Complete Test Sets And Their Approximations

Eugene Goldberg
eu.goldberg@gmail.com

***Abstract*—We use testing to check if a combinational circuit N always evaluates to 0 (written as $N \equiv 0$). We call a set of tests proving $N \equiv 0$ a complete test set (CTS). The conventional point of view is that to prove $N \equiv 0$ one has to generate a *trivial* CTS. It consists of all $2^{|X|}$ input assignments where X is the set of input variables of N. We use the notion of a Stable Set of Assignments (SSA) to show that one can build a *non-trivial* CTS consisting of less than $2^{|X|}$ tests. Given an unsatisfiable CNF formula $H(W)$, an SSA of H is a set of assignments to W that proves unsatisfiability of H. A trivial SSA is the set of all $2^{|W|}$ assignments to W. Importantly, real-life formulas can have non-trivial SSAs that are much smaller than $2^{|W|}$. In general, construction of even non-trivial CTSs is inefficient. We describe a much more efficient approach where tests are extracted from an SSA built for a projection of N on a *subset* of its variables. These tests can be viewed as an approximation of a CTS for N. We describe potential applications of our approach. We show experimentally that it can be used to facilitate hitting corner cases and expose bugs in sequential circuits overlooked due to checking "misdefined" properties.**

I. Introduction

Testing is an important part of verification flows. For that reason, any progress in understanding testing and improving its quality is of great importance. In this paper, we consider the following problem. Given a single-output combinational circuit N, find a set of input assignments (tests) proving that N evaluates to 0 for every test (written as $N \equiv 0$) or find a counterexample. We will call a set of input assignments proving $N \equiv 0$ a *complete test set* (*CTS*)[1]. We will call the set of all possible tests a *trivial CTS*. Typically, one assumes that proving $N \equiv 0$ involves derivation of the trivial CTS, which is infeasible in practice. Thus, testing is used only for finding an input assignment refuting $N \equiv 0$. We present an approach for building a non-trivial CTS consisting only of a subset of all possible tests[2]. In general, finding even a non-trivial CTS for a large circuit is impractical. We describe a much more efficient approach where an *approximation* of a CTS is generated.

The circuit N above usually describes a property ξ of a multi-output combinational circuit M, the latter being the *real object of testing*. For instance, ξ may state that M never produces some output assignments. To differentiate CTSs and their approximations from conventional test sets verifying M "as a whole", we will refer to the former as *property-checking test sets*. Let $\Xi := \{\xi_1, \dots, \xi_k\}$ be the set of properties of M formulated by a designer. Assume that every property of Ξ holds and T_i is a test set generated to check property $\xi_i \in \Xi$. There are at least two reasons why applying T_i to M makes sense. First, if Ξ is *incomplete*[3], a test of T_i can expose a bug breaking a property of M that is not in Ξ. Second, if property ξ_i is defined *incorrectly*, a test of T_i may expose a bug breaking the correct version of ξ_i. On the other hand, if M produces proper output assignments for all tests of $T_1 \cup \dots \cup T_k$, one gets extra guarantee that M is correct. In Section VI, we list some other applications of property-checking test sets such as increasing the probability of hitting corner cases and testing properties of sequential circuits.

Let $N(X, Y, z)$ be a single-output combinational circuit where X and Y specify the sets of input and internal variables of N respectively and z specifies the output variable of N. Let $F_N(X, Y, z)$ be a formula defining the functionality of N (see Section III). We will denote the set of variables of circuit N (respectively formula H) as $\mathit{Vars}(N)$ (respectively $\mathit{Vars}(H)$). Every assignment[4] to $\mathit{Vars}(F_N)$ satisfying F_N corresponds to a consistent assignment[5] to $\mathit{Vars}(N)$ and vice versa. Then the problem of proving $N \equiv 0$ reduces to showing that formula $F_N \wedge z$ is unsatisfiable. From now on, we assume that all formulas mentioned in this paper are *propositional*. Besides, we will assume that every formula is represented in CNF i.e. as a conjunction of disjunctions of literals.

Our approach is based on the notion of a Stable Set of Assignments (SSA) introduced in [9]. Given formula $H(W)$, an SSA of H is a set P of assignments to variables of W that have two properties. First, every assignment of P falsifies H. Second, P is a transitive closure of some neighborhood relation between assignments (see Section II). The fact that H has an SSA means that the former is unsatisfiable. Otherwise, an assignment satisfying H is generated when building its SSA. If H is unsatisfiable, the set of all $2^{|W|}$ assignments is always an SSA of H . We will refer to it as *trivial*. Importantly, a real-life formula H can have a lot of SSAs whose size is much less than $2^{|W|}$. We will refer to them as *non-trivial*. As we show in Section II, the fact that P is an SSA of H is a *structural* property of the latter. That is this property cannot be expressed in terms of the truth table of H (as opposed to a *semantic* property of H). For that reason, if P is an SSA

[1]Term CTS is sometimes used to say that a test set invokes every event specified by a *coverage metric*. Our application of this term is quite different.

[2]In the case of black-box testing, i.e. when only *the number of input variables* of N is known, to prove $N \equiv 0$ one indeed has to enumerate all possible input assignments. In this paper, we consider white-box testing.

[3]That is M can be incorrect even if all properties of Ξ hold.

[4]By an assignment to a set of variables V, we mean a *full* assignment where every variable of V is assigned a value.

[5]An assignment to a gate G of N is called consistent if the value assigned to the output variable of G is implied by values assigned to its input variables. An assignment to variables of N is called consistent if it is consistent for every gate of N.

for H, it may not be an SSA for another formula H' logically equivalent to H. So, a structural property is *formula-specific*.

We show that a CTS for N can be easily extracted from an SSA of formula $F_N \wedge z$. This makes a non-trivial CTS a structural property of circuit N that cannot be expressed in terms of its truth table. Building an SSA for a large formula is inefficient. So, we present a procedure constructing a simpler formula $H(V)$ implied by $F_N \wedge z$ (where $V \subset Vars(F_N \wedge z)$) and building an SSA of H. The existence of such an SSA means that H (and hence $F_N \wedge z$) is unsatisfiable. So, $N \equiv 0$ holds. Formula H is obtained from $F_N \wedge z$ by a resolution-based procedure where *no resolutions* on variables of V are allowed. So H *preserves* some structure of $F_N \wedge z$. A test set extracted from an SSA of H can be viewed as a way to verify a "projection" of N on variables of V. On the other hand, one can consider this set as an approximation of a CTS for N. We will refer to the procedure above as $SeSt$ ("Se-mantics and St-ructure"). $SeSt$ combines semantic and structural derivations, hence the name. The semantic part of $SeSt$ is[6] to derive H. Its structural part consists of constructing an SSA of H thus proving H unsatisfiable.

The contribution of this paper is as follows. First, we introduce the notion of non-trivial CTSs (Section III). Second, we present a method for efficient construction of property-checking tests that are approximations of CTSs (Sections IV and V). Third, we describe applications of such tests (Section VI). Fourth, we experimentally show the efficiency and effectiveness of property-checking tests (Section VII).

