Reduction of Interpolants for Logic Synthesis

John D. Backes and Marc D. Riedel

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

***Abstract*—Craig Interpolation is a state-of-the-art technique for logic synthesis and verification, based on Boolean Satisfiability (SAT). Leveraging the efficacy of SAT algorithms, Craig Interpolation produces solutions quickly to challenging problems such as synthesizing functional dependencies and performing bounded model-checking. Unfortunately, the quality of the solutions is often poor. When interpolants are used to synthesize functional dependencies, the resulting structure of the functions may be unnecessarily complex. In most applications to date, interpolants have been generated directly from the proofs of unsatisfiability that are provided by SAT solvers. In this work, we propose efficient methods based on incremental SAT solving for modifying resolution proofs in order to obtain more compact interpolants. This, in turn, reduces the cost of the logic that is generated for functional dependencies.**

I. Introduction

Craig's Interpolation theorem states that given two formulas A and B with the relationship $A \rightarrow B$, there exists some formula I such that $A \rightarrow I$ and $I \rightarrow B$ (where I contains only symbols that are present in both A and B). Craig provided a method for deriving I based on a proof of the implication of B from A [4].

Boolean Satisfiability (SAT) has found wide applicability for problems in logic synthesis and verification [1], [8], [10]. A SAT instance is a Boolean formula specified in conjunctive normal form (CNF). Note that if an instance of SAT is unsatisfiable and it is divided into clause subsets A and B, these will have the property that $A \rightarrow \overline{B}$, this for any choice of A and B. Recent work has shown how Craig's theorem can be used to derive interpolants from unsatisfiable SAT instances [10], [14]. These algorithms rely on a resolution-based proof to show that a set of clauses forms a contradiction. Modern SAT solvers have been adapted to produce such resolution proofs [17].

Craig Interpolation has been proposed for synthesizing functional dependencies in combinational logic [2], [9]. 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.

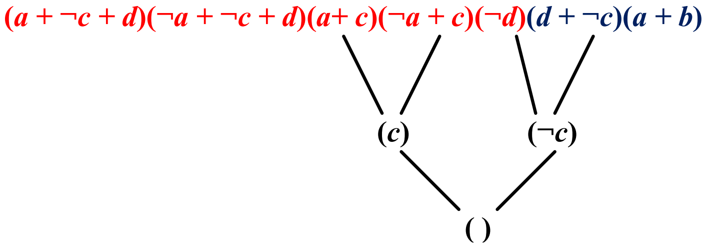
$(a+\neg c+d)(\neg a+\neg c+d)(a+c)(\neg a+c)(\neg d)(d+\neg c)(a+b)$
(c)
$(\neg c)$
$(\,)$

Fig. 1. A resolution proof from an unsatisfiable CNF formula. Clauses of A are shown in red while clauses of B are shown in blue

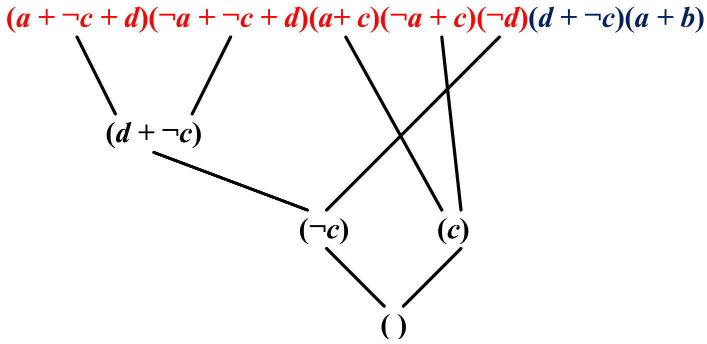
$(a+\neg c+d)(\neg a+\neg c+d)(a+c)(\neg a+c)(\neg d)(d+\neg c)(a+b)$
$(d+\neg c)$
$(\neg c)$
(c)
$(\,)$

Fig. 2. A different resolution proof from the same unsatisfiable CNF formula. Clauses of A are shown in red while clauses of B are shown in blue

While generating functional dependencies in this manner is quick and scalable, it generally does not yield optimal (or even very good) results. The methods to derive interpolants from unsatisfiable SAT instances proposed in [14] and [10] follow fixed trajectories based on the resolution proof generated by a SAT solver. However, I is an over-approximation of the variable assignments that cause A to be true; there may be many different valid implementations for 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 1 and 2. Both proofs start with the same original root clauses and the same clause partitions A and 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

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

function. Using the interpolant generation algorithm proposed in [10], the interpolants for the clause partition in Figures 1 and 2 result in implementations with 4 gates and 8 gates. Even with small problem sizes, poorly structured resolution proofs can result in overly complex interpolants.

