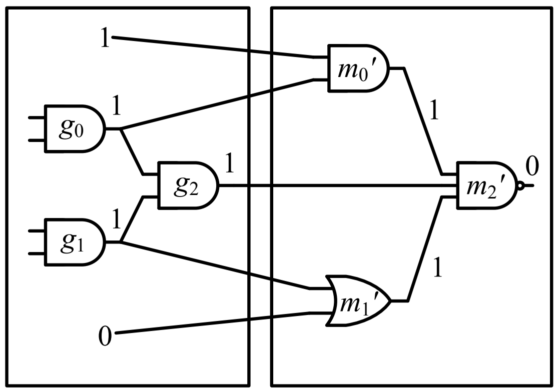


Figure 5.6: An example used to illustrate subtleties when using ternary valued simulation on cubes of gate variables

5.4 Experiment and Results

5.4.1 Experiment Setup

To test our approach, we modified the version of PDR implemented in Berkeley ABC [32, 65]. We changed the ternary valued simulation portion of the implementation in the following way:

- The ternary valued simulation algorithm is run twice, once allowing cubes of gate variables in the cone-of-influence (COI) of the next state variables and once only allowing cubes of state variables in the COI of the next state variables.
- In the first pass, the cube containing gate variables is reduced. The order in which cube variables are set to $\perp$ is determined by the logic level of the gate variable (as discussed in Section 5.3.2) and by a priority assigned to each variable. A variables priority increases if it is unable to be removed from the cube; otherwise, it decreases. Variables with higher priority then have a greater chance of being removed in later rounds of ternary valued simulation.
- In the second pass, a cube containing only state variables is reduced. In this case, the order in which cube variables are set to $\perp$ is determined by a fixed ordering.

- At the end of both passes, whichever cube is smaller (containing fewer literals) is returned.

To compare the performance of allowing gate cubes, rather than just state cubes, we run this version of the algorithm (which we refer to as the *gate cube* version) on the HWMCC '11 benchmarks [66]. We compare the results against a second implementation, which is the same, except only state cubes are allowed in both passes of the ternary valued simulation. We refer to this version of the implementation as the *state cube* version. This way both implementations have two chances to reduce cubes during ternary valued simulation, but only one is allowed to return cubes of gate variables. The runtime differences between the two versions should then be more heavily influenced by types of cubes, rather than by the priority scheme that we introduced. The transition relation is converted into a CNF formula using the standard Tseitin transformation [3] (as opposed to a transformation using variable elimination like in [65, 68]). This was done for the sake of an easier implementation. There are no conceptual problems with implementing our solution with other CNF transformations.

All of the benchmarks were run on a 4-core Intel® Core™ i7-2600 CPU @ 3.40GHz with 8GB of RAM. Only one core was utilized for each benchmark.

5.4.2 Results

The results of our experiment are displayed in Tables I and II. Table I lists benchmarks where the property was proved not to hold (satisfiable benchmarks) and Table II lists benchmarks where the property does hold (unsatisfiable benchmarks). We omitted benchmarks that took less than 10 seconds for both of the methods to solve. We set a timeout of 2 hours (7200 seconds) for all of the benchmarks, and we did not include results where both methods timed out[6] . The "Benchmark" column lists the name of the benchmarks. The "Time States" column lists how long it took to solve the benchmark

[6] We let benchmark `cmudme1` run slightly longer. In this case, the state cube version had still not completed after 3 hours.

with the state cube version. The "Frames States" column lists how many frames were needed to solve the benchmark with the state cube version. Likewise, the "Time Gates" column lists the time it took to solve the benchmark with the gate cube version, and the "Frames Gates" column lists how many frames were needed to solve the benchmark with the gate cube version. The column "Time Ratio" lists the ratio between the "Time Gates" and "Time States" columns; a geometric average of these values is provided at the bottom of the tables. Benchmarks that timed out were not included in this average. In Table II, The "Inv. States" and "Inv. Gates" columns list the number of clauses in the invariant for the state cube and gate cube versions, respectively.