II. Stable Set Of Assignments

A. Definitions

We will refer to a disjunction of literals as a *clause*. Let $\vec{p}$ be an assignment to a set of variables V. Let $\vec{p}$ falsify a clause C. Denote by $\boldsymbol{Nbhd(\vec{p},C)}$ the set of assignments to V satisfying C that are at Hamming distance 1 from $\vec{p}$. (Here *Nbhd* stands for "Neighborhood"). Thus, the number of assignments in $Nbhd(\vec{p},C)$ is equal to that of literals in C. Let $\vec{q}$ be another assignment to V (that may be equal to $\vec{p}$). Denote by $\boldsymbol{Nbhd(\vec{q},\vec{p},C)}$ the subset of $Nbhd(\vec{p},C)$ consisting only of assignments that are farther from $\vec{q}$ than $\vec{p}$ is (in terms of the Hamming distance).

Example 1: Let $V = \{v_1,v_2,v_3,v_4\}$ and $\vec{p}$=0110. We assume that the values are listed in $\vec{p}$ in the order the corresponding variables are numbered i.e. $v_1 = 0$, $v_2 = 1, v_3 = 1, v_4 = 0$. Let $C = v_1 \vee \overline{v_3}$. (Note that $\vec{p}$ falsifies C.) Then $Nbhd(\vec{p},C)$=$\{\vec{p}_1,\vec{p}_2\}$ where $\vec{p}_1$ = 1110 and $\vec{p}_2$=0100. Let $\vec{q}$ = 0000. Note that $\vec{p}_2$ is closer to $\vec{q}$ than $\vec{p}$ is. So $Nbhd(\vec{q},\vec{p},C)$=$\{\vec{p}_1\}$.

Definition 1: Let H be a formula[7] specified by a set of clauses $\{C_1,\dots,C_k\}$. Let $P = \{\vec{p}_1,\dots,\vec{p}_m\}$ be a set of assignments to $Vars(H)$ such that every $\vec{p}_i \in P$ falsifies H. Let Φ denote a mapping $P \rightarrow H$ where $\Phi(\vec{p}_i)$ is a clause C of H falsified by $\vec{p}_i$. We will call Φ an **AC-mapping** where "AC" stands for "Assignment-to-Clause".

Definition 2: Let H be a formula specified by a set of clauses $\{C_1,\dots,C_k\}$. Let $P = \{\vec{p}_1,\dots,\vec{p}_m\}$ be a set of assignments to $Vars(H)$. P is called a **Stable Set of Assignments**[8] (**SSA**) of H with **center** $\vec{p}_{init} \in P$ if there is an AC-mapping Φ such that for every $\vec{p}_i \in P$, $Nbhd(\vec{p}_{init},\vec{p}_i,C) \subseteq P$ holds where $C = \Phi(\vec{p}_i)$.

Example 2: Let H consist of four clauses: $C_1 = v_1 \vee v_2 \vee v_3$, $C_2 = \overline{v}_1$, $C_3 = \overline{v}_2$, $C_4 = \overline{v}_3$. Let $P = \{\vec{p}_1,\vec{p}_2,\vec{p}_3,\vec{p}_4\}$ where $\vec{p}_1 = 000$, $\vec{p}_2 = 100$, $\vec{p}_3 = 010$, $\vec{p}_4 = 001$. Let Φ be an AC-mapping specified as $\Phi(\vec{p}_i) = C_i, i = 1,\dots,4$. Since $\vec{p}_i$ falsifies C_i, $i = 1,\dots,4$, Φ is a correct AC-mapping. P is an SSA of H with respect to Φ and center $\vec{p}_{init}$=$\vec{p}_1$. Indeed, $Nbhd(\vec{p}_{init},\vec{p}_1,C_1)$=$\{\vec{p}_2,\vec{p}_3,\vec{p}_4\}$ where $C_1 = \Phi(\vec{p}_1)$ and $Nbhd(\vec{p}_{init},\vec{p}_i,C_i)$ = $\emptyset$, where $C_i = \Phi(\vec{p}_i)$, $i = 2,3,4$. Thus, $Nbhd(\vec{p}_{init},\vec{p}_i,\Phi(\vec{p}_i)) \subseteq P$, $i = 1,\dots,4$.

B. SSAs and satisfiability of a formula

Proposition 1: Formula H is unsatisfiable iff it has an SSA.

The proof is given in [11]. A similar proposition is proved in [9] for "uncentered" SSAs (see Footnote 8).

```
BuildPath(H, Φ, $\vec{p}_{init}$, $\vec{s}$){
1  $Path := nil$
2  $\vec{p}_1 := \vec{p}_{init}$
3  $i := 1$
4  while ($\vec{p}_i \neq \vec{s}$) {
5    $Path := Extend(Path, \vec{p}_i)$
6    $C := \Phi(\vec{p}_i)$
7    $v := FindVar(C, \vec{p}_i, \vec{s})$
8    $\vec{p}_{i+1} := FlipVar(\vec{p}_i, v)$
9    $i := i+1$ }
10 return($Path$) }
```

Fig. 1. *BuildPath* procedure

The set of all assignments to $Vars(H)$ forms the *trivial* uncentered SSA of H. Example 2 shows a *non-trivial* SSA. The fact that formula H has a non-trivial SSA P is its *structural* property. That is one cannot check whether P is an SSA of H if only the truth table of H is known. In particular, P may not be an SSA of a formula H' logically equivalent to H.

The relation between SSAs and satisfiability can be explained as follows. Suppose that formula H is satisfiable. Let $\vec{p}_{init}$ be an arbitrary assignment to $Vars(H)$ and $\vec{s}$ be a satisfying assignment that is the closest to $\vec{p}_{init}$ in terms of the Hamming distance. Let P be the set of all assignments to $Vars(H)$ that falsify H and Φ be an AC-mapping from P to H. Then $\vec{s}$ can be reached from $\vec{p}_{init}$ by procedure *BuildPath* shown in Figure 1. It generates a sequence of assignments $\vec{p}_1,\dots,\vec{p}_i$ where $\vec{p}_1$ = $\vec{p}_{init}$ and $\vec{p}_i$=$\vec{s}$. First, *BuildPath* checks if current assignment $\vec{p}_i$ equals $\vec{s}$. If so, then $\vec{s}$ has been reached. Otherwise, *BuildPath* uses clause $C = \Phi(\vec{p}_i)$ to generate next assignment. Since $\vec{s}$ satisfies C, there is a variable $v \in Vars(C)$ that is assigned differently in $\vec{p}_i$ and $\vec{s}$. *BuildPath* generates a new assignment $\vec{p}_{i+1}$ obtained from $\vec{p}_i$ by flipping the value of v.

[6]Implication $F_N \wedge z \rightarrow H$ is a *semantic* property of $F_N \wedge z$. To verify this property it suffices to know the truth table of $F_N \wedge z$.

[7]We use the set of clauses $\{C_1,\dots,C_k\}$ as an alternative representation of a CNF formula $C_1 \wedge \dots \wedge C_k$.

