SAT-Based Combinational and Sequential Dependency Computation

Mathias Soeken[1(✉)], Pascal Raiola[2], Baruch Sterin[3], Bernd Becker[2], Giovanni De Micheli[1], and Matthias Sauer[2]

[1] EPFL, Lausanne, Switzerland
mathias.soeken@epfl.ch
[2] University of Freiburg, Freiburg im Breisgau, Germany
[3] UC Berkeley, Berkeley, CA, USA

Abstract. We present an algorithm for computing both functional dependency and unateness of combinational and sequential Boolean functions represented as logic networks. The algorithm uses SAT-based techniques from *Combinational Equivalence Checking* (CEC) and *Automatic Test Pattern Generation* (ATPG) to compute the dependency matrix of multi-output Boolean functions. Additionally, the classical dependency definitions are extended to sequential functions and a fast approximation is presented to efficiently yield a sequential dependency matrix. Extensive experiments show the applicability of the methods and the improved robustness compared to existing approaches.

1 Introduction

In this paper we present an algorithm to compute the *dependency matrix* $D(f)$ for a given combinational or sequential multi-output function f. For every input-output pair, the combinational dependency matrix indicates whether the output depends on the input, and whether the output is positive or negative unate in that input [21].

Several algorithms in logic design use the dependency matrix as a *signature* [23] to speed up computation, e.g., Boolean matching [14], functional verification [11,12], or reverse engineering [29]. Although most of these algorithms make implicit use of the dependency matrix, the name has been used in this paper for the first time. The name is inspired by the output format of functional dependence and unateness properties in the state-of-the-art academic logic synthesis tool ABC [4]. Functional dependency is also related to *transparent logic* [19,24]. Given a set of inputs X_d and a set of outputs Y_d, the problem is to find a set of inputs X_c that is disjoint from X_d and that distinguishes the output values at Y_d for different input assignments to X_d. In contrast, we consider functional dependence without constraints for all input-output pairs.

Existing algorithms for computing the dependency matrix are based on Binary Decision Diagrams (BDDs, [5]) and have been implemented in ABC [4]. It is important to point out that the term *functional dependence* is used

R. Bloem and E. Arbel (Eds.): HVC 2016, LNCS 10028, pp. 1–17, 2016.
DOI: 10.1007/978-3-319-49052-6_1

to describe a different property in a related context: In [12,13,16], the authors refer to functional dependence as the question whether given a set of Boolean functions $\{f_1, \ldots, f_n\}$, there exists an f_i, that can be written as $h(f_1, \ldots, f_{i-1}, f_{i+1}, \ldots, f_n)$. In other words, functional dependence is defined as a Boolean function w.r.t. to a set of Boolean functions. In contrast, we consider the functional dependence of a Boolean function w.r.t. a single variable as functional dependence.

Our algorithm uses techniques from *Combinational Equivalence Checking* (CEC, e.g., [22]) and *Automatic Test Pattern Generation* (ATPG, e.g., [15,26,27]). We employ efficient incremental SAT-based solving techniques and extract incidental information from solved instances to reduce runtime consumption on complex functions.

We furthermore present an extension of the combinational dependency definition to sequential functions. We account the sequential counterpart of a functional dependence relation to an input-output pair if the given relation constantly holds after some finite number of steps. As an example, some output f may be always positive unate in some input x after a certain number of iteration steps of the circuit. In this case, we call f sequential positive unate in x, even if this relation is not guaranteed in the first steps.

An established method to prove properties on sequential circuits is bounded model checking (BMC) as first introduced in [1], used, e.g., in [8,25]. In BMC a circuit is modelled iteratively for k steps as a combinational circuit. With approaches such as k-induction [18] and Craig interpolation [20] BMC becomes a complete model checking method. However, as such complete methods are rather computationally expensive, we rely on an iterative approximation to compute the sequential dependency matrix solely based on the combinational dependency matrix. By iteratively analyzing the combinational dependency until a fixed point is derived, we can accurately conclude structural dependency and unateness.

In an extensive experimental evaluation we demonstrate the applicability of our methods to various combinational and sequential benchmark sets. Within reasonable amounts of computing time we are able to accurately compute the combinational dependency matrix as well as an approximation of our sequential dependency matrix with a small number of iterations. We further show the robustness of our proposed algorithm compared to a previous state-of-the-art algorithm that times out or suffers from memory explosion on complex functions. Finally, we present a case study in which the dependency matrix is used as a signature in reverse engineering to effectively reduce the search space and improve the performance of the underlying application.

The rest of the paper is organized as follows. Section 2 presents the fundamentals of the work. In Sect. 3 we introduce our SAT-based approach to compute the dependency matrix of combinational circuits, and extend it in Sect. 4 to sequential circuits. The experimental results are presented in Sect. 5 and Sect. 6 concludes the work.

2 Background

2.1 Functional Dependencies

A Boolean function $f(x_1, \ldots, x_n)$ is *functionally dependent* in x_i if $f_{\bar{x}_i} \neq f_{x_i}$ where the *co-factors* f_{x_i} or $f_{\bar{x}_i}$ are obtained by setting x_i to 1 or 0 in f, respectively. We call f_{x_i} the positive co-factor and $f_{\bar{x}_i}$ the negative co-factor. The function f is said to be *positive unate* in x_i, if

$$f_{\bar{x}_i} \leq f_{x_i} \tag{1}$$

and *negative unate* in x_i, if

$$f_{\bar{x}_i} \geq f_{x_i}, \tag{2}$$

where the comparisons '$\leq$' and '$\geq$' are applied to the binary strings that represent the truth tables of $f_{\bar{x}_i}$ and f_{x_i}. f is said to be unate in x_i if it is either positive or negative unate in x_i. Clearly, a function f is both positive and negative unate in x_i, if f does not depend on x_i. Hence, we call f *strictly* positive (negative) unate in x_i, if f is positive (negative) unate in x_i *and* depends on x_i. If f is neither positive nor negative unate in x_i, we say that f is *binate* in x_i.