The results demonstrate that for satisfiable benchmarks, allowing cubes of gate variables seems to have a generally positive affect on the performance of the algorithm. The best time ratio for the gate cube approach is .31 (which corresponds to a 3.2X runtime improvement for the benchmark `bc57sensorsp3`). The average time ratio for satisfiable benchmarks is .82 (which corresponds to an average speedup of 1.21X). For the unsatisfiable benchmarks, the performance is not as reliable. The best performance increase was seen by the benchmark `6s34`; which has a time ratio of .26 (a 3.85X speedup). For some of the benchmarks, the performance was close to the same (e.g., `bjrb07amba3andenv` and `nusmvguidancep7`). This likely indicates that both versions of the algorithm frequently chose cubes from the second pass of ternary simulation. Other benchmarks had large performance differences between the gate cube and state cube versions of the algorithm. We initially hypothesized that the structure of the benchmarks circuits would benefit one version of the algorithm over the other. For example, perhaps benchmarks that are more *narrow* (many logic levels with very few gates in each level) might have better performance for the gate cube implementation. However, this does not seem to be the case. Many of the benchmarks are the same underlying circuit, but prove different properties. Yet, different properties for the same circuit can be proved significantly faster using different versions of the algorithm. This is demonstrated with the `bc57sensor` and `bjrb07amba6andenv` benchmarks.

Satisfiable Benchmarks					
Benchmark	Time States (s)	Frames States	Time Gates (s)	Frames Gates	Time Ratio
`abp4p2ff`	12.34	17	**6.57**	15	0.53
`abp4ptimoneg`	22.46	18	**19.99**	17	0.89
`bc57sensorsp0`	353.59	59	**248.85**	41	0.70
`bc57sensorsp0neg`	**339.25**	62	353.39	55	1.04
`bc57sensorsp1`	**217.01**	59	550.17	73	2.53
`bc57sensorsp1neg`	595.57	63	**428.22**	47	0.72
`bc57sensorsp2`	468.53	69	**274.03**	63	0.59
`bc57sensorsp2neg`	**460.64**	79	586.85	71	1.27
`bc57sensorsp3`	731.42	67	**227.82**	58	0.31
`intel017`	—	—	**4878.43**	232	—
`intel046`	2274.65	68	**2191.27**	70	0.96
`intel045`	2101.11	70	**1810.71**	70	0.86
`intel047`	**1371.72**	62	2643.31	69	1.93
`irstdme4`	68.67	31	**21.67**	26	0.32
`irstdme5`	19.46	26	**6.46**	26	0.33
`irstdme6`	31.84	29	**17.94**	28	0.56
`nusmvtcasp5n`	99.66	24	**78.58**	24	0.79
`nusmvtcastp5`	77.46	22	**67.15**	23	0.87
`prodcellp0neg`	**77.71**	60	105.54	78	1.36
`prodcellp1`	155.62	72	**141.58**	72	0.90
`prodcellp1neg`	**96.87**	64	145.41	63	1.50
`prodcellp2`	181.08	60	**141.76**	81	0.78
`prodcellp2neg`	143.65	82	**114.42**	62	0.80
`prodcellp3`	117.70	58	**102.05**	56	0.87
`prodcellp4`	146.54	66	**143.28**	75	0.98
`prodcellp4neg`	**162.73**	80	526.80	62	3.23
Geometric Average	—	—	—	—	**.82**

Table 5.1: Results of running the state cube and gate cube implementations on satisfiable benchmarks

5.4.3 Other Heuristics We Tried

Here we note a few other things we tried to improve the algorithm's runtime.

- We introduced a probabilistic scheme for choosing gate variables. The idea was rather than just considering gates on the boundary between the grey and white blocks in Figure 5.4, we would probabilistically consider all gates in the grey block. This generally yielded better results for benchmarks that the gate cube version performed poorly on, but it also yielded worse results for benchmarks that the gate cube version performed well on.
- We introduced a kind of simulated annealing [69] style approach, where the probability scheme mentioned above was used, but the probability decreased geometrically until a new frame was added to the trace. The intuition behind this idea was that if the algorithm was struggling to prove that the property held up to a certain frame using cubes of gate variables, it would slowly start considering cubes of only state variables. The performance of this scheme was similar to that of the previously mentioned flat probabilistic scheme.
- We tried limiting the height of the logic that was added to cubes. For instance, we would only include gates in cubes that were of logic level n or lower. This again yielded similar results as the two previously mentioned heuristics.

5.5 Discussion

The results demonstrate that allowing cubes of gate variables can cause very drastic performance changes between the benchmarks. In general we find that the gate cube version of PDR works better for satisfiable benchmarks, but the trend is not as clear for unsatisfiable benchmarks. The variation in the results indicates that perhaps there exists a better heuristic for choosing what logic to include in a certain cube. Regardless, the results show that the gate cube version of the algorithm does perform much better in many of the benchmarks.