[8]In [9], the notion of "uncentered" SSAs was introduced. The definition of an uncentered SSA is similar to Definition 2. The only difference is that one requires that for every $p_i \in P$, $Nbhd(\vec{p}_i,C) \subseteq P$ holds instead of $Nbhd(\vec{p}_{init},\vec{p}_i,C) \subseteq P$. The advantage of centered SSAs is that they are usually much smaller than uncentered SSAs.

```
$BuildSSA(H)\{$
  $E = \emptyset$; $\Phi := \emptyset$
  $\vec{p}_{init} := PickInitAssgn(H)$
  $Q := \{\vec{p}_{init}\}$
  while $(Q \neq \emptyset)$ {
     $\vec{p} := PickAssgn(Q)$
     $Q := Q \setminus \{\vec{p}\}$
     if $(SatAssgn(\vec{p}, H))$
        return($\vec{p}, nil, nil, nil$)
     $C := PickFlsCls(H, \vec{p})$
     $R := Nbhd(\vec{p}_{init}, \vec{p}, C) \setminus E$
     $Q := Q \cup R$
     $E := E \cup \{\vec{p}\}$
     $\Phi := \Phi \cup \{(\vec{p}, C)\}\}$
  return($nil, E, \vec{p}_{init}, \Phi$) }
```

Fig. 2. *BuildSSA* procedure

BuildPath reaches $\vec{s}$ in k steps where k is the Hamming distance between $\vec{p}_{init}$ and $\vec{s}$. Importantly, *BuildPath* reaches $\vec{s}$ for *any* AC-mapping. Let P be an SSA of H with respect to center $\vec{p}_{init}$ and AC-mapping Φ. Then if *BuildPath* starts with $\vec{p}_{init}$ and uses Φ as an AC-mapping, it can reach only assignments of P. Since every assignment of P falsifies H, no satisfying assignment can be reached.

A procedure for generation of SSAs called *BuildSSA* is shown in Figure 2. It accepts formula H and outputs either a satisfying assignment or an SSA of H, center $\vec{p}_{init}$ and AC-mapping Φ. *BuildSSA* maintains two sets of assignments denoted as E and Q. Set E contains the examined assignments i.e. those whose neighborhood is already explored. Set Q specifies assignments that are queued to be examined. Q is initialized with an assignment $\vec{p}_{init}$ and E is originally empty. *BuildSSA* updates E and Q in a *while* loop. First, *BuildSSA* picks an assignment $\vec{p}$ of Q and checks if it satisfies H. If so, $\vec{p}$ is returned as a satisfying assignment. Otherwise, *BuildSSA* removes $\vec{p}$ from Q and picks a clause C of H falsified by $\vec{p}$. The assignments of $Nbhd(\vec{p}_{init}, \vec{p}, C)$ that are not in E are added to Q. After that, $\vec{p}$ is added to E as an examined assignment, pair $(\vec{p}, C)$ is added to Φ and a new iteration begins. If Q is empty, E is an SSA with center $\vec{p}_{init}$ and AC-mapping Φ.

III. Complete Test Sets

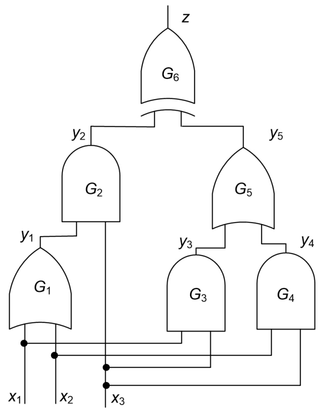

Fig. 3. Example of circuit $N(X, Y, z)$

Let $N(X, Y, z)$ be a single-output combinational circuit where X and Y specify the input and internal variables of N respectively and z specifies the output variable of N. Let N consist of gates $G_1, \ldots, G_k$. Then N can be represented as $F_N = F_{G_1} \wedge \cdots \wedge F_{G_k}$ where $F_{G_i}, i = 1, \ldots, k$ is a CNF formula specifying the consistent assignments of gate G_i. Proving $N \equiv 0$ reduces to showing that formula $F_N \wedge z$ is unsatisfiable.

Example 3: Circuit N shown in Figure 3 represents equivalence checking of expressions $(x_1 \vee x_2) \wedge x_3$ and $(x_1 \wedge x_3) \vee (x_2 \wedge x_3)$ specified by gates G_1, G_2 and G_3, G_4, G_5 respectively. Formula F_N is equal to $F_{G_1} \wedge \cdots \wedge F_{G_6}$ where, for instance, $F_{G_1} = C_1 \wedge C_2 \wedge C_3$, $C_1 = x_1 \vee x_2 \vee \overline{y}_1$, $C_2 = \overline{x}_1 \vee y_1$, $C_3 = \overline{x}_2 \vee y_1$. Every assignment satisfying F_{G_1} corresponds to a consistent assignment to gate G_1 and vice versa. For instance, $(x_1 = 0, x_2 = 0, y_1 = 0)$ satisfies F_{G_1} and is a consistent assignment to G_1 since the latter is an OR gate. Formula $F_N \wedge z$ is unsatisfiable since $(x_1 \vee x_2) \wedge x_3 \equiv (x_1 \wedge x_3) \vee (x_2 \wedge x_3)$. Thus, $N \equiv 0$.

Let $\vec{x}$ be a test i.e. an assignment to X. The set of assignments to $Vars(N)$ sharing the same assignment $\vec{x}$ to X forms a cube of $2^{|Y|+1}$ assignments. (Recall that $Vars(N) = X \cup Y \cup \{z\}$). Denote this set as $Cube(\vec{x})$. Only one assignment of $Cube(\vec{x})$ specifies the correct execution trace produced by N under $\vec{x}$. All other assignments can be viewed as "erroneous" traces under test $\vec{x}$.

Definition 3: Let T be a set of tests $\{\vec{x}_1, \ldots, \vec{x}_k\}$ where $k \leq 2^{|X|}$. We will say that T is a **Complete Test Set (CTS)** for N if $Cube(\vec{x}_1) \cup \cdots \cup Cube(\vec{x}_k)$ contains an SSA for formula $F_N \wedge z$.

```
$SeSt(G, V)\{$
  $H := \emptyset$
  foreach $(C \in G)$
     if $(Vars(C) \subseteq V)$
        $H := H \cup \{C\}$
  while (true) {
     $(\vec{v}, P) := BuildSSA(H)$
     if $(P \neq nil)$
        return($nil, H, P$)
     $(C, \vec{s}) := GenCls(G, V, \vec{v})$
     if $(\vec{s} \neq nil)$
        return($\vec{s}, nil, nil$)
     $H := H \cup \{C\}$ }
```

Fig. 4. *SeSt* procedure

If T satisfies Definition 3, set $Cube(\vec{x}_1) \cup \cdots \cup Cube(\vec{x}_k)$ "contains" a proof that $N \equiv 0$ and so T can be viewed as complete. If $k = 2^{|X|}$, T is the *trivial* CTS. In this case, $Cube(\vec{x}_1) \cup \cdots \cup Cube(\vec{x}_k)$ contains the trivial SSA consisting of all assignments to $Vars(F_N \wedge z)$. Given an SSA P of $F_N \wedge z$, one can easily generate a CTS by extracting all different assignments to X that are present in the assignments of P.