Example 1. The functions $x_1 \wedge x_2$ and $x_1 \vee x_2$ are positive unate in both x_1 and x_2. The function $x_1 \rightarrow x_2$ is negative unate in x_1 and positive unate in x_2. The function $x_1 \oplus x_2$ is binate in both variables.

Let $f : \mathbb{B}^n \rightarrow \mathbb{B}^m$ be a multi-output Boolean function where each output is represented by a Boolean function $f_j(x_1, \ldots, x_n)$. The *dependency matrix* $D(f)$ is an $m \times n$ matrix with entries $d_{j,i}$ where

$$d_{j,i} = \begin{cases} \texttt{p} & \text{if } f_j \text{ is strictly positive unate in } x_i, \\ \texttt{n} & \text{if } f_j \text{ is strictly negative unate in } x_i, \\ \texttt{d} & \text{if } f_j \text{ depends on, but is not unate in } x_i, \\ \bullet & \text{otherwise.} \end{cases} \tag{3}$$

Example 2. Let $f : \mathbb{B}^5 \rightarrow \mathbb{B}^3$ with $f_1 = x_1 \wedge x_2$, $f_2 = x_3 \rightarrow x_5$, and $f_3 = x_1 \oplus x_2 \oplus x_5$. Then

$$D(f) = \begin{bmatrix} \texttt{p} & \texttt{p} & \bullet & \bullet & \bullet \\ \bullet & \bullet & \texttt{n} & \bullet & \texttt{p} \\ \texttt{d} & \texttt{d} & \bullet & \bullet & \texttt{d} \end{bmatrix}.$$

2.2 Boolean Satisfiability

In our algorithm we translate decision problems into instances of the SAT problem [3]. SAT is the problem of deciding whether a function f, has an assignment x for which $f(x) = 1$. Such an assignment is called a *satisfying assignment*. If f has a satisfying assignment it is said to be *satisfiable*. Otherwise, f is said to be *unsatisfiable*.

In general, SAT is NP-complete [6,17]. SAT solvers are algorithms that can solve SAT problems and, while worst-case exponential, are nonetheless very efficient for many practical problems. SAT solvers also return a satisfying assignment if the instance is satisfiable. Most of the state-of-the-art SAT solvers are conflict-driven and employ clause-learning techniques [10]. In *incremental* SAT one asks whether f is satisfiable under the assumption of some variable assignments. These assignments are only temporarily assumed, making it possible to reuse the SAT instance and learned information when solving a sequence of similar SAT problems. In the remainder of the paper, we refer to instances of SAT as if they were calls to an incremental SAT solver. SAT?(f, α) is true if f is satisfiable under the assumptions α, and UNSAT?(f, α) is true if f is unsatisfiable under the assumptions α.

3 SAT-Based Dependency Computation

This section presents the SAT-based algorithm to compute the functional dependencies of a function. We first describe the encoding into SAT, then an implementation of the algorithm, and finally possible optimizations.

3.1 SAT Encoding

We encode the test for functional dependence and unateness as an instance of the SAT problem using the following theorem.

Theorem 1. *Let $f(x_1, \ldots, x_n)$ be a Boolean function. Then*

1. *f is functionally dependent in x_i, if and only if $f_{\bar{x}_i} \oplus f_{x_i}$ is satisfiable,*
2. *f is positive unate in x_i, if and only if $f_{\bar{x}_i} \wedge \bar{f}_{x_i}$ is unsatisfiable, and*
3. *f is negative unate in x_i, if and only if $f_{x_i} \wedge \bar{f}_{\bar{x}_i}$ is unsatisfiable.*

Proof. We only show the direction of "if"; the "only if" direction follows immediately from the definition of functional dependency and unateness.

1. Let x be a satisfying assignment to $f_{\bar{x}_i} \oplus f_{x_i}$. Then, we have $f_{\bar{x}_i}(x) \neq f_{x_i}(x)$.
2. Assume the function was satisfiable and let x be a satisfying assignment. Then $f_{\bar{x}_i}(x) = 1$ while $f_{x_i}(x) = 0$ which contradicts Eq. (1).
3. Analogously to (2). □

In the implementation, we make use of the following immediate consequence of the theorem.

Corollary 1. *f is functionally independent in x_i, if and only if $f_{\bar{x}_i} \oplus f_{x_i}$ is unsatisfiable.*

In the following we consider multi-output functions $f : \mathbb{B}^n \to \mathbb{B}^m$, where each output is a function f_j. In order to compute the full dependency matrix which contains the dependency for each input-output pair, we transform the problem to a sequence of SAT instances as illustrated by the generic miter in Fig. 1. The

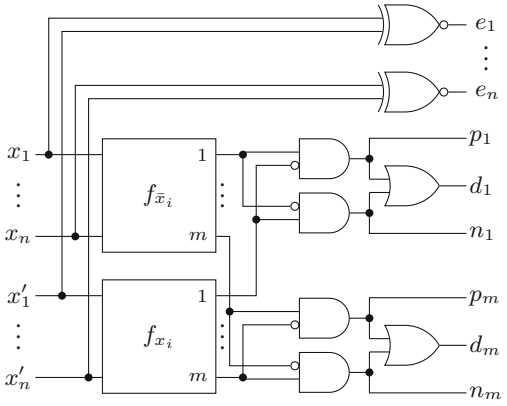

Fig. 1. Generic miter to encode functional dependency as SAT instance

two boxes represent two copies of f. The upper copy, with inputs $x_1, \ldots, x_n$, is used as the negative co-factor, while the lower copy, with inputs $x'_1, \ldots, x'_n$, is used as the positive co-factor of f. The groups of three gates on the lower right hand side realize the XOR operation which connect the outputs of the two copies. The signals of the AND gates are exposed as outputs and will be used to encode the unateness problems. The XNOR gates in the upper right of the figure are used to force all but one of the inputs, to have equal values.