One disadvantage to using gate cubes is that it can greatly increase the size of the CNF formula being solved in certain frames. If cubes are expressed only in terms of state variables, then when solving the query: $F_i \wedge T \wedge m'$, only the logic of T in the transitive fanin of the variables of m' needs to be included in the SAT instance. The standard implementation periodically cleans up each SAT instance to only include the relevant logic for the current query [65]. However, when cubes of gate variables are included in each frame F_i, then the logic corresponding to these gate variables must be included in every query involving F_i (regardless of whether or not the gate variables in a cube are in the transitive fanin of m'). This theoretically could make the gate cube version slower for most of the benchmarks, but empirically this did not seem to have much of a negative effect.

Unsatisfiable Benchmarks							
Benchmark	Time States (s)	Frames States	Inv. States	Time Gates (s)	Frames Gates	Inv. Gates	Time Ratio
6s2	46.67	13	601	**46.35**	13	601	0.99
6s34	3984.89	77	2284	**1053.82**	89	1057	0.26
6s6	**19.19**	18	1709	21.59	21	1796	1.13
bj08amba2g3f3	**1.12**	10	44	14.37	10	48	12.83
bjrb07amba10andenv	2081.17	11	204	**2024.49**	9	246	0.97
bjrb07amba3andenv	10.56	9	103	**10.07**	8	73	0.95
bjrb07amba4andenv	54.87	7	78	**31.60**	7	66	0.58
bjrb07amba5andenv	93.45	8	130	**79.50**	8	109	0.85
bjrb07amba6andenv	**277.88**	8	160	438.81	8	192	1.58
bjrb07amba7andenv	176.79	11	148	**160.12**	9	159	0.91
bjrb07amba9andenv	**974.83**	9	253	1013.88	14	185	1.04
bob05	**69.06**	18	407	101.86	21	505	1.47
bobcohdoptdcd4	84.73	17	1417	**51.61**	13	1144	0.61
bobsmi2c	**48.03**	50	1121	193.33	187	1183	4.03
cmudme1	—	—	—	**7342.07**	90	7148	—
cmudme2	4255.06	97	6197	**2917.46**	99	4746	0.69
eijkbs1512	48.11	176	303	**31.36**	161	312	0.65
eijks382	**28.02**	57	368	34.35	69	380	1.23
eijks420	**64.07**	464	161	67.97	501	161	1.06
eijks444	407.75	62	429	**237.49**	61	394	0.58
eijks526	142.32	64	509	**80.75**	62	500	0.57
intel006	**23.96**	11	543	25.11	13	615	1.05
intel007	243.70	10	1580	**193.47**	8	1554	0.79
intel026	2262.39	52	6504	**2183.98**	49	6239	0.97
intel054	**239.29**	19	1048	305.00	21	1129	1.27
intel055	**69.98**	17	309	70.03	19	211	1.00
intel056	146.26	23	835	91.82	22	720	0.63
intel057	144.43	22	536	**134.20**	20	494	0.93
intel059	**62.88**	18	543	66.52	20	536	1.06
intel062	2095.27	28	4101	**2077.67**	26	4909	0.99
nusmvguidancep5	23.98	18	292	**26.35**	18	292	1.10
nusmvguidancep7	**77.49**	21	628	78.24	21	628	1.01
nusmvguidancep8	**19.90**	22	155	20.17	22	155	1.01
nusmvguidancep9	21.25	20	154	21.25	20	154	1.00
nusmvreactorp2	**1093.58**	172	4437	2394.32	7389	7389	2.19
nusmvreactorp6	**1416.97**	163	4390	1444.97	163	4390	1.02
pdtpmscoherence	68.82	16	1491	**44.50**	12	1217	0.65
pdtpmsheap	18.72	25	653	**11.47**	23	503	0.61
pdtpmsretherrtf	278.97	51	2510	**48.65**	43	925	0.17
pdtpmsvsar	25.64	11	263	**12.90**	11	260	0.50
pdtswvibs8x8p1	18.20	20	867	**13.45**	21	813	0.74
pdtswvqis10x6p1	50.05	73	192	**35.91**	66	208	0.72
pdtswvqis8x8p1	**108.93**	56	285	187.01	62	339	1.72
pdtswvroz10x6p1	**12.84**	58	73	12.99	56	73	1.01
pdtswvroz10x6p2	117.03	88	136	**105.51**	76	166	0.90
pdtswvroz8x8p1	13.54	50	60	**12.43**	50	64	0.92
pdtswvroz8x8p2	**62.52**	71	183	90.46	60	135	1.45
pdtswvsam6x8p3	24.84	40	284	**24.78**	39	311	1.00
pdtswvtma6x4p2	**186.35**	52	1537	221.31	60	1758	1.19
pdtswvtma6x4p3	**1006.19**	58	6150	1754.36	64	7837	1.74
pdtswvtma6x6p1	297.39	50	1191	**184.71**	48	1187	0.62
pdtswvtma6x6p2	**1573.62**	69	5944	1851.38	69	7054	1.18
pdtswvtms10x8p1	107.81	16	1735	**97.07**	15	1521	0.90
pdtswvtms12x8p1	**70.52**	16	1531	201.81	37	1625	2.86
pdtswvtms14x8p1	**73.40**	16	1362	81.38	23	1353	1.11
pdtvisbakery0	**30.29**	32	42	32.83	32	42	1.08
pdtvisbakery1	**17.68**	21	47	32.17	31	46	1.82
pdtvisbakery2	27.42	32	43	**25.01**	27	47	0.91
pdtvisgoodbakery0	**24.07**	27	44	26.79	28	40	1.11
pdtvisgoodbakery1	**19.64**	25	46	38.00	26	55	1.93
pdtvisgoodbakery2	**15.28**	25	43	26.12	27	47	1.71
pdtvisns3p00	**15.51**	11	99	23.34	11	111	1.50
pdtvisns3p01	12.55	12	68	**12.35**	11	82	0.98
pdtvisns3p02	36.21	14	125	**18.30**	14	99	0.51
pdtvisns3p03	**7.14**	9	62	15.22	12	87	2.13
pdtvisns3p04	20.97	13	100	**16.64**	11	97	0.79
pdtvisns3p05	31.82	14	126	**13.44**	12	82	0.42
pdtvisns3p06	29.23	11	124	**20.99**	11	90	0.72
pdtvisns3p07	19.04	16	91	**10.28**	12	84	0.54
pdtvistimeout0	**4649.41**	35	16496	—	—	—	—
pdtvisrethersqo4	49.12	38	454	**31.97**	38	450	0.65
pdtvisvending01	**54.20**	16	1176	58.74	19	1102	1.08
Geometric Average	—	—	—	—	—	—	**0.98**