Example 4: Formula $F_N \wedge z$ of Example 3 has an SSA of 21 assignments to $Vars(F_N \wedge z)$. They have only 5 different assignments to $X = \{x_1, x_2, x_3\}$. The set $\{101, 100, 011, 010, 000\}$ of those assignments is a CTS for N.

Definition 3 is meant for circuits that are not "too redundant". Highly-redundant circuits are discussed in [12], [11].

IV. *SeSt* Procedure

A. Motivation

Building an SSA for a large formula is inefficient. So, constructing a CTS of N from an SSA of $F_N \wedge z$ is impractical. To address this problem, we introduce a procedure called *SeSt* (a short for "Semantics and Structure"). Given formula $F_N \wedge z$ and a set of variables $V \subseteq Vars(F_N \wedge z)$, *SeSt* generates a simpler formula $H(V)$ implied by $F_N \wedge z$ at the same time trying to build an SSA for H. If *SeSt* succeeds in constructing such an SSA, formula H is unsatisfiable and so is $F_N \wedge z$. Then a set of tests T is extracted from this SSA. As we show in Subsection V-A, one can view T as an approximation of a CTS for N (if $X \subseteq V$) or an "approximation of approximation" of a CTS (if $X \not\subseteq V$).

Example 5: Consider the circuit N of Figure 3 where $X = \{x_1, x_2, x_3\}$. Assume that $V = X$. Application of *SeSt* to $F_N \wedge z$ produces $H(X) = (\overline{x}_1 \vee \overline{x}_3) \wedge (\overline{x}_2 \vee \overline{x}_3) \wedge (x_1 \vee x_2) \wedge x_3$. *SeSt* also generates an SSA of H of four assignments to X:

$\{000, 001, 011, 101\}$ with center $\vec{p}_{init}$=000. (We omit the AC-mapping here.) These assignments form an approximation of a CTS for N.

B. *Description of SeSt*

The pseudocode of *SeSt* is shown in Figure 4. *SeSt* accepts formula G (in our case, $G := F_N \wedge z$) and a set of variables $V \subseteq \mathit{Vars}(G)$. *SeSt* outputs an assignment satisfying G or formula $H(V)$ implied by G and an SSA of H. Initially, H consists of the clauses of G depending only on variables of V (if any). Then a *while* loop is performed. First, *SeSt* tries to build an SSA for the current formula H by calling *BuildSSA* (line 6). If H is unsatisfiable, *BuildSSA* computes an SSA P returned by *SeSt* along with H (line 8). Otherwise, *BuildSSA* returns an assignment $\vec{v}$ satisfying H. In this case, *SeSt* calls procedure *GenCls* to build a clause C falsified by $\vec{v}$. Clause C is obtained by resolving clauses of G on variables of $\mathit{Vars}(G) \setminus V$. (Hence C is implied by G.) If $\vec{v}$ can be extended to an assignment $\vec{s}$ satisfying G, *SeSt* terminates (lines 10-11). Otherwise, C is added to H and a new iteration begins.

```
GenCls($G, V, \vec{v}$){
1 $G_{\vec{v}} := \mathit{GenForm}(F, \vec{v})$
2 $(\vec{s}, R) := \mathit{ChkSat}(G_{\vec{v}})$
3 if ($\vec{s} \neq \mathit{nil}$)
4     return($\mathit{nil}, \vec{s} \cup \vec{v}$)
5 $V' := \mathit{Analyze}(R, G_{\vec{v}}, G)$
6 $C := \mathit{FormCls}(V', \vec{v})$
7 return($C, \mathit{nil}$)
```

Fig. 5. *GenCls* procedure

Procedure *GenCls* is shown in Figure 5. First, *GenCls* generates formula $G_{\vec{v}}$ obtained from G by discarding clauses satisfied by $\vec{v}$ and removing literals falsified by $\vec{v}$. Then *GenCls* checks if there is an assignment $\vec{s}$ satisfying $G_{\vec{v}}$. If so, $\vec{s} \cup \vec{v}$ is returned as an assignment satisfying G. Otherwise, a proof R of unsatisfiability of $G_{\vec{v}}$ is produced. Then *GenCls* forms a set $V' \subseteq V$. A variable w is in V' iff a clause of $G_{\vec{v}}$ is used in proof R and its parent clause from G has a literal of w falsified by $\vec{v}$. Finally, clause C is generated as a disjunction of literals of V' falsified by $\vec{v}$. By construction, clause C is implied by G and falsified by $\vec{v}$.

V. Building Approximations Of CTS

A. *Two kinds of approximations of CTSs*

As before, let $H(V)$ denote a formula implied by $F_N \wedge z$ that is generated by *SeSt* and P denote an SSA for H. Projections of N can be of two kinds depending on whether $X \subseteq V$ holds. Let $X \subseteq V$ be true and T be the test set consisting of all different assignments to X present in the assignments of P. Using the reasoning of Section III one can show that T is a CTS for projection of N on V. Since $H(V)$ is essentially an abstraction of $F_N \wedge z$, one can view T an approximation of a CTS for N. For that reason, we will refer to T as a **CTS$^{\mathbf{a}}$** of N where superscript "a" stands for "approximation".

Now assume $X \subseteq V$ is not true. Generation of a test set T from P for this case is described in the next section. Let us relate this case to that of $X \subseteq V$. Assume for the sake of simplicity that $V \cap X = \emptyset$. Let us consider computing a test set T' for a projection of N on set V' where $V' = X \cup V$. Let P' be an SSA for formula $H'(V')$ generated by *SeSt*. Every assignment of P' can be represented as $(\vec{x}$,$\vec{v})$ where $\vec{x}$ and $\vec{v}$ are assignments to X and V respectively. The assignments $(\vec{x}_1, \vec{v}), (\vec{x}_2, \vec{v}), \dots$ of P' sharing the same $\vec{v}$ specify all tests of T' corresponding to $\vec{v}$. On the other hand, since $V \cap X = \emptyset$, to generate T one has to a) use some *heuristic* for generating a test corresponding to $\vec{v}$ and b) *guess* how many tests corresponding to $\vec{v}$ one should generate. Thus, T is an approximation of T' that is itself a CTS^{a} i.e. an approximation of a CTS. So, we will refer to T as **CTS$^{\mathbf{aa}}$**.

