To SAT or Not to SAT: Scalable Exploration of Functional Dependency

Jie-Hong Roland Jiang, *Member*, *IEEE*, Chih-Chun Lee,
Alan Mishchenko, *Member*, *IEEE*, and Chung-Yang (Ric) Huang, *Member*, *IEEE*

Abstract—Functional dependency is concerned with rewriting a Boolean function f as a function h over a set of base functions $\{g_1, \ldots, g_n\}$, i.e., $f = h(g_1, \ldots, g_n)$. It plays an important role in many aspects of electronic design automation (EDA). Prior approaches to the exploration of functional dependency are based on binary decision diagrams (BDDs), which may not be easily scalable to large designs. This paper formulates both single-output and multiple-output functional dependencies as satisfiability (SAT) solving and exploits extensively the capability of a modern SAT solver. Thereby, functional dependency can be detected effectively through incremental SAT solving, and the dependency function h, if it exists, is obtained through Craig interpolation. The proposed method enables 1) scalable detection of functional dependency, 2) fast enumeration of dependency function under a large set of candidate base functions, and 3) potential application to large-scale logic synthesis and formal verification. Experimental results show that the proposed method is far superior to prior work and scales well in dealing with the largest ISCAS and ITC benchmark circuits with up to 200K gates.

Index Terms—Automatic synthesis, design aids, logic design, optimization.

1 Introduction

Functional dependency [1], [2] appears commonly among a set $F = \{f_1, \ldots, f_n\}$ of Boolean functions in VLSI circuit design as a function $f_i \in F$, called the *target function*, can often be reexpressed by some function h, called the *dependency function*, over other functions $F \setminus \{f_i\}$, called the *base functions*. The exploration of functional dependency plays an important role in many aspects of EDA, ranging from logic synthesis to formal verification. For example, functional dependency is useful in the identification of redundant registers in RTL synthesis [3], [4], in resubstitution and simplification of Boolean functions in logic synthesis [5], in BDD minimization [6], in state space reduction in formal verification [1], [7], [8], in search space reduction of SAT solving [9], and in other areas. Advances on the exploration of functional dependency may benefit a wide range of applications.

Given a set of Boolean functions $\{f_1, \ldots, f_n\}$, we study if any f_i can be written as $h(f_1, \ldots, f_{i-1}, f_{i+1}, \ldots, f_n)$. Conventional approaches [1] to the exploration of functional dependency rely mostly on BDDs [10]. Unfortunately, the computation using BDDs suffers from the memory explosion problem and thus is not scalable to manipulate large designs. In contrast, SAT solving consumes little memory resources (linear in the input size) at the cost of computation time and thus is more robust at least in representing large designs. Recent advances have made modern SAT solvers, e.g., [11], [12], an efficient Boolean reasoning engine and a viable alternative to BDDs. More and more logic synthesis and verification algorithms shift their computation paradigms from BDDs to SAT, e.g., [13], [14]. However, formulating the computation of functional dependency as pure SAT solving is not straightforward due to the difficulty in deriving the dependency function h, whose derivation in BDD-based computation, in contrast, is immediate.

This paper demonstrates that the exploration of single-output as well as multiple-output functional dependencies (including the efficient derivation of dependency functions) can be achieved with pure SAT solving. In particular, a dependency function, if it exists, can be obtained with the interpolant constructed from a refutation proof of a SAT solver. Essentially, Craig's interpolation theorem [15] lays the foundation.

To detect functional dependency for different target functions and to obtain different dependency functions for a target function, incremental SAT solving is adopted to reuse learned clauses and to indicate responsive conflicting assumptions. Even though incremental SAT solving has been widely used, we explore its new use in our framework. In essence, it not only speeds up our computation, but also provides an automatic way of selecting sets of base functions of a target function. Experimental results show encouraging improvements over BDD-based approaches.

The main results of this paper include 1) a new SAT-based derivation of dependency function using Craig interpolation, which enables a pure SAT solution to the exploration of functional dependency, and 2) an incremental SAT-based enumeration of target and base functions, which effectively reduces the search space for solving similar SAT instances. Practical experience suggests that a pure SAT formulation of functional dependency avoids the BDD memory explosion problem and is scalable to large

- *J.-H.R. Jiang, C.-C. Lee, and C.-Y. (Ric) Huang are with the Department of Electrical Engineering and Graduate Institute of Electronics Engineering, National Taiwan University, Taipei 10617, Taiwan. E-mail: {jhjiang, ric}@cc.ee.ntu.edu.tw, d98943033@ntu.edu.tw.*
- *A. Mishchenko is with the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, Room 545, Cory Hall, CA 94720. E-mail: alanmi@eecs.berkeley.edu.*

Manuscript received 1 Feb. 2009; revised 23 June 2009; accepted 16 July 2009; published online 4 Jan. 2010.
Recommended for acceptance by I. Markov.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2009-02-0057.
Digital Object Identifier no. 10.1109/TC.2010.12.

designs. The proposed method is powerful in detecting complex functional dependency even among a large set of base functions.

This paper is organized as follows: After preliminaries are introduced in Section 2, our SAT-based formulation of single-output functional dependency is detailed in Section 3. Generalization to multiple-output functional dependency is discussed in Section 4. The proposed approach is evaluated with experimental results in Section 5. Section 6 compares some related work; Section 7 concludes this paper and outlines future work.

2 Preliminaries

As a notational convention, in the sequel symbols "$\wedge$", "$\vee$", and "$\neg$" denote Boolean AND, OR, and COMPLEMENT operations, respectively. The cardinality (or size) of a set S is denoted as |S|.

The problem definition of functional dependency and background on SAT solving are given as follows:

2.1 Functional Dependency

Functional dependency is defined as follows:

Definition 1. *Given a Boolean function $f : \boldsymbol{B}^m \rightarrow \boldsymbol{B}$ and a vector of Boolean functions $G = (g_1(X), \ldots, g_n(X))$ with g_i: $\boldsymbol{B}^m \rightarrow \boldsymbol{B}$ for $i = 1, \ldots, n$, over the same set of variable vector $X = (x_1, \ldots, x_m)$, we say that f **functionally depends** on G if there exists a Boolean function h: $\boldsymbol{B}^n \rightarrow \boldsymbol{B}$ such that $f(X) = h(g_1(X), \ldots, g_n(X))$. We call functions f, G, and h the **target function**, **base functions**, and **dependency function**, respectively.*

Note that functions f and G are over the same domain in the definition. Moreover, h needs not depend on all of the functions in G.

The necessary and sufficient condition of the existence of the dependency function h is given as follows:

Proposition 1. [1] *Given a target function f with its care condition f_c (the characteristic function of the case set of f) and given the base functions G, let $h^0 = \{a \in \boldsymbol{B}^n : a = G(b), f(b) = 0$, and $f_c(b) = 1$, for $b \in \boldsymbol{B}^m\}$ and $h^1 = \{a \in \boldsymbol{B}^n : a = G(b), f(b) = 1$, and $f_c(b) = 1$, for $b \in \boldsymbol{B}^m\}$. Then, h is a feasible dependency function if and only if $\{h^0 \cap h^1\}$ is empty. In this case, h^0, h^1, and $\boldsymbol{B}^n \setminus \{h^0 \cup h^1\}$ are the offset, onset, and don't care set of h, respectively.*

By Proposition 1, one can not only determine the existence of a dependency function, but also deduces a feasible one.

To explore functional dependency for a given circuit netlist, there are many choices of f and G. One may use the support-variable information defined below to effectively select G for a specific f.

Definition 2. *For a Boolean function f with input variables $X = (x_1, \ldots, x_m)$, variable x_i is a* **support variable** *of f if*

$$f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_m) \neq f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_m).$$

With a slight generalization, we define the support variables of a functional vector $G = (g_1, g_2, \ldots, g_n)$ to be the union of those of g_i for $i = 1, \ldots, n$.