Table 5.2: Results of running the state cube and gate cube implementations on unsatisfiable benchmarks

Chapter 6

Future Directions

Since the late 1990s the power of SAT solvers has grown tremendously, and even recently more promising techniques and heuristics have been introduced that make vast improvements [70, 71, 72]. As the power of SAT solvers increases, more and more hard problems in logic synthesis and verification will become tractable. In this final chapter we briefly mention some possible future directions of research in regards to the topics presented in this thesis.

6.1 Cyclic Combinational Circuits

The algorithms presented in this thesis that perform analysis on cyclic circuits scale well with the size of the circuit. However, we consider generating a *good* set of candidate circuits to analyze an unsolved problem. The algorithm we presented in Section 2.4 generates candidate circuits with a cost related to the size of the support set of the functions. We mentioned that this method might work well for mapping to a technology that can represent any function in terms of a fixed number variables (like FPGAs). The algorithm that we presented wont necessarily translate well to technologies that implement elementary logic functions (like AND and OR gates).

Currently, one of the best performing synthesis algorithms is DAG-Aware AIG Rewriting [46]. This algorithm works by enumerating different *cuts* of a netlist. A *k-feasible* cut of a node n in a network is a selection of nodes in the transitive fanin of n such that the number of leaves of the selection does not exceed k. The enumerated cuts are replaced by pre-computed functions of k variables, and then a series of balancing operations are performed on the network to reduce the number of gates. One might suspect that a similar method could be applied with a series of pre-computed cyclic cuts, however this is not as straight forward as the acyclic case. One problem is that analysis would need to be applied after each cut is replaced. This isn't so much of an issue because we have discussed how efficient analysis can be performed in Chapter 2.