In [3], 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 work, we explore methods for modifying resolution proofs in order to obtain more compact interpolants. This, in turn, reduces the cost of the logic that is generated for functional dependencies. We use the concept of Minimum Unsatisfiable Cores (MUCs) [7]. 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.

II. Background and Definitions

A Boolean formula maps an assignment of Boolean variables to a Boolean value ($\mathbf{B^m} \rightarrow \mathbf{B}$). We use the convention that addition denotes an OR operation, multiplication denotes the AND operation, a "→" denotes implication, and an overbar or a "¬" denotes negation. An occurrence of a Boolean variable in a Boolean formula, either negated or non-negated, is referred to as a *literal*. A disjunction of literals is referred to as a *clause*. A Boolean formula is in conjunctive normal form (CNF) if it is represented by a conjunction of clauses. Boolean Satisfiability (SAT) is the problem of determining whether or not a CNF formula can evaluate to true for some assignment of its variables. If there is some variable assignment that causes the formula to evaluate to true, then the formula is said to be *satisfiable*. If there is no variable assignment that causes the formula to be true, then the formula is said to be *unsatisfiable*. We will sometimes use the term *SAT instance* when referring to a CNF formula whose satisfiability we are trying to solve.

Given the conjunction of two clauses that share a literal that is negated in one but not the other, a third clause that is a disjunction of their remaining literals is implied. This identity is known as Boolean *resolution*. The common literal that is negated in one clause but not the other is called the *pivot variable* and the resulting clause is called as the *resolvent*. Consider the identity

$$(z_0 + x_1 + \cdots + x_n)(\bar{z_0} + y_1 + \cdots + y_m) \rightarrow \\ (x_1 + \cdots + x_n + y_1 + \cdots + y_m)$$

Here the pivot variable is z_0. 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. This structure is called a *resolution proof*. 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 were 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 were few resolutions steps taken from the roots 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.

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 4 was presented in [10]. It describes how to generate a circuit that implements an interpolant from a resolution proof and a clause partition. This algorithm was adapted from a procedure presented in [14] to find the Boolean value for an interpolant given a variable assignment.

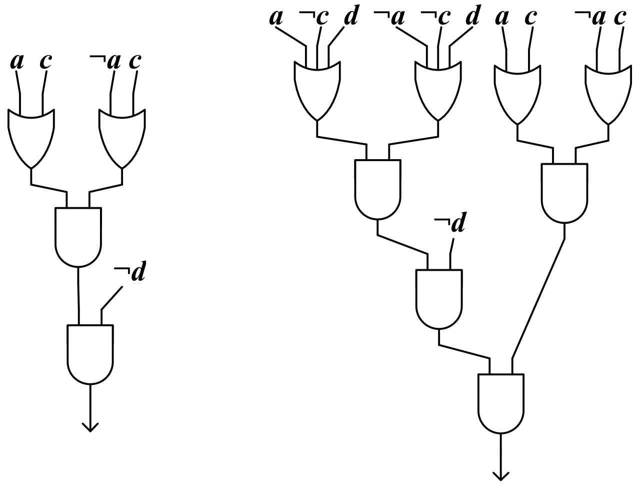

Fig. 3. Two interpolants produced by calling p(c) on the empty clause of a resolution proof. The circuits on the left and right are generated from the proofs in Figures 1 and 2, respectively

Let g(c) take a clause c and return clause c with only its global literals present. Let c_1 and c_2 refer to c's parent clauses. Procedure p(c) is defined in Figure 4. 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 are referring to the number of gates that are needed to represent it. It should be clear that the size of the

```
p($c$):
    if $c$ is a root 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($c_1$) + p($c_2$)
        else
            return (p($c_1$))(p($c_2$))
```

Fig. 4. The algorithm proposed in [10] to produce a circuit that implements an interpolant of a given clause partition, via a proof of unsatisfiability.

interpolant is bounded by the number of nodes in the resolution proof. Figure 3 shows the results of applying this procedure on the resolution proofs in Figures 1 and 2.

III. 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 1

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 root nodes 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* ***A*** *and* ***B****.*

Proof:

Consider two root nodes n_1 and n_2 in ***A*** (***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 root nodes of ***A*** (***B***) without reducing or expanding the space of Boolean assignments that satisfy ***A*** (***B***). Since the space of Boolean assignments representing ***A*** and ***B*** does not change – only the clause representation changes – an interpolant generated by the procedure in Figure 4 while considering n_3 to be a root clause will still be a valid interpolant of the original clause partition.