Proposition 2 [1]. *There exists a feasible dependency function of f with respect to G only if every support variable of f is also a support variable of G.*

Proposition 2 can be used for fast screening in selecting the base functions G.

2.1.1 BDD-Based Exploration of Functional Dependency

Conventional BDD-based exploration of functional dependency is reviewed below in order to contrast with the novel SAT-based approach.

Proposition 1 suggests a way of determining the existence of a dependency function and its derivation. Essentially standard image computation applies. Let y_i be the output variable of g_i. Then, the onset, offset, and don't care set of h can be derived by

$$h^0(Y) = \exists X[R(X,Y) \wedge \neg f(X) \wedge f_c(X)],$$
$$h^1(Y) = \exists X[R(X,Y) \wedge f(X) \wedge f_c(X)], \text{ and}$$
$$h^{\mathrm{dc}}(Y) = \neg(h^0 \vee h^1),$$

respectively, where relation

$$R(X,Y) = (y_1 \equiv g_1(X)) \ \wedge \ (y_2 \equiv g_2(X)) \wedge \cdots \wedge (y_n \equiv g_n(X)).$$

The dependency function $h(Y)$ exists if and only if $h^0(Y) \wedge h^1(Y)$ has no satisfying assignments, i.e., $h^0(Y) \wedge h^1(Y)$ equals constant 0.

Note that all of the above operations can be done using BDDs [1], [5]. Constructing the relation $R(X, Y)$ along with the image computation may suffer from memory explosion, especially when $|G|$ is large, even though the final BDDs of h^0 and h^1 can be small. Therefore, it is necessary to restrict the size of the set of base functions at the cost of losing completeness. Keeping $|G|$ small may often result in a failure to compute some dependency that truly holds in a circuit. Once the search for a feasible dependency function with respect to a set of base functions fails, another set of base functions is selected and the computation of functional dependency repeats. Consequently, although some fast filtering techniques, e.g., by Proposition 2, are available [1], BDD-based computation is inefficient in that there may be too many selections of G tested before some functional dependency is discovered. As will be seen later, this deficiency can be overcome in SAT-based exploration of functional dependency.

2.2 Propositional Satisfiability

Let $V = \{v_1, \ldots, v_k\}$ be a finite set of Boolean variables. A **literal** l is either a Boolean variable v_i or its negated form $\neg v_i$. A **clause** c is a disjunction of literals. Without loss of generality, we shall assume there are no repeated or complementary literals in the same clause. A **SAT instance** is a conjunction of clauses, i.e., in the so-called **conjunctive normal form** (**CNF**). In the sequel, a clause set $S = \{c_1, \ldots, c_k\}$ shall mean to be the CNF formula $(c_1 \wedge \cdots \wedge c_k)$. An **assignment** over V gives every variable v_i a Boolean value either true or false. A SAT instance is **satisfiable** if there exists a satisfying assignment such that the CNF formula evaluates to true. Otherwise, it is **unsatisfiable**. Given a SAT instance,

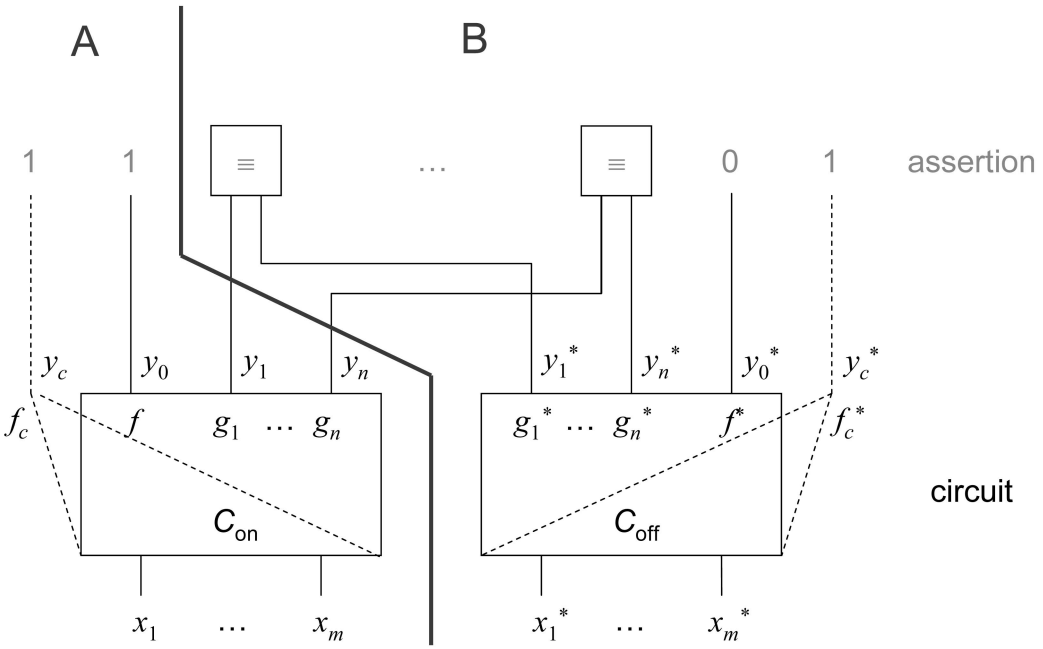

Fig. 1. Dependency logic network for checking single-output functional dependency.

the **satisfiability (SAT) problem** asks whether it is satisfiable or not. A SAT solver is designated to solve the SAT problem.

2.2.1 Refutation Proof and Craig's Interpolation

Definition 3. *Assume literal v is in clause c_1 and $\neg v$ in c_2. A **resolution** of clauses c_1 and c_2 on variable v yields a new clause c containing all literals in c_1 and c_2 except for v and $\neg v$. The clause c is called the **resolvent** of c_1 and c_2, and variable v the **pivot variable**.*

Proposition 3. *A resolvent c of c_1 and c_2 is a logical consequence of $c_1 \wedge c_2$, that is, $c_1 \wedge c_2$ implies c.*

Theorem 1 [16]. *For an unsatisfiable SAT instance, there exists a sequence of resolution steps leading to an empty clause.*

Theorem 1 can be easily proved by Proposition 3 since an unsatisfiable SAT instance must imply a contradiction. Often only a subset of the clauses, called an **unsatisfiable core**, of the SAT instance participates in the resolution steps leading to an empty clause.

Definition 4. *A **refutation proof** Π of an unsatisfiable SAT instance S is a directed acyclic graph (DAG) $\Gamma = (N, A)$, where every node in N represents a clause which is either a root clause in S or a resolvent clause having exactly two predecessor nodes, and every arc in A connects a node to its ancestor node. The unique leaf of Π corresponds to the empty clause.*

Modern SAT solvers, such as Chaff [11] and MiniSat [12], are capable of producing a refutation proof from an unsatisfiable SAT instance.

Theorem 2 (Craig Interpolation Theorem) [15]. *Given two inconsistent propositional formulas A and B (i.e., their conjunction $A \wedge B$ is unsatisfiable), then there exists a Boolean formula $A^{\#}$ referring only to the common variables of A and B such that $A \Rightarrow A^{\#}$ and $A^{\#} \Rightarrow \neg B$.*

The Boolean formula $A^{\#}$ is referred to as the **interpolant** of A and B. Detailed exposition on how to construct an interpolant from a refutation proof in linear time can be found in [17], [18], [19]. Note that the so-derived interpolant is in a circuit structure, which can then be converted into the CNF as discussed below.

2.2.2 Circuit-to-CNF Conversion

Given a circuit netlist, it can be converted to a CNF formula in such a way that the satisfiability is preserved. The conversion is achievable in linear time by introducing some intermediate variables [20], [21].

3 Single-Output Functional Dependency

In this section, we first consider **single-output functional dependency**, i.e., functional dependency with a single target function.