However, a second issue with this approach is finding cyclic combinational circuits that implement a single Boolean function (and are smaller than any acyclic equivalent). DAG-Aware AIG Rewriting makes use of a pre-computed library of compact implementations for specific functions. It is unclear of whether or not there even exists cyclic combinational circuits implementing a single Boolean function that are actually smaller than any acyclic equivalent. All of the arguments about lower bounds for certain classes of circuits made in Riedel's dissertation were in reference to circuits implementing multiple functions [8]. We make the conjecture that such cyclic combinational circuits do not exist. This intuition strings from the circuit shown in Figure 6.1

Figure 6.1 shows a hypothetical cyclic combinational circuit that contains feedback at its primary output. However, the feedback that occurs at the output must be *useless*. Given some assignment to the primary inputs $x_0, x_1, x_2, \ldots x_{n-1}$, the initial state of f must be $\perp$. Eventually f must evaluate to a definite value of 0 or 1 (we are assuming that the circuit is combinational). Once this occurs, the value on the wire feeding back into the circuit is essentially worthless; the output has already reached a stable value.

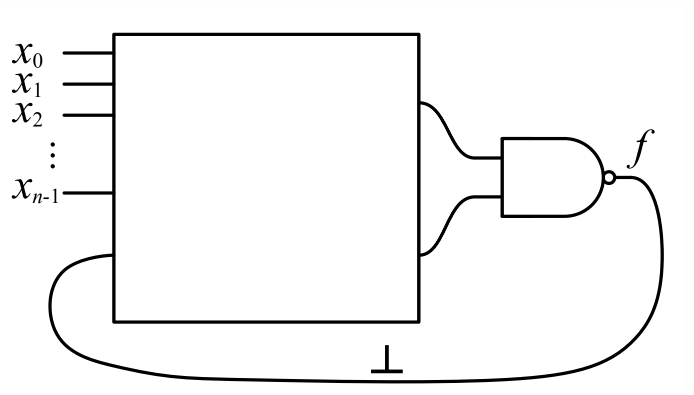


Figure 6.1: A cyclic combinational circuit that must be larger than some equivalent acyclic circuit.

This example proves that for circuits containing a single primary output it never makes sense to have feedback at the output. However, what about single output circuit containing feedback internally? Such a circuit might look something like the one described in Figure 6.2.

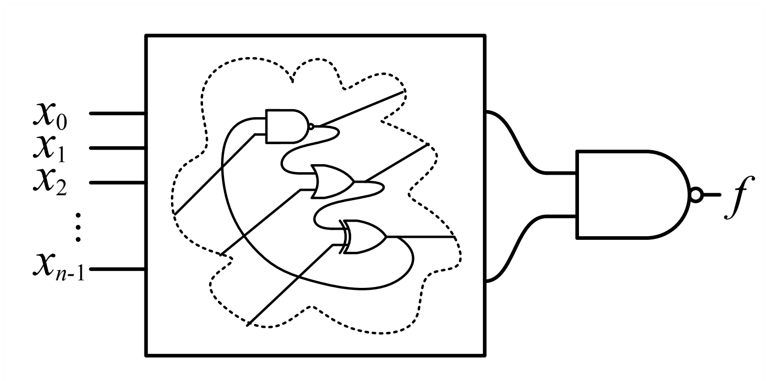


Figure 6.2: A cyclic combinational circuit that might be smaller than all equivalent acyclic circuits.

Even if there exists no cyclic combinational circuits with a single primary output that is smaller than any acyclic equivalent, this doesn't mean a methodology similar to DAG-Aware AIG rewriting could not be applied to cyclic structures. It might be the case that pre-computed functions may need to contain multiple outputs. However, this would dramatically increase the complexity of both the cut enumeration portion of the algorithm and the ability to pre-compute the library functions.

6.2 Reducing Interpolants and Resolution Proofs

In Section 5.2.4 we discussed how Interpolation is used in symbolic model checking. For this application, interpolation can produce messy over-approximations that generate spurious counter examples. Note that our methods for minimizing interpolant size do not directly apply to the problem of producing smaller over-approximations. One goal for future work could be to extend our methodology to this domain. Perhaps the resolutions can be biased to involve certain variables that could positively influence the structure of the abstraction provided by interpolation.