Let $\Pi(f_j)$ be the characteristic Boolean function which is obtained by encoding the miter in Fig. 1 for the function f_j using encodings such as Tseytin [32] or EMS [9]. Also let $E_i = \{x_i = 0,\ x'_i = 1\} \cup \{e_k = 1 \mid k \neq i\}$ be assignments that lead to a correct interpretation of the miter for input x_i, i.e., x_i is set to 0, x'_i is set to 1 and all the other inputs need to have the same value. We define three problems on top of the incremental SAT interface:

$$\text{DEP}(f_j, x_i) = \text{SAT?}(\Pi(f_j), E_i \cup \{d_j = 1\}) \tag{4}$$

$$\text{POS\_UNATE}(f_j, x_i) = \text{UNSAT?}(\Pi(f_j), E_i \cup \{p_j = 1\}) \tag{5}$$

$$\text{NEG\_UNATE}(f_j, x_i) = \text{UNSAT?}(\Pi(f_j), E_i \cup \{n_j = 1\}) \tag{6}$$

Then the problems described in Theorem 1 and Corollary 1 can be solved as follows. The function f_j functionally depends on x_i, if $\text{DEP}(f_j, x_i)$ holds. And the function f_j is positive (negative) unate in x_i, if $\text{POS\_UNATE}(f_j, x_i)$ ($\text{NEG\_UNATE}(f_j, x_i)$) holds.

3.2 Algorithm

Figure 2 displays the general flow of the algorithm. For each pair of an input $x = x_i$ and an output $y = f_j$ the algorithm starts with a simple structural dependency check. If x is outside of y's structural cone of influence, it can be concluded that y is independent of x. This is a very efficient check. Otherwise,

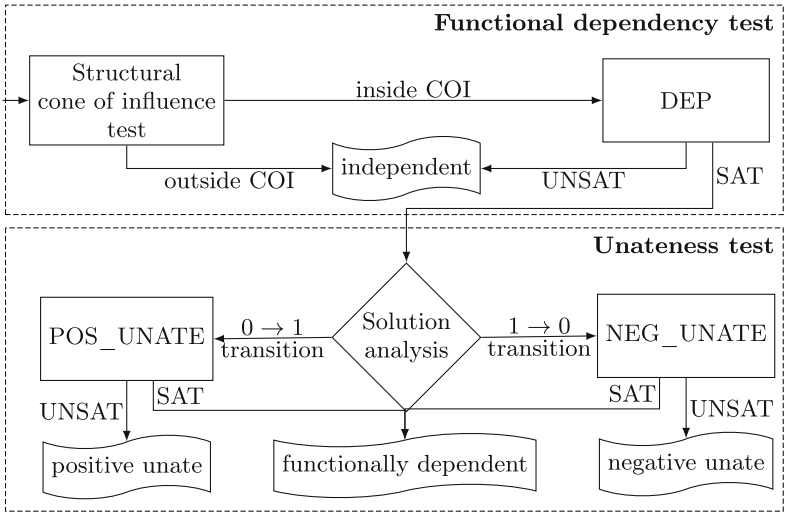

Fig. 2. Functional dependency and unateness computation flow

the algorithm proceeds with a functional dependency check $\mathrm{DEP}(y, x)$ as defined in Eq. (4). We omit the arguments from the boxes.

If the instance is unsatisfiable, y is independent from x as no assignment exists that results in different logic values for y under the side constraint of $y_{\bar{x}} \oplus y_x$. In case the instance is satisfiable, x and y are at least functionally dependent. Additionally, the SAT solver returns a satisfying assignment which is analyzed for the logic value of y. In case $y_{\bar{x}}$ is 1 (and therefore y_x is 0), y cannot be positive unate in x as a counter example for Eq. (1) is found. Likewise, negative unateness can be falsified if $y_{\bar{x}}$ is 0. Note that one of the two cases must hold as the original instance requires a difference between $y_{\bar{x}}$ and y_x.

In a last step, the algorithm specifically checks for unateness with an additional call to the SAT solver, unless it has been ruled out previously. If this SAT call is unsatisfiable, unateness can be concluded, otherwise the algorithm returns functional dependence.

3.3 Optimizations

As discussed above, we use incremental SAT solving because many of the calls to the SAT solver are very similar. Hence, instead of encoding a miter-like structure as illustrated in Fig. 1 for each input-output pair in an individual instance, we encode the complete output cone of a target input x_i in a single instance to profit from incremental SAT solving. We enforce the co-factors of x_i as unit clauses in this instance. As we target combinational circuits, the direction of the co-factors does not influence the satisfiability of the instance. Hence, we can restrict the search space by enforcing x_i to logic 1 and x_i' to logic 0 without loss of generality. Furthermore, XOR gates are encoded for output pairs to enforce them to differ using assumptions in the SAT solver.

On the output side, we iteratively run through each output in x's cone of influence and enforce a difference between $f_{\bar{x}_i}$ and f_{x_i} using an assumption. If the resulting instance is UNSAT we can conclude independence. Otherwise, the input-output pair is at least functionally dependent. By observing the direction of the difference at the output, we consider the pair either as a candidate for positive or negative unateness and run the respective check as described earlier.[1]

Additionally, we perform a forward looking logic analysis of each satisfiable SAT instance to extract incidental information for future solver calls. In our experiments we found quite often, that the difference not only propagates to the target output, but also to multiple other outputs. Hence, we check the logic values of all following outputs as well. Additionally, an output may incidentally show differences in both directions and hence unateness can be ruled out without additional SAT calls.

The described SAT instances are very similar to detecting a stuck-at-1 fault at the input x. Hence, we employ encoding based speed-up techniques that are known from solving such ATPG problems. By adding D-chains [31] that add redundant information to the instance, the SAT solver can propagate the differences more easily. Additionally, we tuned the SAT solver's internal settings towards the characteristics of the circuit-based SAT instances which are dominated by a large number of rather simple SAT calls. For instance, we do not use preprocessing techniques on the SAT instances.

4 Sequential Functional Dependency

While the prior definitions and algorithms are specified for combinational circuits, we also investigate the definition of dependency in sequential circuits.