3.1 Main Construct

To formulate the exploration of functional dependency as SAT solving, we introduce the **dependency logic network** (DLN) shown in Fig. 1. For a given circuit netlist C consisting of $n+1$ Boolean functions $\{f_0, \ldots, f_n\}$, suppose function f_0 and the others $f_1, \ldots, f_n$ are identified to be the target function and base functions, respectively. (That is, in the notation of Section 2, f_0 corresponds to f, and f_i corresponds to g_i for $i = 1, \ldots, n$.) The circuit netlist is instantiated into two copies, identified as C_{on} and C_{off}, in Fig. 1, where the two dotted cones indicate the care condition f_c, if available, of the target function. For every variable v (respectively, function f) in C_{on}, there is a corresponding starred version v^* (respectively, f^*) in C_{off}. Let y_i and y_i^* be the output variables of f_i and f_i^*, respectively. The circuits C_{on} and C_{off} can be converted into CNF formulas S_{on} and S_{off}, respectively, in linear time. In addition, the output of the target function f in C_{on} is asserted to true, i.e., $y_0 \equiv 1$; that of f^* in C_{off} is asserted to false, i.e., $y_0^* \equiv 0$. Furthermore, equality constraints ($y_i \equiv y_i^*$)

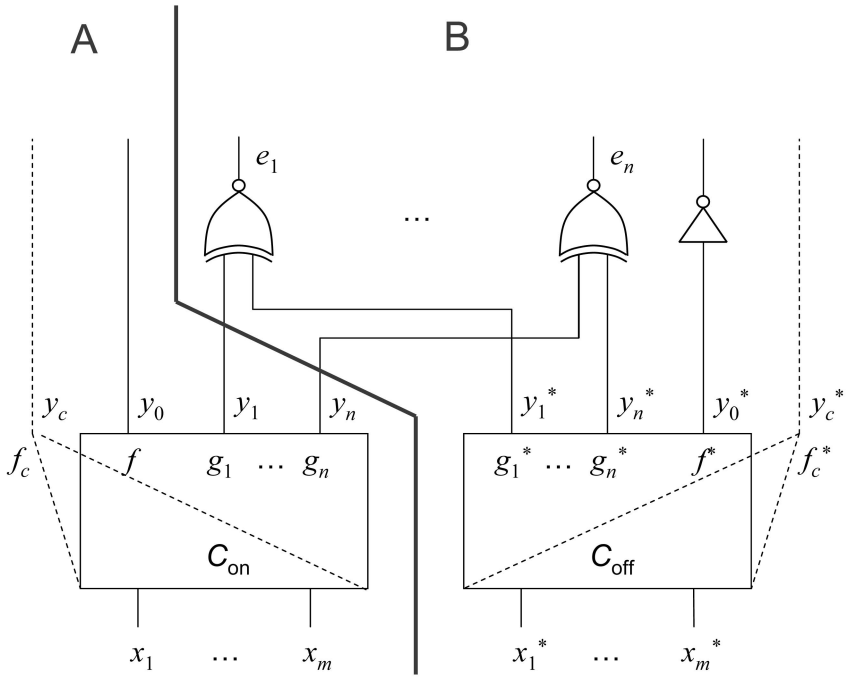

Fig. 2. Alternate dependency logic network for checking single-output functional dependency.

are imposed for $i = 1, \ldots, n$. The overall CNF formula S_{DLN} for the DLN shown in Fig. 1 is

$$S_{\mathrm{on}} \wedge S_{\mathrm{off}} \wedge y_0 \wedge y_c \wedge \neg y_0^* \wedge y_c^* \wedge (y_1 \equiv y_1^*) \wedge \cdots \wedge (y_n \equiv y_n^*), \quad (1)$$

where $(y_i \equiv y_i^*)$ is the shorthand for $(y_i \vee \neg y_i^*) \wedge (\neg y_i \vee y_i^*)$.

With a minor implementation difference, the DLN of Fig. 1 can be equivalently represented with the logic network of Fig. 2, where we assert all the primary outputs to be constant one. This representation will be more convenient later for our discussion on multiple-output functional dependency.

The intuition behind this construct is that formula $S_{\mathrm{on}} \wedge y_0 \wedge y_c$ (respectively, $S_{\mathrm{off}} \wedge \neg y_0^* \wedge y_c^*$) imposes the constraint that the valuations over input variables $(x_1, \ldots, x_m)$ (respectively, $(x_1^*, \ldots, x_m^*)$) must be the care onset of f (respectively, care offset of f^*). By Proposition 1, we can, thus, test if h^0 and h^1 are disjoint.

Theorem 3. *Given a target function f with care condition f_c and given a set of base functions g_i for $i = 1, \ldots, n$, a dependency function h exists if and only if the CNF formula S_{DLN} of the corresponding DLN is unsatisfiable.*

Proof. ($\Rightarrow$) By Definition 1, f can be expressed as $h(g_1(X), \ldots, g_n(X))$. Proposition 1 asserts that the onset h^1 and offset h^0 of h must be disjoint. Observe that h^0 and h^1 are essentially the sets of satisfying assignments of variables y_i of C_{on} and y_i^* of C_{off}, respectively, for $i = 1, \ldots, n$. Hence, CNF formula S_{DLN}, which is the conjunction of S_{on} and S_{off}, is unsatisfiable.

($\Leftarrow$) If S_{DLN} is unsatisfiable, then there are two possibilities. In the first case, S_{on} or S_{off} is unsatisfiable. It happens only when f is a constant function. In particular, S_{on} (respectively, S_{off}) is unsatisfiable if and only if f under the care condition f_c is constant zero (respectively, constant one). In this case, the target function f can be trivially expressed by an arbitrary set of base functions, and thus h exists. (In the sequel, we shall assume that a target function is nonconstant.) In the second case, both formulas S_{on} and S_{off} are satisfiable. Because formulas S_{on} and S_{off} have disjoint variables, the unsatisfiability of S_{DLN} must be due to the equality constraints $(y_i \equiv y_i^*)$ of S_{DLN}. That is, the sets of images of the care onset and care offset of f under the base functions are disjoint. By Proposition 1, we know that h must exist. □

We show how the dependency function h can be derived using interpolation provided that the clause set S_{DLN} is unsatisfiable. To apply Theorem 2, we partition the clause set S_{DLN} into two subsets A and B as indicated in Fig. 1 (also in Fig. 2). Thereby, we have the following claim.

Corollary 1. *For unsatisfiable $S_{\mathrm{DLN}} = A \wedge B$ with*

$$A = S_{\mathrm{on}} \wedge y_0 \wedge y_c \text{ and}$$
$$B = S_{\mathrm{off}} \wedge \neg y_0^* \wedge y_c^* \wedge (y_1 \equiv y_1^*) \wedge \cdots \wedge (y_n \equiv y_n^*),$$

the resultant interpolant $A^{\#}$ derived from a refutation proof yields a desired dependency function h.

Proof. Observe that the common variables of A and B are $Y = (y_1, \ldots, y_n)$, which are the desirable inputs of the dependency function. Since $A \wedge B$ are unsatisfiable, by Theorem 2 there exists an interpolant $A^{\#}$ which refers only to Y. In addition, the conditions $A \Rightarrow A^{\#}$ and $A^{\#} \Rightarrow \neg B$ suggest that the set of valuations over variables Y satisfying $A^{\#}$ must be an overapproximation of $h^1(Y)$ and must be disjoint from $h^0(Y)$. Hence, $A^{\#}(Y)$ is a valid implementation of the dependency function $h(Y)$ with respect to the underlying target and base functions. □

```
FunctionalDependencyBySAT
input: target function f and base functions {g1, ..., gn}
output: a dependency function h
begin
01  Construct clause set S_DLN
02  if (S_DLN is UNSAT)
03    Partition S_DLN into clause sets A and B
04    Derive an interpolant A# from refutation proof
05    return A#
06  return no solution
end
```

Fig. 3. Algorithm of SAT-based computation of functional dependency.