Other optimizations that could be investigated include methods for generating better initial resolution proofs from the SAT solver. We have noticed that changing the order of variable decisions in the SAT solver can significantly reduce the initial size of the resolution proof. Much of the current research in SAT solving pertains to heuristics for making variable decisions that will lead to solving SAT instances *faster* [33, 71, 73]. Perhaps some of this intuition could be applied to variable decision heuristics that result in resolution proofs that produce smaller interpolants.

6.3 Property Directed Reachability

The Property Directed Reachability algorithm is a very young; many new applications and heuristics will be proposed. Some recent research has gone into extending PDR to other theories [74, 75]. Extensions to some of the work presented in this thesis might focus on different heuristics for choosing what logic to include in each cube and different methods for cube minimization. Other work might focus on how PDR can be applied to probablistic symbolic model checking problems [76, 77].

References

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

[2] H. Jin and F. Somenzi. Circus: A hybrid satisfiability solver. In HolgerH. Hoos and DavidG. Mitchell, editors, *Theory and Applications of Satisfiability Testing*, volume 3542 of *Lecture Notes in Computer Science*, pages 211–223. Springer Berlin Heidelberg, 2005.

[3] G. S. Tseitin. *On the Complexity of Derivations in Propositional Calculus.* 1968.

[4] A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability.* IOS Press, 2009.

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

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

[7] V. Khrapchenko. Depth and delay in a network (in Russian). *Soviet Mathematics – Doklady*, 19:1006–1009, 1978.

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

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

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

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

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

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

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

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

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

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

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

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

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

[21] Y. Kukimoto and R. K. Brayton. Exact required time analysis via false path detection. In *Design Automation Conference*, pages 220–225, 1997.

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

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

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

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

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

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

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

[29] T. Takata and Y. Matsunaga. An efficient cut enumeration for depth-optimum technology mapping for lut-based fpgas. In *ACM Great Lakes symposium on VLSI*, pages 351–356, 2009.

[30] A. Mishchenko, S. Chatterjee, and R. K. Brayton. Improvements to technology mapping for lut-based fpgas. In *IEEE Transactions on Computer Aided Design*, pages 41–49, 2007.

[31] Benchmarks. Benchmarks from the 2005 international workshop on logic synthesis available at http://iwls.org/iwls2005/benchmarks.html, 2005.

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

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

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

[35] M. K. Ganai, P. Ashar, A. Gupta, L. Zhang, and S. Malik. Combining strengths of circuit-based and CNF-based algorithms for a high-performance SAT solver. In *Design Automation Conference*, pages 747–750, 2002.

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

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

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

[39] Benchmarks from the 2005 International Workshop on Logic Synthesis available at http://iwls.org/iwls2005/benchmarks.html.

[40] P. Pudlak. Lower bounds for resolution and cutting plane proofs and monotone computations. *Journal of Symbolic Logic*, 62(3):981–998, 1997.

[41] M. L. Case, A. Mishchenko, R. K. Brayton, J. Baumgartner, and H. Mony. Invariant-strengthened elimination of dependent state elements. In *Formal Methods in Computer-Aided Design*, pages 9–17, 2008.

[42] R. Gershman, M. Koifman, and O. Strichman. An approach for extracting a small unsatisfiable core. *Formal Methods in System Design*, 33(1-3):1–27, 2008.

[43] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[44] G. Audemard and L. Simon. Predicting learnt clauses quality in modern sat solvers. In *Proceedings of the 21st international jont conference on Artifical intelligence*, IJCAI'09, pages 399–404, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[45] L. Zhang and S. Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Design, Automation and Test in Europe*, pages 10880–10885, 2003.

[46] A. Mishchenko, S. Chatterjee, and R. Brayton. DAG-aware AIG rewriting: A fresh look at combinational logic synthesis. In *Design Automation Conference*, pages 532–536, 2006.

[47] Q. Zhu, N. Kitchen, A. Kuehlmann, and A. S. Vincentelli. SAT sweeping with local observability dont-cares. In *Design Automation Conference*, pages 229–234, 2006.

[48] J. R. Jiang and R. K. Brayton. Functional dependency for verification reduction. In *Proceddings of CAV*, pages 268–280, 2004.

[49] O. Bar-Ilan, O. Fuhrmann, S. Hoory, O. Shacham, and O. Strichman. Linear-time reductions of resolution proofs. In *Haifa Verification Conference on Hardware and Software: Verification and Testing*, pages 114–128, 2009.

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

[51] S. A. Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16(1963):83–94, 1963.