To translate the properties from Sect. 2.1 to sequential circuits, we use a similar approach as used in (un)bounded model checking: An output y_j is called *sequential functionally dependent* in an input x_i if and only if there exists a number $k \in \mathbb{N}$, such that $f_j^{(k)}$ is functionally dependent in x_i, where $f_j^{(k)}$ represents the Boolean function of the output modelled over k steps.

For $f_j^{(1)}$ the sequential circuit can be treated as a combinational one. For $f_j^{(k)}$ with $k > 1$, the definition of sequential dependence follows the combinational one if the sequential circuit is considered as an iterative logic array with an unlimited number of time frames. Hence, such a definition allows to extend combinational dependency in a natural way to sequential dependency.

In contrast to the complexity of the combinational dependency computation of a single input-output pair (which is NP-complete since it is an ATPG problem), sequential dependency computation is identical to invariant checking which can be expressed by an unbounded model checking approach and is PSPACE-complete.

[1] We also performed a structural check for each input-output pair if there potentially exists an inverting path between them. If this is not the case, the additionally SAT-call to check for unateness may be skipped. However, the performance impact was insignificant and hence we did not employ this optimization.

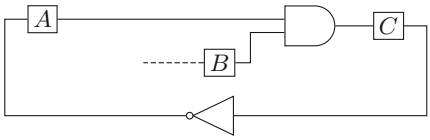

Fig. 3. Example circuit

Sequential independence is defined as the contrary of sequential dependence. An output y_j is called (strictly) *sequential positive/negative unate* in x_i, if there exists a k_0, such that for every number $k \in \mathbb{N}$ with $k > k_0$, $f_j^{(k)}$ is (strictly) positive/negative unate in x_i.

Example 3. Let $f^{(k)}$ be the Boolean function corresponding to the flip flop C in the circuit in Fig. 3 in the k^{th} step. Then C is alternating strictly positive and negative unate in A. Thus, C is neither sequential positive nor negative unate in A. However, C is sequential dependent in A.

4.1 Approximative Algorithm

We use the methods from Sect. 3 to compute the combinational dependency matrix $D(f)$ for $f^{(0)}$ and then initialize the sequential dependency Matrix $D^s(f)$ as $D(f)$. For clarity, to refer to an entry of the dependency matrix with output y_j and input x_i we write d_{y_j,x_i} instead of $d_{j,i}$. Respective entries of the sequential dependency matrix are denoted as $d^s_{j,i}$.

For each output $y = f_j$ and input x we check, if there exist x_k and y_l, such that

- x_k and y_l correspond to the same flip flop φ,
- $d^s_{y,x_k} \neq \bullet$ and
- $d^s_{y_l,x} \neq \bullet$.

The path-dependence of y in x over φ is defined with the equation

$$\mathrm{pd}_\varphi(y,x) = \begin{cases} p & \text{if } d_{y,x_k}, d_{y_l,x} \text{ unate and } d_{y,x_k} = d_{y_l,x}, \\ n & \text{if } d_{y,x_k}, d_{y_l,x} \text{ unate and } d_{y,x_k} \neq d_{y_l,x}, \\ \mathtt{d} & \text{otherwise.} \end{cases} \tag{7}$$

If $\mathrm{pd}_\varphi(y,x) \neq d^s_{y,x}$, we may need to update the dependence value of the sequential dependency matrix $d^s_{y,x}$:

$$d^s_{y,x} \leftarrow \begin{cases} \mathrm{pd}_\varphi(y,x) & \text{if } \mathrm{pd}_\varphi(y,x) = d^s_{y,x} \ \vee \ d^s_{y,x} = \bullet, \\ \mathtt{d} & \text{otherwise.} \end{cases} \tag{8}$$

Now we choose different y_l, x_k, y and/or x and start from the top until we reach a fixed point. Our algorithm focuses on positive unateness (p) and negative unateness (n), in contrast to strict positive unateness ($\mathtt{p}$) and strict negative unateness ($\mathtt{n}$).

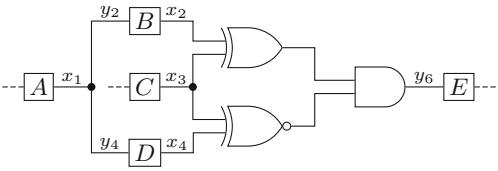

Fig. 4. Example circuit (2)

According to the definitions in the previous section, all dependencies marked as seq. positive unate (p), seq. negative unate (n) or seq. independent ($\bullet$) by our approximation are correctly classified as we will show in Sect. 4.2.

However, the dependencies marked as seq. functionally dependent (d) may be inaccurate as $d^s_{j,i} = \mathtt{d}$ is an over-approximation. Hence, the algorithm allows an accurate classification for three dependency conditions, while avoiding the computational complexity of a completely accurate algorithm (that still can be applied if completeness is needed).

To see that $d^s_{j,i} = \mathtt{d}$ does not generally imply sequential dependence, see Fig. 4, where $d^s_{6,2} = \mathtt{d}$, $d^s_{2,1} = p$, $d^s_{6,4} = \mathtt{d}$ and $d^s_{4,1} = p$. Therefore $d^s_{6,1} = \mathtt{d}$, but because of the partly inverse reconvergence, y_6 is sequentially independent in x_1.

If the XNOR-Gate in Fig. 4 was replaced by an XOR-Gate, y_6 would be sequentially dependent in x_1, while no values of its combinational dependency matrix would differ from the combinational dependency matrix of the original circuit. Since these two circuits have the same combinational dependency matrix, but different sequential dependency matrices, it is not possible to build an exact algorithm for sequential dependency, solely based on the combinational dependency matrix.