Therefore, as long as a SAT solver can produce a refutation proof, it can be exploited to generate the corresponding interpolant and, thus, the dependency function. The overall algorithm of the exploration of functional dependency is sketched in Fig. 3.

Note that the choice of clause sets A and B is not unique. For example, letting

$$A = S_{\text{on}} \wedge y_0 \wedge y_c \wedge (y_1 \equiv y_1^*) \wedge \cdots \wedge (y_n \equiv y_n^*) \text{ and}$$
$$B = S_{\text{off}} \wedge \neg y_0^* \wedge y_c^*,$$

is valid as well. In fact, different refutation proofs and different choices of A and B can be exploited to obtain better implementation for the dependency function.

3.2 Incremental SAT Solving

The previous discussion assumes that the target function and base functions are given. However, for a given circuit netlist, there may be a wide choice of target and base functions to explore functional dependency. Often we may need to detect functional dependency for different target functions one at a time, or we may want to explore different base functions for a specific target function. Among these cases, the DLNs can be very similar and incremental SAT solving [22] can be helpful in amortizing computation overhead. For example, suppose that we have explored the functional dependency for target function f_0 under base functions $\{f_1, \ldots, f_n\}$. If we now intend to switch the target function to f_1 under base functions $\{f_0, f_2, \ldots, f_n\}$, only slight modification is needed to migrate from the original SAT instance, S_{DLN0}, to the new one, S_{DLN1}, because the sets of base functions are almost the same. Since the search spaces for the two SAT instances S_{DLN0} and S_{DLN1} are similar, the effort in solving S_{DLN0} can be partly reused to solve S_{DLN1}.

We investigate how to incorporate incremental SAT solving in our framework by reusing useful clauses learned from solving previous SAT instances for subsequent computation. Since not all previously learned clauses are valid in solving a current SAT instance, invalid clauses need to be disabled. To avoid sophisticated clause removal, we adopt the MiniSat [12] interface, where *unit assumptions* [12] can be made on a list of literals such that irrelevant clauses impose no constraint despite their presence, and thus SAT solving is conducted within the desired solution space.

To fast switch among various target and base-function selections and to effectively reuse learned clauses, we make unit assumptions on certain specific variables to enable or disable constraints. Depending on the type of DLN chosen, that of Figs. 1 or 2, we make different unit assumptions as follows:

For the DLN of Fig. 1, we introduce an auxiliary variable α_i for every equality constraint $(y_i \equiv y_i^*)$ with $i = 0, \ldots, n$. Further, in the CNF formula S_{DLN} of (1), every constraint $(y_i \equiv y_i^*)$ is replaced by $(\alpha_i \Rightarrow (y_i \equiv y_i^*))$. Thereby, the equality constraint $(y_i \equiv y_i^*)$ is enabled and disabled if α_i equals 1 and 0, respectively. For f_i being the target function with care condition f_c and the others being the base functions, the CNF formula S_{DLN} of (1) now becomes

$$\begin{aligned} S_{\text{on}} \wedge S_{\text{off}} \wedge y_i \wedge y_c \wedge \neg y_i^* \wedge y_c^* \wedge (\alpha_0 \Rightarrow (y_0 \equiv y_0^*)) \\ \wedge \cdots \wedge (\alpha_n \Rightarrow (y_n \equiv y_n^*)) \wedge \alpha_0 \wedge \cdots \wedge \alpha_{i-1} \wedge \neg\alpha_i \wedge \alpha_{i+1} \\ \wedge \cdots \wedge \alpha_n, \end{aligned} \tag{2}$$

where the unit clauses $\{y_i, \neg y_i^*, y_c, y_c^*, \alpha_0, \ldots, \alpha_{i-1}, \neg\alpha_i, \alpha_{i+1}, \ldots, \alpha_n\}$ are on the list of unit assumption. Note that, in (2) to remove any function f_j from the set of base functions, we can simply add $\neg\alpha_j$, instead of α_j, in the unit assumption. To perform interpolation for dependency function computation, we partition the above clause set into two subsets A and B with

$$\begin{aligned} A &= S_{\text{on}} \wedge y_i \wedge y_c \text{ and} \\ B &= S_{\text{off}} \wedge \neg y_i^* \wedge y_c^* \wedge (\alpha_0 \Rightarrow (y_0 \equiv y_0^*)) \wedge \cdots \wedge (\alpha_n \\ &\quad \Rightarrow (y_n \equiv y_n^*)) \wedge \alpha_0 \wedge \cdots \wedge \alpha_{i-1} \wedge \neg\alpha_i \wedge \alpha_{i+1} \wedge \cdots \wedge \alpha_n. \end{aligned}$$

On the other hand, for the DLN of Fig. 2, no auxiliary variable is needed. Rather, primary output variables are on the list of unit assumptions. Note that, to remove any function f_j from the set of base functions, we make no unit assumption on output variable e_j because such a free variable e_j imposes no constraint on the equality between y_j and y_j^*.

3.3 Enumeration and Minimization of Base-Function Set

The above incremental SAT solving is confined to the solution space induced by the unit assumption. As noted earlier, base functions can be selected by adjusting the unit assumption. Thereby, we can enumerate different sets of base functions and test their feasibility.

In addition to base-function enumeration, another use of incremental SAT solving is to reduce the size of a nonminimal set of base functions. If a SAT solving result is satisfiable, no functional dependency exists for the target function with respect to the selected base functions. Otherwise, the SAT solver returns a final conflict clause, which may refer only to a subset of the variables on the list of unit assumption. It indicates which variables on the unit assumption list are inconsistent, i.e., causing the unsatisfiability. It reveals also that there exists a dependency function depending only on the base functions that correspond to these inconsistent variables. Hence, it leads to a quick answer about the input size of the dependency function. Furthermore, a dependency function with fewer inputs can

be obtained. Because such a final conflict clause may not correspond to a *minimal* unsatisfiable core, it is possible to iteratively remove redundant base functions by adjusting the unit assumption. Upon finishing this iterative removal, the derived set of base functions is minimal.

Remark 1. One reason that our SAT-based approach outperforms BDD-based ones is due to its efficiency in enumerating base functions. Because of the scalability of our method, we can simply include all candidate base functions in the DLN and then perform iterative minimal unsatisfiable core (MUC) refinement to reduce the number of true base functions. In contrast, BDD-based methods can only detect functional dependency under a small set of candidate base functions; finding a good set of base functions can be very time consuming.

3.4 Flexibility Characterization of Dependency Functions

For a fixed target function f functionally depending on a set of base functions, it is often the case that the don't care set $\boldsymbol{B}^n \setminus \{h^0 \cup h^1\}$ for the dependency function h is not empty. This flexibility can be exploited to search a better implementation of h. The capability of SAT solvers, however, is limited in this respect of optimizing h as they tend to find "a" satisfying assignment or "a" refutation proof. A refutation proof uniquely determines an interpolant and, thus, an implementation of the dependency function. To overcome this deficiency, two orthogonal methods can be applied as follows:

First, by swapping the A and B-clause sets, the derived interpolant represents the offset of h, rather than the onset. Complementing this interpolant yields a legitimate dependency function. The Boolean difference of the two derived dependency functions under the two opposite interpretations of A and B-clause sets corresponds to the don't care condition of h. Note that the method only works for the interpolant construction procedure of [19], but not [18]. (With the procedure of [18], the computed don't care set will be empty because switching the roles of clause sets A and B results in complementary interpolants and thus the same dependency function.)

Second, by reordering the resolution sequence of a refutation proof, we may obtain different interpolants and different corresponding dependency functions. The reordering method suggested in [23] is one possibility (where variables local to the clause set A are resolved as early as possible to strengthen interpolants). By proper strengthening and weakening an interpolant, the difference between the strengthened and weakened interpolants provides a subset of the don't cares of h.