[52] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[53] E. A. Emerson and J. Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, STOC '82, pages 169–180, New York, NY, USA, 1982. ACM.

[54] E. A. Emerson and J. Y. Halpern. "sometimes" and "not never" revisited: on branching versus linear time temporal logic. *J. ACM*, 33(1):151–178, January 1986.

[55] A. Biere, C. Artho, and V. Schuppan. Liveness checking as safety checking. In *In FMICS02: Formal Methods for Industrial Critical Systems, volume 66(2) of ENTCS*. Elsevier, 2002.

[56] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. pages 49–58. North-Holland, 1991.

[57] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[58] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[59] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV : a new symbolic model checker. *Software Tools for Technology Transfer*, 1998.

[60] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using sat procedures instead of bdds. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, DAC '99, pages 317–320, New York, NY, USA, 1999. ACM.

[61] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a sat-solver. In *Proceedings of the Third International Conference on Formal*

Methods in Computer-Aided Design, FMCAD '00, pages 108–125, London, UK, UK, 2000. Springer-Verlag.

[62] P. A. Abdulla, P. Bjesse, and N. Eén. Symbolic reachability analysis based on sat-solvers. In *Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '00, pages 411–425, London, UK, UK, 2000. Springer-Verlag.

[63] G. Stålmarck. Stålmarck's method and qbf solving. In *Proceedings of the International Conference on Computer-Aided Verification*, 1999.

[64] A. R. Bradley. SAT-based model checking without unrolling. In *Proceedings of the 12th international conference on Verification, model checking, and abstract interpretation*, VMCAI'11, pages 70–87, Berlin, Heidelberg, 2011. Springer-Verlag.

[65] N. Eén, A. Mishchenko, and R. Brayton. Efficient implementation of property directed reachability. In *Formal Methods in Computer-Aided Design (FMCAD), 2011*, pages 125 –134, 2011.

[66] A. Biere and K. Heljanko. Hardware Model Checking Competition (HWMCC) 2011 Benchmarks. Available at: http://fmv.jku.at/hwmcc11/, 2011.

[67] H. Chockler, A. Ivrii, A. Matsliah, S. Moran, and Z. Nevo. Incremental formal verification of hardware. In *Formal Methods in Computer-Aided Design (FMCAD), 2011*, pages 135 –143, 2011.

[68] N. Eén and A. Biere. Effective preprocessing in sat through variable and clause elimination. In *In Proceedings of SAT05*, pages 61–75. Springer, 2005.

[69] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition.* MIT Press, 2001.

[70] J. P. M. Silva and K. A. Sakallah. GRASP: a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided*

design, ICCAD '96, pages 220–227, Washington, DC, USA, 1996. IEEE Computer Society.

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

[72] M. Heule, O. Kullmann, S. Wieringa, and A. Biere. Cube and conquer: Guiding CDCL sat solvers by lookaheads. In Kerstin Eder, Joo Loureno, and Onn Shehory, editors, *Hardware and Software: Verification and Testing*, volume 7261 of *Lecture Notes in Computer Science*, pages 50–65. Springer Berlin Heidelberg, 2012.

[73] N. Dershowitz, Z. Hanna, and E. Nadel. A clause-based heuristic for SAT solvers. In *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 46–60. Springer-Verlag, 2005.

[74] K. Hoder and N. Bjϕrner. Generalized property directed reachability. In A. Cimatti and R. Sebastiani, editors, *Theory and Applications of Satisfiability Testing SAT 2012*, volume 7317 of *Lecture Notes in Computer Science*, pages 157–171. Springer Berlin Heidelberg, 2012.

[75] A. Cimatti and A. Griggio. Software model checking via IC3. In *Proceedings of the 24th international conference on Computer Aided Verification*, CAV'12, pages 277–293, Berlin, Heidelberg, 2012. Springer-Verlag.

[76] L. Alfaro, M. Kwiatkowska, G. Norman, D. Parker, and R. Segala. Symbolic model checking for probabilistic processes using mtbdds and the kronecker representation. In *Tools and Algorithms for the Analysis and Construction of Systems, LNCS 1785*, pages 395–410. Springer-Verlag, 2000.

[77] C. Baier, E. M. Clarke, V. Hartonas-Garmhausen, M. Z. Kwiatkowska, and M. Ryan. Symbolic model checking for probabilistic processes. In *ICALP 1997*, pages 430–440, 1997.