4.2 Proof of Correctness for p, n and $\bullet$

The correctness of the classification of an input-output pair as either p, n or $\bullet$ can be shown as follows:

p: Proof by contradiction: For the correctness of the return value p, let the algorithm return p for output (or flip flop) y and input (or flip flop) x, but y is not sequential positive unate in x. Then there exists an (arbitrary high) $k \in \mathbb{N}$, such that $f^{(k)}$, the Boolean function of y, is not positive unate in x. Following from the definition of unateness (cf. Sect. 2.1), there exists an input sequence $\hat{x}$, such that $f^{(k)}_{\overline{x}}(\hat{x}) = 1$ and $f^{(k)}_{x}(\hat{x}) = 0$. For clarity, we use the abbreviations $x^{[\overline{x}]} = 0$, $y^{[\overline{x}]} = 1$ and $x^{[x]} = 1$, $y^{[x]} = 0$ where $[\overline{x}]$ and $[x]$ indicate the logic value for the respective case. There must exist a path from x to y, where the path follows $x = p_0, p_1, \ldots, p_{m-1}, p_m = y$, all p_i with $0 < i < m$ represent flip flops and $\forall i \leq m : p_i^{[\overline{x}]} \neq p_i^{[x]}$.

For any $i < m$, p_{i+1} combinationally depends on p_i, therefore the entry in the combinational dependency matrix for p_{i+1} on p_i (d_{p_{i+1},p_i}) is not $\bullet$, thus

$\mathtt{d}$, p or n. As seen in Eq. 8, no dependency value gets overwritten by $\bullet$, which leads to $d^s_{p_{i+1},p_i} \in \{\mathtt{d}, n, p\}$ for all i. If $d^s_{p_{i+1},p_i}$ in any calculation step was $\mathtt{d}$, $d^s_{p_{i+1},p_i}$ would be $\mathtt{d}$ in the sequential dependency matrix, as $\mathtt{d}$ can not get overwritten. Then, by Eq. 7, $\mathrm{pd}_\varphi(x, y)$ would be step-wise calculated as $\mathtt{d}$, which would result in $d^s_{y,x} = \mathtt{d}$ in contradiction to the algorithm returning p. Thus, for any $i < m$, it holds that $d^s_{p_{i+1},p_i} \in \{n, p\}$.

Let $I_{\text{Same}} = \{i < m : p_i^{[\overline{x}]} = p_{i+1}^{[\overline{x}]}\}$ and $I_{\text{Diff}} = \{i < m : p_i^{[\overline{x}]} \neq p_{i+1}^{[\overline{x}]}\}$, then I_{Diff} contains an odd number of elements, because $p_0^{[\overline{x}]} \neq p_m^{[\overline{x}]}$. For any $i \in I_{\text{Same}}$, it holds that $d^s_{p_{i+1},p_i} \neq n$ resp. $d^s_{p_{i+1},p_i} = p$ in every calculation step and similarly for any $i \in I_{\text{Diff}}$, always $d^s_{p_{i+1},p_i} = n$. The calculated dependency pd for the given path along $p_0, \ldots, p_m$ will then be calculated based on an odd number of n and otherwise only p, which will by Eq. 7 result in path dependence n. Therefore, by Eq. 8, the algorithm does not return p, a contradiction.

n: The proof of the correctness of the return value n is analogous to the proof of the correctness of p. The major difference is that I_{Diff} contains an even number of elements. This will force a path calculation to result in p, making impossible, that the algorithm returns n.

$\bullet$: Proof by contradiction: For the correctness of the return value $\bullet$, let the algorithm return $\bullet$ for output y and input x, but y is not sequential independent in x, i.e. sequential dependent in x. Following from a similar argument as for p, there must exist a path, which follows $x = p_0, p_1, \ldots, p_{m-1}, p_m = y$, all p_i with $0 < i < m$ represent flip flops and $\forall i \leq m : p_i^{[\overline{x}]} \neq p_i^{[x]}$. By Eq. 8, every $d^s_{p_{i+1},p_i}$ in any calculation step is not $\bullet$. Then $\mathrm{pd}_\varphi(x, y)$ would by Eq. 7 be step-wise calculated not as $\bullet$, which would by Eq. 8 result in $d^s_{p_{i+1},p_i} \neq \bullet$ in the sequential dependency matrix, in contradiction to the algorithm returning $\bullet$. □

5 Experimental Results

We implemented the proposed approach in C++ on top of the ATPG framework PHAETON [26] and the SAT solver *antom* [28]. All experiments were carried out on a single core of an Intel Xeon machine running at 3.3 GHz, 64 GB main memory running Linux 3.13. For the evaluations, we used combinational arithmetic benchmarks from EPFL[2] as well as sequential benchmarks from the ITC'99 [7] benchmark suite and industrial benchmarks provided by NXP (starting with 'b' and 'p' followed by a number, respectively). Finally, we applied the method to the OpenCore benchmarks from the IWLS'05 [2] family. In order to keep the section compact, we skipped the benchmarks that had either negligible runtime or that could not be solved within a timeout of 2 h.

[2] lsi.epfl.ch/benchmarks.