Practical experience, however, suggests that often the computed don't care set by the above two methods may not be large. Perhaps, a more effective way to derive different dependency functions is to force the SAT solver to produce a new refutation proof. This process can be achieved, for example, by varying the variable decision order of SAT solving.

4 Multiple-Output Functional Dependency

In this section, we generalize the result of Section 3 to **multiple-output functional dependency**, i.e., functional dependency with multiple target functions.

Simultaneous consideration of functional dependency among multiple target functions can be beneficial to achieve logic sharing. As a motivating example, let $f_1(X) = h_1(g_1(X), g_2(X), g_3(X))$ and $f_2(X) = h_2(g_4(X), g_5(X), g_6(X))$ be in some Boolean network originally. Suppose in this Boolean network, h_1 is the only output of g_1 and of g_2, and h_2 is the only output of g_5 and of g_6. If f_1 can be rewritten as $h_1'(g_3, g_4)$ and f_2 as $h_2'(g_3, g_4)$ for some dependency functions h_1' and h_2', then functions g_1, g_2, g_5, and g_6 are redundant and can be eliminated after the rewriting. Therefore, multiple-output functional dependency can be useful to logic synthesis for circuit simplification.

Such functional dependency can be particularly suitable for FPGA logic minimization, where any function with no more than k inputs (typically $k = 5$ or 6) can be realized using a lookup table (LUT) no matter how complex the function is. Since an interpolant can be very complex and highly redundant, it nevertheless can be implemented compactly within a single LUT as long as it has no more than k support variables. Moreover, for certain FPGA architectures, a single LUT may be optionally configured to produce two outputs implementing two different functions when their combined inputs are fewer than k (e.g., less than or equal to 4). Hence, two dependency functions with a few combined inputs can even be packed into a single LUT and achieve further simplification.

For multiple-output functional dependency, the existence conditions of functional dependency among a set of target functions, $f_1, \ldots, f_j$, with respect to the same set of base functions, $g_1, \ldots, g_n$, must hold simultaneously. Let f_{ci} be the care condition of function f_i. Then, this requirement can be represented as the Boolean network shown in Fig. 4. The multiple-output functional dependency exists if and only if all of the primary outputs of this Boolean network cannot be satisfied simultaneously. Once the existence of multiple-output functional dependency is asserted, the computation of dependency function for each f_i can be accomplished by using the single-output formulation of Section 3.

In fact, multiple-output functional dependency can be applied to the identification of a minimal set of independent base functions to express simultaneously a set of target functions. The computation can be achieved using incremental SAT solving and minimal unsatisfiable core refinement. More specifically, for the given set of target functions, when the output variable e_i is added as a unit clause by unit assumption, the corresponding g_i function is treated as a candidate base function for all of the target functions. Similar to the single-output case of Section 3.3, incremental SAT solving can be applied to enumerate and minimize the base functions. Notice that, given a computed minimal set of base functions for the target functions, the dependency function of an individual target function does not necessarily refer to all of the base functions.

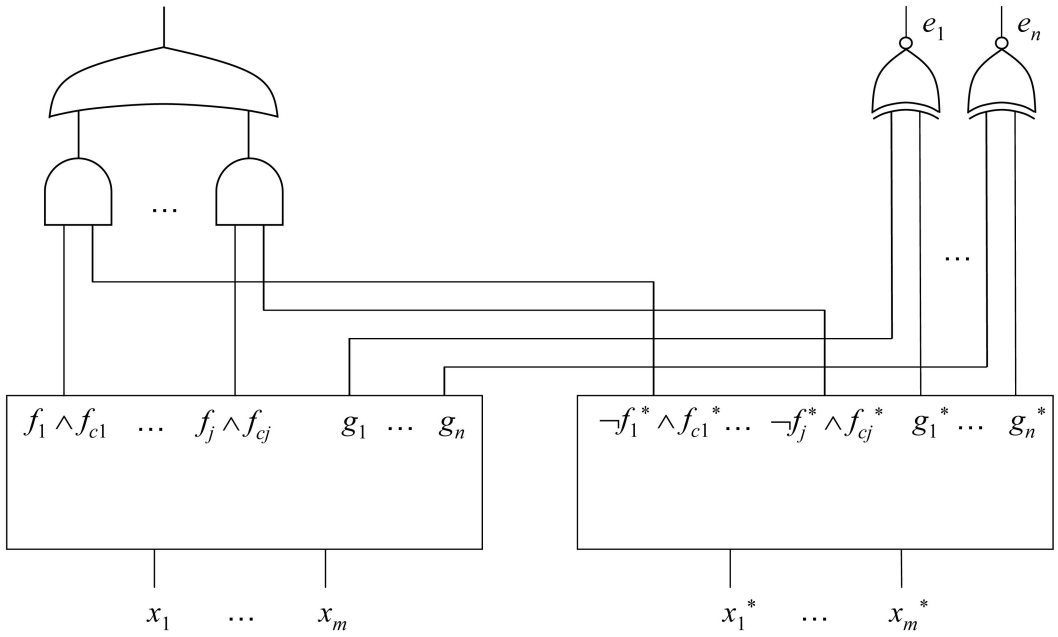

Fig. 4. Dependency logic network for checking multiple-output functional dependency.

5 Experimental Results

The proposed algorithm was implemented in ABC [24], which was modified to equip with the proof-logging version of MiniSat [12]. All the experiments are conducted on a 3.2 GHz Linux machine with 2 GB memory. The experiments are designed so as to demonstrate

1. the efficiency and scalability of the SAT-based method, in contrast to prior BDD-based computation [1],
2. the benefit of using incremental SAT formulation,
3. the characteristics of the derived dependency functions, and
4. the effectiveness of minimal unsatisfiable core refinement for support-variable reduction of dependency functions.

Large circuits from the ISCAS89 and ITC99 benchmark suites are chosen. To have fair comparison with [1], functional dependency among the transition functions of a circuit is explored. Among the transition functions of a given circuit, each of them is specified in turn as the target function and all others as base functions. We then explore the corresponding functional dependency and compute dependency functions if they exist. Here, we do not aim at identifying a minimal set of independent base functions to express all other functions, and rather focus on demonstrating the scalability of the new method. As mentioned in Section 4, multiple-output functional dependency can be applied to identifying a minimal set of independent transition functions to express all other functions. A discussion on the application to redundant register identification and verification reduction is in [1].

Table 1 compares our approach with the prior work [1]. Columns 1 and 2, respectively, list the name and the number of nodes of each circuit. The numbers of flip-flops, denoted #FF, of a circuit and its retimed version (min-period retiming using ABC) are listed in Columns 3 and 6, respectively. Among the flip-flops of an original circuit (respectively, a retimed circuit), those whose transition functions possess functional dependency are counted in Columns 4 and 5 (respectively, Columns 7 and 8), denoted as #Dep. In particular, #Dep-S and #Dep-B are obtained by the SAT and BDD-based methods, respectively. In fact, #Dep-S data are exact and complete. In comparison, the BDD-based method only succeeded in a few circuits and detected only a subset of the dependency over a few support variables. The runtime (in seconds) and memory (in MB, megabytes) usages are shown in the following columns. Note that the reported memory usage includes the underlying system memory. Specifically, the prior work was built on VIS [25], whereas ours was on ABC. Despite the uneven comparison, the scalability of our approach is evident and outperforms the prior work. As plotted in Fig. 5, it is easily seen from the power regression lines that the runtime of our approach scales polynomially with the circuit size.

The strength of incremental SAT solving in accelerating computation is shown in Fig. 6. The x-axis and y-axis, respectively, represent the iteration number and the runtime of solving the SAT instance at a particular iteration. The y-axis is log-scaled. Four sample circuits of different sizes from Table 1 are plotted for the first 100 iterations. As can be seen in all of the plots, the runtimes of the first iterations are the maximum among their first 100 iterations. In fact, all the circuits of Table 1 exhibit the same behavior. After the first iteration, the runtimes for SAT solving decrease rapidly and become relatively short and stable within about 10 iterations. This demonstrates the effectiveness of incremental SAT solving.