B. *Construction of CTS^{aa}*

Consider extraction of a test set T from SSA P of formula $H(V)$ when $X \not\subseteq V$. Since V, in general, contains internal variables[9] of N, translation of P to a test set T needs a special procedure *GenTests* shown in Figure 6. As we mentioned in Subsection V-A, building a test $\vec{x}$ corresponding to an assignment $\vec{v}$ of P requires some heuristic. In *GenTests*, we use the following idea. One can view building an SSA (see Fig. 2) as a try to reach a satisfying assignment, if any. So, intuitively, every assignment of a good SSA falsifies a very small number of clauses of G. For that reason, when building a test $\vec{x}$ corresponding to $\vec{v}$, we look for an assignment to $\mathit{Vars}(F_N \wedge z)$ that contains $\vec{x}$ and $\vec{v}$ and falsifies as few clauses of $F_N \wedge z$ as possible.

```
GenTests($F_N$,$X$,$P$,$tr_1$,$tr_2$){
1  $T := \emptyset$
2  for each $\vec{v} \in P$ {
3    $\vec{s} := \mathit{SatAssgn}(F_N, \vec{v})$
4    if ($\vec{s} \neq \mathit{nil}$) {
5      $\mathit{AddTest}(T, \vec{s}, X)$
6      for ($i = 1$;$i < tr_1$;$i$++){
7        $\vec{s} := \mathit{SatAssgn}(F_N, \vec{v})$
8        $\mathit{AddTest}(T, \vec{s}, X)$}
9    else
10     for ($i = 0$;$i < tr_2$;$i$++){
11       $F^*_N := \mathit{Relax}(F_N)$
12       $\vec{s} := \mathit{SatAssgn}(F^*_N, \vec{v})$
13       if ($\vec{s} = \mathit{nil}$) continue
14       $\mathit{AddTest}(T, \vec{s}, X)$}}
15 return($T$)}
```

Fig. 6. *GenTests* procedure

Parameters tr_1 and tr_2 control the number of tests generated for one assignment of P (*tr* here stands for "tries"). For every $\vec{v} \in P$, *GenTests* checks if formula F_N is satisfiable under assignment $\vec{v}$ i.e. if there exists a test under which N assigns $\vec{v}$ to V. If so, *GenTests* calls procedure *AddTest* that forms a new test by extracting the values assigned to X in $\vec{s}$ and adds it to T. (Note that the only clause of $F_N \wedge z$ falsified by $\vec{s}$ is the unit clause z.) Then *GenTests* runs a *for* loop (lines 6-8) to generate tr_1-1 more tests producing the same assignment $\vec{v}$. We assume that the SAT-solver invoked in line 7 generates different satisfying assignments in different calls.

If F_N is unsatisfiable under $\vec{v}$, *GenTests* runs another *for* loop of tr_2 iterations (lines 10-14). In every iteration, *GenTests* relaxes F_N by removing the clauses specifying a small random subset of gates. If the relaxed version of F_N has a satisfying assignment $\vec{s}$ (line 12), a test is extracted from $\vec{s}$ and added to T. Note that $\vec{s}$ falsifies only a small number of clauses of $F_N \wedge z$, namely, a subset of clauses removed to relax F_N and possibly the unit clause z.

C. *Finding a set of variables to project on*

[9]If the special case $V \subset X$ holds, every assignment of P can be easily turned into a test by assigning values to variables of $X \setminus V$ (e.g. randomly).

```
GenCut(N, Size){
1  $G_{out} := OutGate(N)$
2  $Gts := \{G_{out}\}$
3  $Dpth(G_{out}) := 0$
4  $Inps := \emptyset$
5  while ($|Gts \cup Inps| < Size$) {
6    $G := MinDepth(Gts, Dpth)$
7    $Gts := Gts \setminus \{G\}$
8    $Seen(G) := true$
9    foreach $G' \in FanIn(G)${
10     if ($Seen(G')$) continue
11     if ($G' \in Inputs(N)$) {
12       $Inps = Inps \cup \{G'\}$
13       continue }
14     $Dpth(G') := Dpth(G)+1$
15     $Gts := Gts \cup \{G'\}$}}
16 return($Gts \cup Inps$)}
```

Fig. 7. *GenCut* procedure

Intuitively, a good choice of the set V to project N on is a (small) coherent subset of variables of N reflecting its structure and/or semantics. One obvious choice of V is the set X of input variables of N. In this section, we describe generation of a set V whose variables form an internal cut of N denoted as *Cut*. Procedure *GenCut* for generation of set *Cut* consisting of *Size* gates is shown in Figure 7. Set V is formed from output variables of the cut gates.

The current cut is specified by $Gts \cup Inps$. Set *Gts* is initialized with the output gate G_{out} of circuit N and *Inps* is originally empty. *GenCut* computes the *depth* of every gate of *Gts*. The depth of G_{out} is set to 0. Set *Gts* is processed in a *while* loop (lines 5-15). In every iteration, a gate of the smallest depth is picked from *Gts*. Then *GenCut* removes gate G from *Gts* and examines the fan-in gates of G (lines 9-15). Let G' be a fan-in gate of G that has not been seen yet and is not a primary input of N. Then the depth of G' is set to that of G plus 1 and G' is added to *Gts*. If G' is a primary input of N it is added to *Inps*.

VI. Applications Of Property-Checking Tests

Given a multi-output circuit M, traditional testing is used to verify M "as a whole". In this paper, we describe generation of a test set meant for checking a *particular property* of M specified by a single-output circuit N. In this section, we present some applications of property-checking test sets.

A. Verification of corner cases

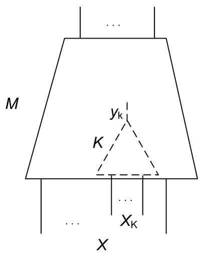

Fig. 8. Subcircuit K of circuit M

Let K be a single-output subcircuit of circuit M as shown in Figure 8. For the sake of simplicity, here, we consider the case where the set X_K of input variables of K is a subset of the set X of input variables of M. (The technique below can also be applied when input variables of K are *internal* variables of M.) Suppose K evaluates, say, to value 0 much more frequently then to 1. Then one can view an input assignment of M for which K evaluates to 1 as specifying a "corner case" i.e. a rare event. Hitting such a corner case by a random test can be very hard. This issue can be addressed by using a coverage metric that *requires* setting the value of K to both 0 and 1. (The task of finding a test for which K evaluates to 1 can be solved, for instance, by a SAT-solver.) The problem however is that hitting a corner case only once may be insufficient.

One can increase the frequency of hitting the corner case above as follows. Let N be a miter of circuits K' and K'' (see Figure 9) i.e. a circuit that evaluates to 1 iff K' and K'' are functionally inequivalent. Let K' and K'' be two copies of circuit K. So $N \equiv 0$ holds. Let test set T_K be extracted from an SSA built for a projection of N on a set $V \subset Vars(N)$. Set T_K can be viewed as a result of "squeezing" the truth table of K. Since this truth table is dominated by input assignments for which K evaluates to 0, this part of the truth table is *reduced the most*. So, one can expect that the ratio of tests of T_K for which K evaluates to 1 is higher than in the truth table of K. In Subsection VII-B, we substantiate this intuition experimentally. One can easily extend an assignment $\vec{x}_K$ of T_K to an assignment $\vec{x}$ to X e.g. by randomly assigning values to the variables of $X \setminus X_K$.

B. Testing sequential circuits

There are a few ways to apply property-checking tests meant for combinational circuits to verification of *sequential* circuits. Here is one of them based on bounded model checking [2]. Let M be a sequential circuit and ξ be a property of M. Let $N_k(X,Y,z)$ be a circuit such that $N_k \equiv 0$ holds iff ξ is true for k time frames. Circuit N_k is obtained by unrolling M k times and adding logic specifying property ξ. Set X consists of the subset X' specifying the state variables of M in the first time frame and subset X'' specifying the combinational input variables of M in k time frames.