■

We can mark nodes resolved from only nodes of ***A*** or ***B*** as root nodes of ***A*** or ***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 root nodes). However, for most applications this optimization by itself doesn't yield a significant improvement in interpolant size[1].

Example 1

Let us see how this optimization affects the interpolants of the proofs in Figures 1 and 2. In Figure 1, we notice that resolvent clause (c) was resolved from two root clauses of ***A*** *(clauses $(a + c)$ and $(\bar{a} + c)$). This means that we can consider clause (c) to be a root of* ***A****. Calling p(c) on this proof with clause (c) marked as a root node will yield the interpolant $(c)(\bar{d})$.*

In the proof shown in Figure 2, we can see that the empty clause is derived from implications that only occur from partition ***A****. This means that the empty clause can be considered as a root of* ***A****. Calling p(c) on this proof with the empty clause marked as a root of* ***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 1 to 1 gate and the number of gates to implement the proof in Figure 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 ***A*** and a clause resolved from ***B*** will tend to have smaller interpolants. This is because more of the internal nodes can be considered as root nodes 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 ***A*** and a clause of ***B*** (e.g., the proof in Figure 2 is more disjoint then the proof in Figure 1).

In [7], 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 root nodes 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 4.

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 root nodes), we eventually arrive at a set of root nodes that were used to derive c. Let ***R*** be this set of root nodes.

Proposition 2

For all variable assignments that satisfy the set of clauses ***R****, c must also be satisfied.*

Proof:

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

[1] Often, CNF formulas are generated from a Tstein Decomposition of a logic circuit [16]. 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 [9], [10]). 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 root 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 [11] – this optimization will likely yield little benefit to the interpolant size.

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 [7], the authors use this observation to determine if a clause c could be derived through resolution from a set of clauses $\boldsymbol{R}$. They propose a SAT-Based algorithm in which intermediate nodes are iteratively checked to see if they can be implied by a smaller set of root nodes. The goal of the algorithm is to find a smaller set of root 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 roots 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 root of $\boldsymbol{A}$. If $(\boldsymbol{B})(\bar{c})$ is unsatisfiable then we know that c can be considered to be a root 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 root nodes of $\boldsymbol{A}$ or $\boldsymbol{B}$.

Example 2
Consider the proofs in Figures 1 and 2 again. The proof in Figure 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 2. This tells us that we can consider clause* $(\bar{c})$ *to be a root of partition* $\boldsymbol{A}$. *As shown earlier, marking* $(\bar{c})$ *as a root 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 root nodes of $\boldsymbol{A}$ or $\boldsymbol{B}$ respectively. Mark every other node as unvisited.
2) Select a clause c that is not a root 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 root of $\boldsymbol{A}$. Otherwise, If $(\boldsymbol{B})(\bar{c})$ is unsatisfiable, mark c as a root of $\boldsymbol{B}$. Mark c as visited.
4) If c is now marked as a root of either $\boldsymbol{A}$ or $\boldsymbol{B}$, check to see if its children can trivially be marked as root nodes 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 roots 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.

A. 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 root nodes 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 root, 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 root 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 root node, then we might not need to check some of the node's ancestors.

Proposition 3
Consider some mixed node n *which we prove to be a root 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 this node will not need to be checked to be a root node.*

Proof:
The procedure in Figure 4 terminates on root nodes. If n is marked as a root 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.

■

Example 3
To better illustrate this point, consider the resolution proof shown in Figure 5. 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 root 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 roots of* $\boldsymbol{A}$ *(or* $\boldsymbol{B}$*) 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 root node. Showing that a mixed node can be considered root may allow us to check fewer of the node's ancestors, but it does not imply that its ancestors can be considered root[2].

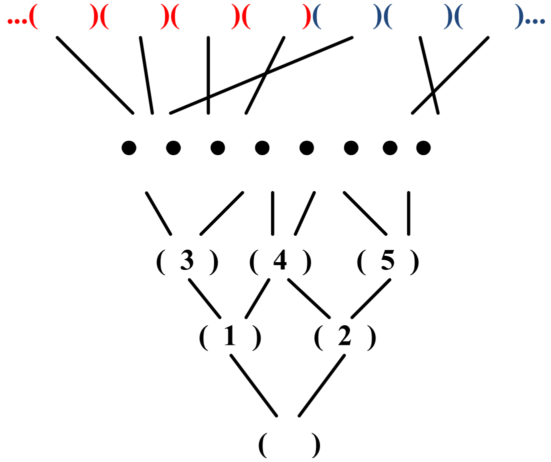

Fig. 5. A resolution proof from an unsatisfiable CNF formula. Clauses of $\boldsymbol{A}$ are shown in red while clauses of $\boldsymbol{B}$ are shown in blue. Nodes 1, 2, 3, 4, and 5 are nodes somewhere in the proof

Proposition 4
If we prove that two parents of a mixed node can be considered roots of the same clause partition, then this implies their child clause can be considered as a root of this partition.