The above experiments tend to suggest that 1) the average runtime for a circuit is linear in the number of its nodes and 2) the solving time for an unsatisfiable SAT instance is often much shorter than that for a satisfiable one. The statistics are plotted in Fig. 7. The x-axis and y-axis, respectively, represent the number of nodes of a circuit and the corresponding

TABLE 1
SAT versus BDD-Based Exploration of Single-Output Functional Dependency

Circuit	#Nodes	Original			Retimed			SAT (original)		BDD (original)		SAT (retimed)		BDD (retimed)	
		#FF.	#Dep-S	#Dep-B	#FF.	#Dep-S	#Dep-B	Time	Mem	Time	Mem	Time	Mem	Time	Mem
s5378	2794	179	52	25	398	283	173	1.2s	18m	1.6s	20m	0.6s	18m	7s	51m
s9234.1	5597	211	46	x	459	301	201	4.1s	19m	x	x	1.7s	19m	194.6s	149m
s13207.1	8022	638	190	136	1930	802	x	15.6s	22m	31.4s	78m	15.3s	22m	x	x
s15850.1	9785	534	18	9	907	402	x	23.3s	22m	82.6s	94m	7.9s	22m	x	x
s35932	16065	1728	0	--	2026	1170	--	176.7s	27m	1117s	164m	78.1	27m	--	--
s38417	22397	1636	95	--	5016	243	--	270.3s	30m	--	--	123.1	32m	--	--
s38584	19407	1452	24	--	4350	2569	--	166.5s	21m	--	--	99.4s	30m	1117s	164m
b12	946	121	4	2	170	66	33	0.15s	17m	12.8s	38m	0.13s	17m	2.5s	42m
b14	9847	245	2	--	992	724	--	3.3s	22m	--	--	5.2s	22m	--	--
b15	8367	449	0	--	1134	793	--	5.8s	22m	--	--	5.8s	22m	--	--
b17	30777	1415	0	--	3967	2350	--	119.1s	28m	--	--	161.7s	42m	--	--
b18	111241	3320	5	--	9254	5723	--	1414.9s	100m	--	--	2842.6s	100m	--	--
b19	224624	6642	0	--	7164	337	--	8184.8s	217m	--	--	11040.6s	234m	--	--
b20	19682	490	4	--	1604	1167	--	25.7s	28m	--	--	36	30m	--	--
b21	20027	490	4	--	1950	1434	--	24.6s	29m	--	--	36.3	31m	--	--
b22	29162	735	6	--	3013	2217	--	73.4s	36m	--	--	90.6	37m	--	--

("--": memory usage exceeds 1Gb; "x": runtime exceeds 12,000 seconds.)

average runtime of SAT iterations. Both axes are log-scaled. Every circuit in Table 1 is plotted as a spot in Fig. 7. The first tendency can be seen from the two power regression lines indicating highly (positive) correlated data sets. The second tendency can be seen from the fact that the line for the retimed circuits is well below that for the original circuits. As evident in Columns 4 and 7 of Table 1, more functional dependency exists for the retimed circuits. Effectively, more unsatisfiable SAT instances are there. It reflects the fact that in our experiments a satisfiable instance usually takes longer time to solve than an unsatisfiable one. It seems contradicting with common sense. However, the tendency can be explained as follows: Because the input sizes of interpolants are mostly very small (to be shown in Fig. 8), it suggests that conflicts can be found locally. Also, incremental SAT solving increases *implicativity* [26] and thus enhances early conflict detection. As a result, decisions over only a few variables might be enough to draw a conclusion about unsatisfiability. In contrast, in a satisfiable case to obtain a single satisfying assignment, decisions must be made over all variables. (Note that it may not be necessarily so for circuit-SAT and other ODC-aware CNF solvers, where not all variables are valuated for a satisfying assignment.)

We characterize the derived dependency functions (i.e., interpolants) in terms of their input sizes in Fig. 8, where a single dependency function, if it exists, is derived for each transition function of a given circuit. The x-axis and y-axis indicate the number of support variables and the corresponding number of dependency functions of a particular size, respectively. The y-axis is log-scaled. As can be seen, most of the functions have less than 10 support variables. On the other hand, complex functional dependency can also be detected easily by the SAT-based approach. For instance, in the retimed b18 circuit, a dependency function of input size around 300 is obtained, which is not possible using BDD-based methods.

To see if the input variables of an interpolant are indeed irredundant (i.e., support variables), Fig. 9 gives the

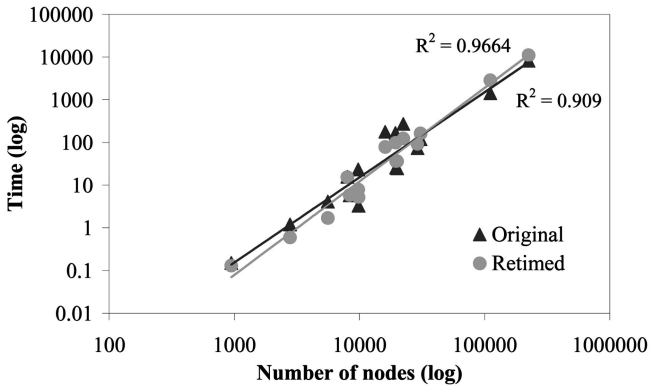

Fig. 5. Runtime versus circuit size.

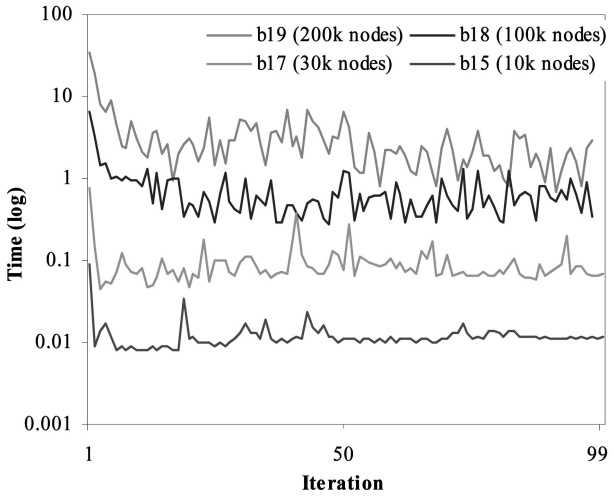

Fig. 6. Runtime of the first 100 SAT iterations.

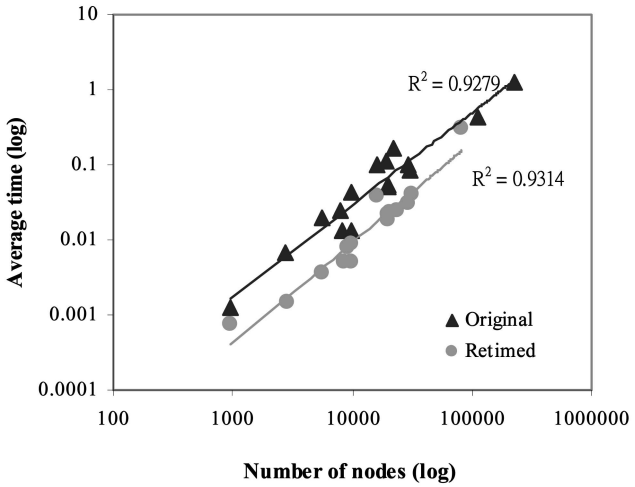

Fig. 7. Average runtime of SAT iterations versus circuit size.