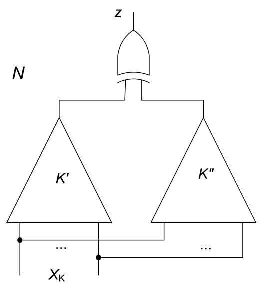

Fig. 9. The miter of circuits K' and K''

Having constructed N_k, one can build CTSs, CTS^{a}s and CTS^{aa}s for testing property ξ of M. The only difference here from the problem we have considered so far is as follows. Circuit M starts in a state satisfying some formula $I(X')$ that specifies the initial states. So, one needs to check if $N_k \equiv 0$ holds only for the assignments to X satisfying $I(X')$. A test here is an assignment $(\vec{x'}_1, \vec{x''}_1, \dots, \vec{x''}_k)$ where $\vec{x'}_1$ is an initial state and $\vec{x''}_i$, $1 \leq i \leq k$ is an assignment to the combinational input variables of i-th time frame. Given a test, one can easily compute the corresponding sequence of states $(\vec{x'}_1, \dots, \vec{x'}_k)$ of M. In Subsection VII-C, we give examples of building CTS^{aa}s for testing sequential circuits.

C. Exposing bugs overlooked due to misdefining properties

One can use property-checking tests to mitigate the problem of incomplete specifications. By running tests generated for an incomplete set of properties of M, one can expose bugs overlooked due to missing some properties. An important special case of this problem is as follows. Let ξ be a property of M that holds. Assume that the correctness of M requires proving a slightly *different* property ξ' that *does not hold*. By running a test set T built for property ξ, one may expose a bug overlooked in formal verification due to proving ξ instead of

ξ'. In Subsection VII-C, we illustrate this idea experimentally. Note that the problem above has nothing to do with the complexity of proving ξ' false. The designer simply does not know that *there is* a problem and so can overlook a bug even if proving ξ' false is very easy.

VII. Experiments

In this section, we describe experiments with property-checking tests (PCT) generated by procedure *GenPCT* shown in Figure 10. *GenPCT* accepts a single-output circuit N and outputs a set of tests T. (For the sake of simplicity, we assume here that $N \equiv 0$ holds.) *GenPCT* starts with generating formula $F_N \wedge z$. Then it builds a set of variables $V \subseteq \mathit{Vars}(F_N \wedge z)$. Parameter *type* specifies whether *GenPCT* is supposed to generate a CTS, CTS^{a} or CTS^{aa}. After that, *GenPCT* calls SeSt (see Fig. 4) to compute a formula $H(V)$ implied by $F_N \wedge z$ and its SSA.

```
$\mathit{GenPCT}(N, X, \mathit{type}, \mathit{tr}_1, \mathit{tr}_2)\{$
1  $F_N \wedge z := \mathit{GenForm}(N)$
2  $V := \mathit{GenVars}(F_N \wedge z, \mathit{type})$
3  $(H, P) := \mathit{SeSt}(F_N \wedge z, V)$
4  if $(X \subseteq V)$
5    $T := \mathit{ExtrTests}(X, P)$
6  else {
7    $\mathit{RedVars} := V \setminus \mathit{Vars}(H)$
8    $P := \mathit{Drop}(P, \mathit{RedVars})$
9    $T := \mathit{GenTests}(F_N, X, P, \mathit{tr}_1, \mathit{tr}_2)\}$
10 return$(T)\}$
```

Fig. 10. *GenPCT* procedure

If $X \subseteq V$ holds (where X is the set of input variables of N), *GenPCT* computes T as the set of all different assignments to X present in assignments of P (line 5). Otherwise, *GenPCT* calls procedure *GenTests* (see Fig. 6). Every variable $w \in V \setminus \mathit{Vars}(H)$ is redundant in the sense that its value is the same in all assignments of P. So the values assigned to $V \setminus \mathit{Vars}(H)$ are dropped by *GenTests* (lines 7-8). If $V = \mathit{Vars}(F_N \wedge z)$, then $H(V)$ is $F_N \wedge z$ itself and *GenPCT* produces a CTS of N. Otherwise, according to definitions of Subsection V-A, *GenPCT* generates a CTS^{a} (if $X \subseteq V$) or CTS^{aa} (if $X \not\subseteq V$).

In the following subsections, we describe results of three experiments. In the first two experiments we used circuits specifying next state functions of latches of HWMCC-10 benchmarks. (The motivation was to employ realistic circuits.) In the third experiment, we used combinational circuits obtained by unfolding HWMCC-10 benchmarks. In our implementation of SeSt, as a SAT-solver, we used Minisat 2.0 [6], [17]. We also employed Minisat to run simulation. To compute the output value of N under test $\vec{x}$, we added unit clauses specifying $\vec{x}$ to formula $F_N \wedge z$ and checked its satisfiability.

A. *Comparing CTSs, CTS^{a}s and CTS^{aa}s*

The objective of the first experiment was to give examples of circuits with non-trivial CTSs and compare the efficiency of computing CTSs, CTS^{a}s and CTS^{aa}s. In this experiment, N was a miter specifying equivalence checking of circuits M' and M'' (see Figure 9). M'' was obtained from M' by optimizing the latter with ABC [15].

The results of the first experiment are shown in Table I. The first two columns specify an HWMCC-10 benchmark and its latch whose next state function was used as M'. The next two columns give the number of input variables and that of gates in the miter N. The following pair of columns describe computing a CTS for N. The first column of this pair gives the size of the SSA P found by *GenPCT* in thousands. The number of tests in the set T extracted from P is shown in the parentheses in thousands. The second column of this pair gives the run time of *GenPCT* in seconds.

TABLE I
Computing CTSs, CTS^{a}s and CTS^{aa}s

name	la-tch	#inp vars	#ga-tes	CTS		CTS^{a} or CTS^{aa}			
				$\lvert SSA \rvert$ (#tests) $\times 10^3$	time (s.)	test set type	$\lvert V \rvert$	$\lvert SSA \rvert$ (#tests) $\times 10^3$	time (s.)
bob3	L26	14	41	46 (2.0)	0.1	CTS^{a}	14	0.6 (0.6)	0.01
eijks258	L10	16	45	259 (8.2)	0.5	CTS^{a}	16	0.1 (0.1)	0.02
cmudme1	L230	19	50	2,184 (63)	5.4	CTS^{a}	19	13 (13)	0.1
mutexp0	L60	29	199	memout	$*$	CTS^{a}	29	659 (659)	26
pdtpmsmiim	L118	31	136	memout	$*$	CTS^{a}	31	936 (936)	4.2
abp4pold	L270	129	1,178	memout	$*$	CTS^{aa}	22	0.9 (0.5)	0.6
pj2009	L1318	366	25,160	memout	$*$	CTS^{aa}	22	0.6 (0.3)	51
mentorb..00	L8670	626	3,156	memout	$*$	CTS^{aa}	22	1.2 (0.6)	11
139454p0	L1676	791	19,843	memout	$*$	CTS^{aa}	22	0.1 (0.1)	99

The last four columns of Table I describe results of computing test sets for a projection of N on a set of variables V. The first column of this group shows if CTS^{a} or CTS^{aa} was computed whereas the next column gives the size of V. The third column of this group provides the size of SSA P and the test set T extracted from P (in parentheses). Both sizes are given in thousands. The last column shows the run time of *GenPCT*. For the first five examples, we used a projection of N on X, thus constructing a CTS^{a} of N. For the last four examples we computed a projection of N on an internal cut (see Subsection V-C) thus generating a CTS^{aa} of N. *GenPCT* was called with $\mathit{tr}_1 = 1, \mathit{tr}_2 = 5$ (see Fig. 6 and 10).