Proof:
This is basically the same condition as Proposition 1. If nodes n_1 and n_2 are marked as roots 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 $\boldsymbol{A}$ ($\boldsymbol{B}$).

■

Example 4
Once again consider the proof shown in Figure 5. If we show that nodes 3 and 4 can be considered to be roots 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 root of $\boldsymbol{A}$ ($\boldsymbol{B}$) because it can trivially be considered a root node of one partition since both of its parents are roots 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 root nodes.

By initially checking nodes that are close to the roots 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 roots. Also, these nodes are more likely to be converted to root nodes because they are in a sense "closer" to the set of original root nodes.

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 root nodes because they are in a sense "further" from the set of original root nodes.

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 [17]. 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.

B. *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 [6]. 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 root 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 root 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 5
Consider again the proof shown in Figure 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:

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

When we solve this SAT instance we assume $a_{\text{off}} = 0$ and $b_{\text{off}} = 1$. 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 [6].

[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 $\boldsymbol{A}$ and $\boldsymbol{B}$. Unless the root nodes 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.

IV. 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 [9]. 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 root. 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 [12] using MiniSAT for SAT solving [15].

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 root of ***A*** or ***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 root nodes 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. 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 V.

V. DISCUSSION

In this work, our goal was to minimize the size of the DAG that is used to represent an interpolant generated from a resolution proof. With a smaller representation of the interpolant, we obtain more compact functional dependencies.

Interpolation has also been shown to be a useful tool for bounded model checking [10]. For this application, the less that the interpolant over-approximates the on-set of a transition relation the better. Note that our methods for minimizing interpolant size do not directly apply to the problem of producing smaller on-sets. One goal for future work would be to extend our methodology to this domain. Perhaps we can bias the resolutions to involve more variables that are not local to the clause partitions. If so, we could generate an interpolant with more internal AND operations. This should decrease the size of the on-set of the interpolant.

Other optimizations that we will investigate 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. A lot of research in SAT solving pertains to heuristics for making variable decisions that will lead to solving SAT instances *faster* [15], [13], [5]. Perhaps some of this intuition could be applied to variable decision

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 I

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 II

RESULTS OF THE BACKWARD-SEARCH METHOD ON THE table3 BENCHMARK. EACH FUNCTION IS A PO EXPRESSED IN TERMS OF THE PIS AND OTHER POS.

heuristics that result in resolution proofs that produce smaller interpolants.

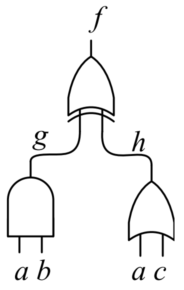

Fig. 6. A circuit with an observability don't care of g = 1, h = 0

If we use interpolants as a starting point to generate a structure to perform traditional synthesis, these traditional techniques can show better performance. When computing interpolants for the application of 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 6. 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 6 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

[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 was not considered in isolation from the rest of the circuit, then the 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

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 III
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 IV
A TABLE OF THE AVERAGED RESULTS USING THE BACKWARD-SEARCH METHOD AMONG DIFFERENT BENCHMARKS.

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 [9] 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 make vast changes to a network's structure.

REFERENCES

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

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

[3] 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.

[4] W. Craig. Linear Reasoning: A New Form of the Herbrand-Gentzen Theorem. *Symbolic Logic*, 22(3):250–268, 1957.

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

[6] 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.

[7] 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.

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

[9] 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.

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

[11] 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.

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

[13] 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.

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

[15] N. Sörensson et al. Minisat v1.13 – a SAT solver with conflict-clause minimization. Technical report.

[16] G. S. Tseitin. On the complexity of derivations in propositional calculus. pages 115–125, 1968.

[17] L. Zhang and S. Malik. Validating sat solvers using an independent resolution-based checker: Practical implementations and other applications. In *In Proc, DATE '03.*, pages 10880–10885. IEEE Computer Society, 2003.