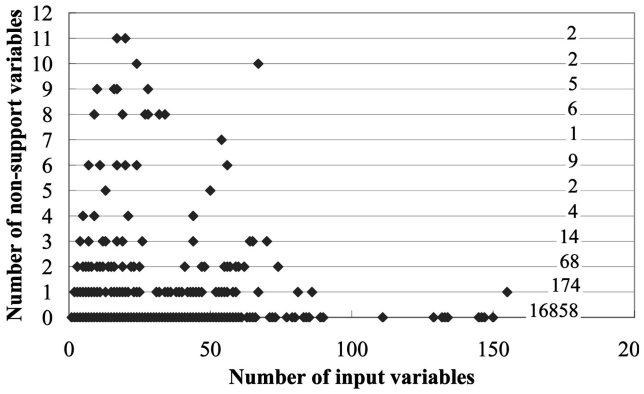

Fig. 9. Number of total input variables versus number of nonsupport input variables of derived interpolants.

statistics. The x-axis and y-axis indicate the number of input variables and the corresponding nonsupport variables of an interpolant, respectively. (Note that in the figure, multiple interpolants may locate at the same spot.) For every y-axis index, say i, the corresponding number on the right-hand side of Fig. 9 indicates the number of interpolants with i nonsupport variables. As can be seen, most (98.3 percent) interpolants have irredundant input variables. Only in a few occasions, the derived interpolants contain redundancy.

On the other hand, Fig. 10 shows the relation between the interpolant size in terms of And-Inverter Graph (AIG) nodes and the number of input variables. The size grows polynomially with respect to the number of input variables, as seen from the two power regression lines. Figs. 8, 9, and 10 suggest that the derived interpolants are reasonably small.

In Fig. 11, we study the usefulness of minimal unsatisfiable core refinement to minimize the set of base functions and, thus, reduce the support variables of a dependency function. (Note that the removal of a base function in the refinement does not necessarily mean that it is a nonsupport variable of the original interpolant since the refinement can produce new different interpolants.) In the figure, a base-function set reducible under such refinement is plotted. As can be seen, the reduction can be substantial. In particular, for retimed circuits, the average reduction of reducible base-function sets is about 47 percent.

From practical experience in enumerating different dependency functions for a target function, we note that the number of available dependency functions (with different support-variable sets) varies greatly from function to function. Moreover, a great amount of trivial dependency may exist due to the transitivity of dependency. This transitivity can result in vast redundant enumeration. How to effectively avoid unnecessary enumeration remains to be solved. Nevertheless, if the candidate base functions are specified (e.g., for circuit rewiring), finding a dependency function is easy. On the other hand, we emphasize that the BDD-based method is more effective than the SAT-based one in computing the don't care set for a dependency function. This deficiency of the SAT-based computation is due to the fact that an interpolant (i.e., a dependency function) is with respect to a refutation and contains no don't care information.

6 Related Work

The previous efforts closest to ours are [1] and [5]. Both of them rely on BDD-based computation. In [1], combinational functional dependency was generalized to sequential dependency. In this paper, we focus on combinational dependency. In [5], similar to our enumeration for different dependency functions, a BDD-based technique was used. It allows a more implicit enumeration. However, the size of the

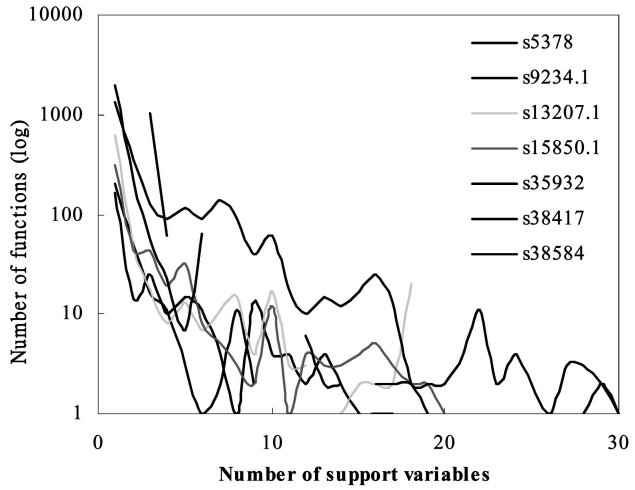

Fig. 8. Frequency distribution of different support sizes.

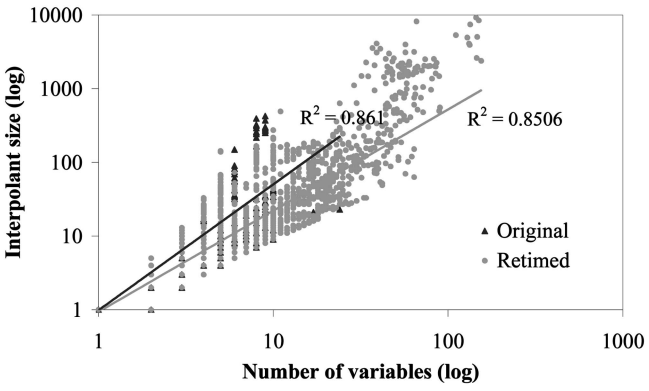

Fig. 10. Interpolant size versus number of input variables.

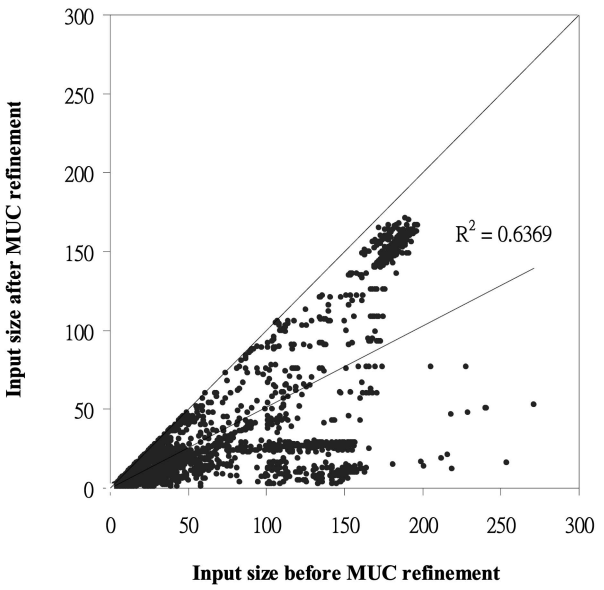

Fig. 11. Number of input variables before and after MUC refinement.

set of base functions was limited to no more than 16. Other work [19] applied interpolation in the context of SAT-based model checking for approximated image computation.

7 Conclusions and Future Work

We have shown that the exploration of functional dependency can be solved by a pure SAT-based formulation. Experimental results demonstrated the great success of the proposed method. The approach is scalable to large designs and discovers much more functional dependency far beyond the capability of prior methods. The success is attributed to several key ingredients including Craig interpolation and incremental SAT solving. We hope that our results may benefit several areas of logic synthesis and formal verification, for example, in finding rewiring and resubstitution candidates for circuit optimization, in identifying redundant registers in RTL synthesis, in reducing state space in formal sequential equivalence checking, etc.

Future work will include integrating our technique in logic synthesis and generalizing it for other applications. Some preliminary results in synthesizing industrial designs have been demonstrated in [27]. Generalizing our method for sequential dependency [1] is an important subject to pursue. In addition, it would be interesting to explore new applications of Craig interpolation. In fact, it was demonstrated recently that bidecomposition [28] and Ashenhurst decomposition [29] can be applied to large Boolean functions through SAT solving and Craig interpolation.

Acknowledgments

This work was supported in part by the National Science Council under Grants 95-2221-E-002-432 and 95-2218-E-002-064-MY3. AM was supported by the Intel-custom SRC grant 1444.001 "Innovative Sequential Synthesis and Verification." The authors are grateful to Robert Brayton for helpful discussions and to Ruei-Rung Lee for preparing some of the experimental data. This manuscript is an extended version of the earlier publication [2].

References

[1] J.-H.R. Jiang and R.K. Brayton, "Functional Dependency for Verification Reduction," *Proc. Int'l Conf. Computer Aided Verification,* pp. 268-280, 2004.