Table 1. Combinational experiments

Circuit	IO	Dependencies				Statistics			Runtimes	
		Struct.	Func.	Pos.	Neg.	Incidental	Instances	Solves	Unateness	Total
adder	256129	0	16512	256	0	16512	256	17024	4.07	4.40
bar	135128	0	896	16384	0	777	135	33670	1.08	17.70
divisor	128128	0	12220	66	2082	13026	128	9519	4502.32	4871.53
log2	3232	0	1022	0	2	979	32	686	3746.98	3750.90
max	512130	32512	32512	1024	512	15271	512	82241	37.13	110.24
sin	2425	0	577	22	0	441	24	472	17.86	31.23
square	64128	0	6041	68	3	5956	64	3926	447.34	450.51
b14	277299	11	21803	705	68	18403	277	22368	20.74	27.71
b15	485519	19504	40704	3338	292	33017	485	57931	77.33	231.29
b17	14511511	44054	135906	11271	1225	109432	1451	177747	234.55	637.20
b18	33073293	1084	331223	22367	2250	266523	3307	326342	439.45	690.52
b20	522512	5298	51508	1212	370	42795	522	52742	120.06	232.61
b21	522512	5310	51508	1238	136	42171	522	53808	94.69	203.93
b22	735725	5313	78254	1740	359	65388	735	76254	163.15	280.46
p35k_s	28612229	0	140676	10802	10697	123321	2861	147727	628.73	1090.66
p45k_s	37392550	409	24910	13638	860	14256	3739	62307	20.84	59.95
p78k_s	31483484	377	52338	6032	0	48970	3148	50197	47.34	73.73
p81k_s	40293952	1380	387839	11724	18737	324849	4029	308638	419.38	897.82
p100k_s	55575489	756	77829	24347	3406	51415	5557	147954	4959.58	5236.16
des_area	367192	0	11328	288	0	9756	367	5623	2.78	6.16
spi	272273	16256	4205	982	117	1977	272	24649	1.54	10.03
systemcdes	312255	592	2341	590	8	1580	312	4222	0.61	1.57
wb_dma	747748	1195	10880	2364	842	6674	747	19879	2.25	4.54
tv80	372391	9452	15265	1917	218	11906	372	28468	9.71	21.49
systemcaes	928799	14613	17158	939	44	13273	928	29743	7.43	19.21
ac97_ctrl	22532247	10	7940	7083	831	5315	2253	25510	0.87	2.67
pci_bridge32	35173559	9906	59360	12179	2512	38839	3517	108600	57.35	106.20
aes_core	788659	0	7290	541	29	6285	788	5636	2.17	4.43
wb_conmax	18992186	1976	46132	23968	16346	37884	1899	125420	39.40	98.11
des_perf	90418872	0	29344	7448	0	18627	9041	51718	16.63	68.60

All times are in seconds.

5.1 Combinational Dependency

Table 1 lists the results of the evaluation. The first three columns list the name of the circuit as well as the number of inputs and outputs. The following four columns list the identified dependencies. The number of only structural dependencies (that are found to be independent) are given first followed by the number of dependent classifications (excluding unateness) and finally the number of positive or negative unate classifications, respectively. The next three columns list statistics of the proposed SAT-based approach: The number of functional dependencies that were found incidentally followed by the number of generated SAT instances and calls to the SAT solver. The final two columns list the runtime for unateness checking and the total runtime in seconds.

As can be seen, our approach is able to completely compute the dependency matrix on a wide range of mid-sized circuits taken from various academic and industrial benchmark circuits within a maximum computation time of 2 h (7200 s).

Interestingly, the number of input-output pairs that are positive unate are roughly an order of magnitude higher than those that are negative unate. This is most prominent for the barrelshifter circuit 'bar' from the EPFL benchmarks that contains mostly positive unate pairs but no negative one.

The effect of the optimizations described in Sect. 3.3 can be witnessed by the high number of dependencies identified incidentally as well the high ratio between the number of instances as well as the calls to the SAT solvers. Hence, these methods effectively keep the runtimes in check.

5.2 Comparison to Existing Approach

We compared our approach to the BDD-based implementation in ABC [4] where identical circuit definitions readable for both tools were available. We listed the results in Table 2.

The proposed SAT-based approach shows superior performance for the rather complex benchmark sets of the EPFL as well as the ITC'99 benchmarks where the approach does not suffer from excessive memory usage. For complex functions, the BDD-based approach did not terminate due to insufficient memory requirements.

For the EPFL benchmarks, the BDD-based approach did not terminate due to a timeout which we set to 7200 s. 7 of the 10 arithmetic EPFL benchmarks can be solved using the SAT-based approach, and for 6 of them the SAT-based

Table 2. Comparison to the BDD-based approach from ABC [4]

		Runtimes		ABC [4]	
Circuit	In/Out	Unate.	Total	Unate.	Total
adder	256/129	4.07	4.40	0.01	0.54
bar	135/128	1.08	17.70	18.96	19.05
divisor	128/128	4502.32	4871.53	TO	TO
log2	32/32	3746.98	3750.90	TO	TO
max	512/130	37.13	110.24	TO	TO
sin	24/25	17.86	31.23	0.15	866.99
square	64/128	447.34	450.51	TO	TO
b14	277/299	20.74	27.71	74.17	120.07
b15	485/519	77.33	231.29	199.45	368.25
b17	1451/1511	234.55	637.20	MO	MO
b18	3307/3293	439.45	690.52	MO	MO
b20	522/512	120.06	232.61	MO	MO
b21	522/512	94.69	203.93	MO	MO
b22	735/725	163.15	280.46	MO	MO

All times are in seconds; MO: memory out; TO: timeout ($\geq$7200 s).

approach found the solution faster. The three remaining benchmarks cannot be solved within 7200 s by both approaches. It is worth noting that for benchmarks that are rather small or structurally simple (such as the adder) the BDD-based approach performs faster than the SAT-based approach.

5.3 Sequential Dependency

Table 3 shows the results of the sequential dependency computation algorithm as presented in Sect. 4 that was executed on the sequential versions of the benchmark circuits from the previous experiment where possible. At first, the name of the circuit, the number of flip flops as well as the number of inputs and outputs are given. Following, as in the previous section we list the different dependencies as well as the number of iterations through the combinational dependency matrix. Finally, the runtimes for the generation of the combinational dependency matrix, the extension to the sequential matrix as well as the total runtime (all in seconds) are given.