For the first three examples, *GenPCT* managed to build non-trivial CTSs that are smaller than $2^{|X|}$. For instance, the trivial CTS for example *bob3* consists of 2^{14}=16,384 tests, whereas *GenPCT* found a CTS of 2,004 tests. (So, to prove M' and M'' equivalent it suffices to run 2,004 out of 16,384 tests.) For the other examples, *GenPCT* failed to build a non-trivial CTS due to exceeding the memory limit (1.5 Gbytes). On the other hand, *GenPCT* built a CTS^{a} or CTS^{aa} for all nine examples of Table I. Note, however, that CTS^{a}s give only a moderate improvement over CTSs. For the last four examples *GenPCT* failed to compute a CTS^{a} of N due to memory overflow whereas it had no problem computing an CTS^{aa} of N. So CTS^{aa}s can be computed efficiently even for large circuits. Further, we show that CTS^{aa}s are also very effective.

B. *Testing corner cases*

In the second experiment, we generated CTS^{a}s and CTS^{aa}s to test corner cases (see Subsection VI-A). First, we formed a circuit K that evaluates to 0 for almost all input assignments. So, the assignments for which K evaluates to 1 are corner cases[10]. We compared the frequency of hitting corner cases by random tests and by tests of a set T built by *GenPCT* as

[10]We assume here that K is a subcircuit of some circuit M. The input assignments for which K evaluates to 1 are corner cases for M.

follows. Let N be a miter of copies K' and K'' (see Figure 9). The set T was generated using a projection of N either on the set X of input variables or an internal cut of N.

TABLE II
Testing corner cases

name	la-tch	#inp vars	#and vars	#ga-tes	random testing		testing by CTS^{a} and CTS^{aa}				
					#te-sts	#hits %	test set	$\|V\|$	#te-sts	#hits %	time (s.)
pd..gigamax5	L46	43	10	512	10^5	0.02	CTS^{a}	43	547	7.1	0.2
pd..gigamax5	L46	63	30	512	10^8	0	CTS^{a}	63	1,243	3.0	0.2
pdtvisbpb1	L48	46	10	108	10^5	0.04	CTS^{a}	46	398	9.0	0.01
pdtvisbpb1	L48	66	30	108	10^8	0	CTS^{a}	66	736	3.1	0.03
abp4pold	L270	139	10	637	10^5	0.02	CTS^{aa}	35	2,047	8.5	0.9
abp4pold	L270	159	30	637	10^8	0	CTS^{aa}	55	5,256	3.3	2.1
mentorbm1p00	L8670	636	10	1,630	10^5	0.1	CTS^{aa}	35	594	11	3.7
mentorbm1p00	L8670	656	30	1,630	10^8	0	CTS^{aa}	55	2,009	4.7	8.7

To build circuit K, we extracted the circuit R specifying the next state function of a latch of a HWMCC-10 benchmark and composed it with an n-input AND gate as shown in Figure 11. The circuit K outputs 1 only if R evaluates to 1 and the first $n-1$ inputs variables of the AND gate are set to 1 too. So the input assignments for which K evaluates to 1 are "corner cases".

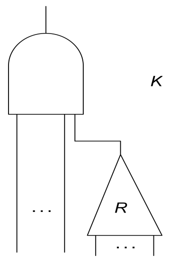

Fig. 11. Circuit K whose output value is biased to 0

The results of the experiment are given in Table II. The first two columns name the benchmark and latch whose next state function was used as circuit R. The next three columns give the total number of input variables of K, the value of n in the n-input AND gate fed by R and the number of gates in circuit K. The following pair of columns describes the performance of random testing. The first column of this pair gives the total number of tests. The next column shows the percentage of times circuit K evaluated to 1 (and so a corner case was hit). The last five columns of Table II describe the results of *GenPCT*. The first column of the five indicates whether a CTS^{a} or CTS^{aa} was generated. The second column gives the size of set V on which a projection of N was computed. CTS^{a}s were generated with $V = X$. When computing CTS^{aa}s, the set V formed an internal cut of N and parameters tr_1 and tr_2 were both set to 1. The next column shows the size of the test set. The fourth column gives the percentage of times a corner case was hit. The last column shows the total run time.

The examples of Table II were generated in pairs that shared the same circuit R and were different only in the size of the AND gate fed by R. For instance, in the first and second entry of Table II, circuit K was obtained by composing the same circuit R extracted from benchmark *pdtvisgigamax5* with 10-input and 30-input AND gates respectively. Table II shows that for circuits with a 10-input AND gate, random testing hit corner cases but the percentage of those events was much lower than for CTS^{a}s and CTS^{aa}s. On the other hand, even 100 millions of random tests failed to hit a single corner case for examples with a 30-input AND gate in sharp contrast to CTS^{a}s and CTS^{aa}s.

C. Testing properties defined incorrectly

TABLE III
Testing "misdefined" properties. CTS^{aa}s were computed for $|V| = 20$. Test sets with a counterexample are shown ***in bold.***

name	#ti-me fra-mes	#inp vars	#ga-tes $\times 10^3$	cov. met. tests		random tests		testing by CTS^{aa}		
				#tests	time (s.)	#tests	time (s.)	#iter	#tests	time (s.)
bobcount	19	38	1.6	740	0.4	$\mathbf{1.0*10^7}$	294	1	**3,339**	1.1
boblivea	5	65	8.0	3,778	7.2	$\mathbf{9.7*10^3}$	2.1	100	9,982	74
p..gigamax0	4	88	4.3	2,150	6.3	$\mathbf{1.4*10^6}$	158	20	**923**	3.7
kenflashp01	2	108	2.5	1,076	0.8	10^8	1,625	48	**6,027**	1.7
pdtpmsudc8	10	110	3.7	2,066	2.5	$6.8*10^7$	5,000	100	51,123	283
eijks526	39	117	18	**8,976**	70	$4.5*10^6$	5,000	1	**183**	31
kenopp1	3	129	1.7	1,202	0.5	10^8	695	13	**1,344**	0.4
vis..cellp01	5	135	14	**4,581**	16	$8*10^7$	5,000	13	**1,354**	4.4
cmugigamax	5	159	3.1	1,826	2.3	10^8	2,671	100	8,985	13
eijks5378	6	209	17	8,318	56	$\mathbf{3.4*10^4}$	58	1	**387**	3.6
eijks208o	25	250	4.0	1,506	3.6	$\mathbf{1.9*10^7}$	2,207	3	**1,811**	4.9
eijks420	18	324	6.6	1,115	3.7	$\mathbf{4.1*10^6}$	1,140	86	**26,199**	82
n..guidancep1	6	504	10	7,922	27	$2.1*10^7$	5,000	6	**378**	2.3
pdt..feistel	12	816	115	**68,006**	4,066	$3.9*10^6$	5,000	5	**804**	49
nusmvtcasp2	7	1,029	19	11,510	82	$4.5*10^7$	5,000	38	**3,549**	53
cmuperiodic	34	1,220	51	**30,999**	760	$9.5*10^6$	5,000	85	**5,611**	240
pj2002	4	4,054	137	**61,113**	3.868	$0.6*10^6$	5,000	2	**161**	7.9