[2] C.-C. Lee, J.-H.R. Jiang, C.-Y. Huang, and A. Mishchenko, "Scalable Exploration of Functional Dependency by Interpolation and Incremental SAT Solving," *Proc. Int'l Conf. Computer-Aided Design,* pp. 227-233, 2007.

[3] E. Sentovich, H. Toma, and G. Berry, "Latch Optimization in Circuits Generated from High-Level Descriptions," *Proc. Int'l Conf. Computer-Aided Design,* pp. 428-435, 1996.

[4] B. Lin and A.R. Newton, "Exact Redundant State Registers Removal Based on Binary Decision Diagrams," *Proc. Int'l Conf. Very Large Scale Integration,* pp. 277-286, 1991.

[5] V. Kravets and P. Kudva, "Implicit Enumeration of Structural Changes in Circuit Optimization," *Proc. Design Automation Conf.,* pp. 438-441, 2004.

[6] A.J. Hu and D.L. Dill, "Reducing BDD Size by Exploiting Functional Dependencies," *Proc. Design Automation Conf.,* pp. 266-271, 1993.

[7] C. Berthet, O. Coudert, and J.-C Madre, "New Ideas on Symbolic Manipulations of Finite State Machines," *Proc. Int'l Conf. Computer Design,* pp. 224-227, 1990.

[8] C.A.J. van Eijk and J.A.G. Jess, "Exploiting Functional Dependencies in Finite State Machine Verification," *Proc. European Design and Test Conf.,* pp. 9-14, 1996.

[9] E. Gregoire, R. Ostrowski, B. Mazure, and L. Sais, "Automatic Extraction of Functional Dependencies," *Proc. Int'l Conf. Theory and Applications of Satisfiability Testing,* pp. 122-132, 2004.

[10] R.E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. Computers,* vol. 35, no. 8, pp. 677-691, Aug. 1986.

[11] M. Moskewicz, C. Madigan, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," *Proc. Design Automation Conf.,* pp. 530-535, 2001.

[12] N. Eén and N. Sörensson, "An Extensible SAT-Solver," *Proc. Int'l Conf. Theory and Applications of Satisfiability Testing,* pp. 502-518, 2003.

[13] A. Mishchenko and R. Brayton, "SAT-Based Complete Don't-Care Computation for Network Optimization," *Proc. Design, Automation and Test in Europe,* pp. 418-423, 2005.

[14] A. Mishchenko, J. Zhang, S. Sinha, J. Burch, R.K. Brayton, and M. Chrzanowska-Jeske, "Using Simulation and Satisfiability to Compute Flexibilities in Boolean Networks," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems,* vol. 25, no. 5, pp. 742-755, May 2006.

[15] W. Craig, "Linear Reasoning: A New Form of the Herbrand-Gentzen Theorem," *J. Symbolic Logic,* vol. 22, no. 3, pp. 250-268, 1957.

[16] J.A. Robinson, "A Machine-Oriented Logic Based on the Resolution Principle," *J. ACM,* vol. 12, no. 1, pp. 23-41, 1965.

[17] J. Krajicek, "Interpolation Theorems, Lower Bounds for Proof Systems, and Independence Results for Bounded Arithmetic," *J. Symbolic Logic,* vol. 62, no. 2, pp. 457-486, June 1997.

[18] P. Pudlak, "Lower Bounds for Resolution and Cutting Plane Proofs and Monotone Computations," *J. Symbolic Logic,* vol. 62, no. 3, pp. 981-998, Sept. 1997.

[19] K.L. McMillan, "Interpolation and SAT-Based Model Checking," *Proc. Int'l Conf. Computer Aided Verification,* pp. 1-13, 2003.

[20] G.S. Tseitin, "On the Complexity of Derivation in Propositional Calculus," *Studies in Constructive Mathematics and Mathematical Logic, Part II,* pp. 115-125, Consultants Bureau, 1970.

[21] D. Plaisted and S. Greenbaum, "A Structure Preserving Clause form Translation," *J. Symbolic Computation,* vol. 2, pp. 293-304, 1986.

[22] J. Whittemore, J. Kim, and K. Sakallah, "SATIRE: A New Incremental Satisfiability Engine," *Proc. Design Automation Conf.,* pp. 542-545, 2001.

[23] R. Jhala and K.L. McMillan, "Interpolant-Based Transition Relation Approximation," *Proc. Int'l Conf. Computer Aided Verification,* pp. 39-51, 2005.

[24] Berkeley Logic Synthesis and Verification Group, ABC: A System for Sequential Synthesis and Verification, http://www.eecs.berkeley.edu/~alanmi/abc/, 2005.

[25] R.K. Brayton et al., "VIS: A System for Verification and Synthesis," *Proc. Int'l Conf. Computer Aided Verification,* pp. 428-432, 1996.

[26] Y. Novikov and R. Brinkmann, "Foundations of Hierarchical SAT-Solving," *Proc. Int'l Workshop Boolean Problems,* 2004.
[27] A. Mishchenko, R. Brayton, J.-H.R. Jiang, and S. Jang, "SAT-Based Logic Optimization and Resynthesis," *Proc. Int'l Workshop Logic and Synthesis,* pp. 358-364, 2007.
[28] R.-R Lee, J.-H.R. Jiang, and W.-L Hung, "Bi-Decomposing Large Boolean Functions via Interpolation and Satisfiability Solving," *Proc. Design Automation Conf.,* pp. 636-641, 2008.
[29] H.-P. Lin, J.-H.R. Jiang, and R.-R Lee, "To SAT or Not to SAT: Ashenhurst Decomposition in a Large Scale," *Proc. Int'l Conf. Computer-Aided Design,* 2008.

Jie-Hong Roland Jiang (S'03-M'04) received the B.S. and M.S. degrees in electronics engineering, both from National Chiao Tung University, Hsinchu, Taiwan, in 1996 and 1998, respectively, and the Ph.D. degree in electrical engineering and computer sciences from the University of California, Berkeley, in 2004. During his compulsory military service, from 1998 to 2000, he was a Second Lieutenant with the Air Force, R.O.C. In 2005, he joined the Department of Electrical Engineering of National Taiwan University, where he is an associate professor. His research interests include logic synthesis, verification, and computational aspects of physical and biological systems. Dr. Jiang is a member of the IEEE, Phi Tau Phi and the Association for Computing Machinery (ACM).

Chih-Chun Lee received the B.S. degree and the M.S. degree in Electronics Engineering from National Taiwan University in 2005 and 2007, respectively. Afterwards, he worked as a research assistant in Academia Sinica, Taiwan. He became a Ph.D. student in Electronics Engineering at National Taiwan University since Fall 2009. His research interests include logic synthesis, formal verification, and model checking.

Alan Mishchenko (M'99) received the M.S. degree in applied mathematics and information technology from the Moscow Institute of Physics and Technology, Moscow, Russia, in 1993, and the Ph.D. degree in computer science from the Glushkov Institute of Cybernetics, Kiev, Ukraine, in 1997. Since 1998, he has been a Re-search Scientist in the U.S. From 1998 to 2002, he was with the Portland State University, Portland, OR. Since 2002, he has been with the University of California, Berkeley. His research interests include developing computationally efficient methods for logic synthesis and verification. He is a member of the IEEE.

Chung-Yang (Ric) Huang (S'96-M'00) received his B.S. degree from Department of Electrical Engineering, National Taiwan University (NTUEE), in 1992. He obtained his PhD from Department of Electrical and Computer Engineering, University of California at Santa Barbara, in 2000. Before joining NTUEE as an assistant professor in 2004, he was with Cadence Design Systems (ex-Verplex Systems) for 6 years, where he served as a senior R&D manager and was in charge of the core engine development of their formal functional verification tools. He also worked in Avant! Corp as a summer intern on the post-layout timing optimization problems in 1996 and 1997. Professor Huang's research interests include (1) design verification for System-on-Chips, (2) design for verifiability, (3) design automation and optimization, and (4) constraint satisfaction problems. He is a member of the IEEE.