As can be seen, the sequential algorithm needs only a few iterations to conclude the sequential dependency for all benchmarks. Hence, the overall impact on the runtime is limited and for most of the circuits less than the runtime of the combinational method. When comparing the results of the dependencies, one

Table 3. Sequential experiments

Circuit	FFs	IO	Sequential dependencies					Runtime SAT-based		
			Struct.	Func.	Pos.	Neg.	Iterations	Comb.	Sequential	Total
b14	245	3254	62	60702	108	1	3	27.71	0.06	27.77
b15	449	3670	18112	161327	282	4	3	231.29	0.40	231.69
b17	1414	3797	3020	1680385	217	0	3	637.20	17.27	654.47
b18	3270	3723	0	10575260	87	2	3	690.52	362.10	1052.62
b20	490	3222	0	246240	38	2	3	232.61	0.85	233.46
b21	490	3222	0	246240	38	2	2	203.93	0.57	204.50
b22	703	3222	0	487257	59	3	3	280.46	5.21	285.68
p35k_s	2173	68856	0	243835	10786	10712	3	1090.66	2.39	1093.05
p45k_s	2331	1408219	90	1826005	1093874	3253	5	59.95	208.98	268.94
p78k_s	2977	171507	0	616075	26788	0	5	73.73	28.76	102.50
p81k_s	3877	15275	1566	2974664	6693	8024	3	897.82	85.70	983.51
p100k_s	5395	16294	92	4797360	1091341	5894	5	5236.16	301.64	5537.80
des_area	128	23964	0	70464	0	0	3	6.16	0.03	6.19
spi	229	4344	27456	21010	1050	11	3	10.03	0.05	10.08
systemcdes	190	12265	0	47603	3202	2	4	1.57	0.04	1.61
wb_dma	533	214215	0	128209	1268	79	4	4.54	0.89	5.43
tv80	359	1332	0	132696	59	2	3	21.49	0.23	21.72
systemcaes	670	258129	0	716373	1	0	3	19.21	2.32	21.53
ac97_ctrl	2199	5448	5	132290	155065	658	3	2.67	7.11	9.79
pci_bridge32	3358	159201	20030	3825065	12332	155	3	106.20	138.70	244.90
aes_core	530	258129	0	285777	390	4	3	4.43	2.00	6.43
wb_conmax	770	11291416	0	647425	13152	512	3	98.11	5.39	103.50
des_perf	8808	23364	0	13852506	116088	0	3	68.60	1129.18	1197.78

can note that the number of functional dependencies increases at the cost of the other classifications. This is expected as many structural dependencies get functional when considering multiple timeframes. Additionally, the requirements for sequential positive as well as sequential negative unateness are much harder to meet than their combinational counterparts and hence such classifications tend to be changed to a functional dependency.

5.4 Application to Reverse Engineering

We show the applicability of functional dependency and unateness information in a small case study of reverse engineering. We consider the *Permutation-Independent Subset Equivalence Checking* (SPIEC) problem [29]: Given a block $f_b : \mathbb{B}^n \to \mathbb{B}^m$ and a component $f_c : \mathbb{B}^r \to \mathbb{B}^s$ with $n \geq r$ and $m \geq s$, SPIEC asks whether there exists a mapping from all primary inputs and primary outputs of f_c to primary inputs and primary outputs in f_b such that the block realizes the same function as the component w.r.t. this mapping.

The algorithm presented in [29] solves this problem by finding subgraph isomorphisms of simulation graphs for the block and the component. A simulation graph has input vertices, output vertices, and vertices for some characteristic simulation vectors. A subgraph isomorphism in these graphs provides a candidate mapping that can be verified using combinational equivalence checking [22]. Subgraph isomorphism is translated into a constraint satisfaction problem according to [30] while additionally considering application-specific information extracted from the circuits, e.g., functional dependency and unateness properties.

The constraint satisfaction implementation starts by creating a domain for each vertex in the component's simulation graph. The domain is a set of possible candidate vertices in the block's simulation graph. Filtering methods then try to reduce the size of the domains such that eventually either (i) some domain is empty and therefore no matching exists, or (ii) all domains contain a single element from which the mapping can directly be extracted. If the filtering techniques cannot advance to any of these two cases, one has to start branching using a backtracking algorithm. The aim is to avoid backtracking, which can be achieved by effective filtering methods.

In our experiment we considered the impact of the dependency matrix by comparing three different scenarios: (i) no information is provided, (ii) the dependency matrix is provided for the component which allows the use of structural dependency information as a signature, and (iii) the dependency matrix is provided for both the block and the component allowing the use of functional dependency and unateness properties as signatures for filtering. We measure the quality by comparing the accumulated domain sizes after all filtering methods are exhausted right before backtracking is initiated.

Table 4 shows the results of our experiments. The circuits for blocks (c1–c10) and components (adder, multi, shift,[3] and subtract) are the same that were

[3] In [29] shift-left and shift-right are considered separately. Since these operations are equivalent under permutation, the measured numbers in the experiment also do not differ.

Table 4. Reverse engineering experiment

	adder	multi	shift	subtract
c1-8	0/0/0	730/661/609	44/44/44	0/0/0
c2-8	890/770/482	1753/706/612	1217/595/578	860/639/428
c3-8	489/24/24	797/455/425	577/256/240	0/0/0
c4-8	712/0/0	1280/0/0	1024/421/421	26/26/26
c5-8	690/462/24	1405/423/401	1089/44/44	26/26/26
c6-8	1234/1140/273	1820/0/0	1600/930/930	719/989/422
c7-8	141/25/25	796/0/0	576/401/401	27/27/27
c8-8	368/0/0	576/0/0	427/44/44	0/0/0
c9-8	1291/984/24	1885/566/476	1665/645/636	951/881/388
c10-8	131/24/24	1596/0/0	456/253/253	0/0/0

evaluated in [29] in their 8-bit variant. Each cell in the matrix represents the application of SPIEC to the block and component of the respective row and column, respectively. Each cell shows three numbers. These numbers are the accumulated domain sizes of primary inputs and outputs for each of the three considered scenarios. The cell is shaded gray if the component is contained in the block. As can be seen in the table, the dependency matrix has a strong influence on the results since the domain sizes can be significantly reduced, often resulting in a matching that provides a solution. For example, in the case of c9 and the adder a mapping has been found only if the dependency matrices for both the block and the component are provided. In the case of c4 and the adder one needs to compute at least the component's dependency matrix to conclude that it is not contained in that block without backtracking.

6 Conclusions

We presented a SAT-based algorithm to compute functional dependence properties of combinational as well as sequential Boolean functions. We inspect which outputs in a multi-output function are functionally dependent on which inputs. Furthermore, the algorithms checks whether the input-output pair is unate if it is dependent, which is a stronger property. Furthermore, incremental encoding techniques known from ATPG problems are employed to speed up the algorithm. Additionally, we extended the classical dependency classifications to sequential circuits and presented an iterative approximative algorithm to compute such sequential dependencies.

In extensive experimental studies on different benchmarks suites we detailed the robustness of the algorithms especially for hard combinational as well as sequential benchmarks. Additionally, our methods show better performance compared to previously presented BDD-based approaches with which many of the instances cannot be solved due to memory limitations or timeouts.

Acknowledgments. The authors wish to thank Robert Brayton and Alan Mishchenko for many helpful discussions. This research was partially financed by H2020-ERC-2014-ADG 669354 CyberCare and the Baden-Württemberg Stiftung gGmbH Stuttgart within the scope of its IT security research programme.

References

1. Biere, A., Cimatti, A., Clarke, E.M., Fujita, M., Zhu, Y.: Symbolic model checking using SAT procedures instead of BDDs. In: Design Automation Conference, pp. 317–320 (1999)
2. Albrecht, C.: IWLS 2005 benchmarks. In: International Workshop for Logic Synthesis (IWLS) (2005). http://www.iwls.org
3. Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press, Amsterdam (2009)
4. Brayton, R., Mishchenko, A.: ABC: an academic industrial-strength verification tool. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 24–40. Springer, Heidelberg (2010). doi:10.1007/978-3-642-14295-6_5
5. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Comput. **35**(8), 677–691 (1986)
6. Cook, S.A.: The complexity of theorem-proving procedures. In: Symposium on Theory of Computing, pp. 151–158 (1971)
7. Corno, F., Reorda, M., Squillero, G.: RT-level ITC'99 benchmarks and first ATPG results. IEEE Des. Test Comput. **17**(3), 44–53 (2000)
8. Saab, D.G., Abraham, J.A., Vedula, V.M.: Formal verification using bounded model checking: SAT versus sequential ATPG engines. In: VLSI Design, pp. 243–248 (2003)
9. Een, N., Mishchenko, A., Sörensson, N.: Applying logic synthesis for speeding up SAT. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 272–286. Springer, Heidelberg (2007). doi:10.1007/978-3-540-72788-0_26
10. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004). doi:10.1007/978-3-540-24605-3_37
11. van Eijk, C.A.J., Jess, J.A.G.: Exploiting functional dependencies in finite state machine verification. In: European Design and Test Conference, pp. 9–14 (1996)
12. Jiang, J.-H.R., Brayton, R.K.: Functional dependency for verification reduction. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 268–280. Springer, Heidelberg (2004). doi:10.1007/978-3-540-27813-9_21
13. Jiang, J.R., Lee, C., Mishchenko, A., Huang, C.: To SAT or not to SAT: scalable exploration of functional dependency. IEEE Trans. Comput. **59**(4), 457–467 (2010)
14. Katebi, H., Markov, I.L.: Large-scale Boolean matching. In: Design, Automation and Test in Europe, pp. 771–776 (2010)
15. Larrabee, T.: Test pattern generation using Boolean satisfiability. IEEE Trans. CAD Integr. Circuits Syst. **11**(1), 4–15 (1992)
16. Lee, C., Jiang, J.R., Huang, C., Mishchenko, A.: Scalable exploration of functional dependency by interpolation and incremental SAT solving. In: International Conference on Computer-Aided Design, pp. 227–233 (2007)
17. Levin, L.A.: Universal sequential search problems. Probl. Inf. Transm. **9**(3), 115–116 (1973)

18. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Hunt, W.A., Johnson, S.D. (eds.) FMCAD 2000. LNCS, vol. 1954, pp. 127–144. Springer, Heidelberg (2000). doi:10.1007/3-540-40922-X_8
19. Marhöfer, M.: An approach to modular test generation based on the transparency of modules. In: IEEE CompEuro 1987, pp. 403–406 (1987)
20. McMillan, K.L.: Interpolation and SAT-based model checking. In: Hunt, W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003). doi:10.1007/978-3-540-45069-6_1
21. McNaughton, R.: Unate truth functions. IRE Trans. Electron. Comput. **10**(1), 1–6 (1961)
22. Mishchenko, A., Chatterjee, S., Brayton, R.K., Eén, N.: Improvements to combinational equivalence checking. In: International Conference on Computer-Aided Design, pp. 836–843 (2006)
23. Mohnke, J., Molitor, P., Malik, S.: Limits of using signatures for permutation independent Boolean comparison. Form. Methods Syst. Des. **21**(2), 167–191 (2002)
24. Murray, B.T., Hayes, J.P.: Test propagation through modules and circuits. In: International Test Conference, pp. 748–757 (1991)
25. Reimer, S., Sauer, M., Schubert, T., Becker, B.: Using MaxBMC for pareto-optimal circuit initialization. In: Conference on Design, Automation and Test in Europe, pp. 1–6, March 2014
26. Sauer, M., Becker, B., Polian, I.: PHAETON: a SAT-based framework for timing-aware path sensitization. IEEE Trans. Comput. **PP**(99), 1 (2015)
27. Sauer, M., Reimer, S., Polian, I., Schubert, T., Becker, B.: Provably optimal test cube generation using quantified Boolean formula solving. In: ASP Design Automation Conference, pp. 533–539 (2013)
28. Schubert, T., Reimer, S.: antom (2013). https://projects.informatik.uni-freiburg.de/projects/antom
29. Soeken, M., Sterin, B., Drechsler, R., Brayton, R.K.: Reverse engineering with simulation graphs. In: Formal Methods in Computer-Aided Design, pp. 152–159 (2015)
30. Solnon, C.: AllDifferent-based filtering for subgraph isomorphism. Artif. Intell. **174**(12–13), 850–864 (2010)
31. Stephan, P., Brayton, R.K., Sangiovanni-Vincentelli, A.L.: Combinational test generation using satisfiability. IEEE Trans. CAD Integr. Circuits Syst. **15**(9), 1167–1176 (1996)
32. Tseytin, G.: On the complexity of derivation in propositional calculus. In: Studies in Constructive Mathematics and Mathematical Logic (1968)