The objective of the third experiment was to expose bugs overlooked due to incorrect definition of properties (see Subsection VI-C). In contrast to the previous two experiments, here we employed "complete" HWMCC-10 benchmarks, each benchmark specifying a safety property ξ of a sequential circuit M. In our experiment, we used benchmarks with *true* properties. We assumed that ξ was defined incorrectly and formed a new property ξ' of M that failed. Property ξ' served as the "real" property to check. It was obtained by changing the functionality of a gate of M involved in specifying property ξ. The fact that ξ' indeed failed was established by running IC3 [3]. Let k denote the length of the counterexample found by IC3 for ξ'. We unrolled the transition relation of M k times to generate single-output circuits N_k and N'_k. These circuits evaluated to 1 iff no counterexample of length k existed for ξ and ξ' respectively. By construction, $N_k \equiv 0$ held whereas $N'_k \equiv 0$ did not.

In our experiment, we compared three different methods of breaking property ξ'. In the first method, we used testing driven by a coverage metric. Namely, we generated a test set T aimed at setting the output[11] of every gate G of N_k both to 0 and 1. Then we applied T to N'_k to disprove $N'_k \equiv 0$. Note that a single test sets the output of every gate of N_k to 0 or 1. To make T stronger, when processing a gate G of N_k we tried to find a new test setting the output of G to $b \in \{0,1\}$, even if this goal was "inadvertently" achieved earlier. In the

[11]In [11], we give results for the coverage metric based on stuck-at faults.

second method, we simply applied random tests[12] to N'_k until a counterexample was generated or a resource was exceeded. In the third method, we applied *GenPCT* to circuit N_k to generate a CTS^{aa} T. Then we used T to break $N'_k \equiv 0$.

A sample of 17 benchmarks is shown in Table III. When compiling this sample we dropped the easy examples solved by all three methods. The first column of Table III lists names of benchmarks. The second column specifies the value of k in N_k and N'_k. The third column gives the number of input variables in N_k (and N'_k) minus[13] the number of latches in M. The fourth column of Table III shows the number of gates in N_k and N'_k (in thousands). The following pair of columns describes the performance of testing driven by the coverage metric above (the number of tests and the run time required to generate and run them). The next two columns provide the results of random testing limited to 100 million tests and the runtime of 5,000 secs.

The final three columns describe the results of CTS^{aa}s. The first column of the three gives the number of iterations we tried when building a CTS^{aa}. Each iteration was a separate run of *GenPCT* generating a different set of tests due to randomization of internal procedures[14]. CTS^{aa}s were built for a projection of N_k on a set of variables V forming an internal cut of N_k. *GenPCT* was run with $tr_1 = 20$ and $tr_2 = 5$. Iterating *GenPCT* went on until $N'_k \equiv 0$ was broken or the number of iterations reached 100. The final two columns describe the total number of tests and run time (over all iterations).

The results of Table III show the high efficiency and effectiveness of CTS^{aa}s on the examples we tried. In particular, for four examples (*kenflashp01*, *kenopp1*, *nusmvguidancep1* and *nusmvtcasp2*) a CTS^{aa} was the only test set to break $N'_k \equiv 0$. Our experiment suggests that one can run the procedure below to check if a bug is overlooked due to misdefining a true property ξ of circuit M. (This procedure does not require knowledge of the "right" property ξ'.) 1) Pick a number k (by an educated guess) to form circuit N_k. 2) Pick a number p of tests to build when proving $N_k \equiv 0$. Run *GenPCT* in a loop until a set T of p tests is generated. 3) Make sure that M correctly behaves on tests of T "as a whole" e.g. by checking that the properties of M related to ξ hold for T.

VIII. Background

As we mentioned earlier, traditional testing checks if a circuit M is correct as a whole. This notion of correctness means satisfying a conjunction of *many* properties of M. For this reason, one tries to spray tests uniformly in the space of all input assignments. To improve the effectiveness of testing, one can try to run many tests at once as it is done in symbolic simulation [4]. To avoid generation of tests that for some reason should be or can be excluded, a set of constraints can be used [13]. Another method of making testing more reliable is to generate tests exciting a particular set of events specified by a coverage metric [16]. Our approach is different from those above in that it is aimed at testing a particular property of M.

The method of testing introduced in [10] is based on the idea that tests should be treated as a "proof encoding" rather than a sample of the search space. (The relation between tests and proofs have been also studied in software verification, e.g. in [7], [8], [1]). In this paper, we take a different point of view where testing becomes a *part* of a formal proof namely the part that performs structural derivations.

Reasoning about SAT in terms of random walks was pioneered in [14]. The centered SSAs we introduce in this paper bear some similarity to sets of assignments generated in derandomization of Schöning's algorithm [5].

The first version of $SeSt$ procedure is presented in report [12]. It has a much tighter integration between the structural part (computation of SSAs) and semantic part (derivation of formula H implied by the original formula). The advantage of the new version of $SeSt$ described in this paper is twofold. First, it is much simpler than $SeSt$ of [12]. In particular, any resolution based SAT-solver that generates proofs can be used to implement the new $SeSt$. Second, the simplicity of the new version makes it much easier to achieve the level of scalability where $SeSt$ becomes practical.

IX. Conclusion

We consider the problem of finding a Complete Test Set (CTS) for a combinational circuit N that is a test set proving $N \equiv 0$. We use the machinery of stable sets of assignments to derive non-trivial CTSs i.e. those that do not include all possible input assignments. Computing a CTS for a large circuit N is inefficient. So, we present a procedure that generates a test set for a "projection" of N on a subset V of variables of N. Depending on the choice of V, this procedure generates a test set CTS^{a} that is an approximation of an CTS or a test set CTS^{aa} that is an approximation of CTS^{a}. We give experimental results showing that CTS^{aa}s can be efficiently computed even for large circuits and are effective in solving verification problems.

X. Acknowledgment

This research was supported in part by NSF grants CCF-1117184 and CCF-1319580.

[12]Even in a random test, the values assigned to the input variables of N_k and N'_k corresponding to state variables of circuit M had to satisfy the predicate specifying the initial states of M (see Subsection VI-B).

[13]The HWMCC-10 benchmarks have only one initial state. So in every test generated in our experiment, the input variables of N_k and N'_k corresponding to the state variables of M were simply set to a constant value.

[14]In particular, a different center was used for the SSA of formula H implied by $F_{N_k} \wedge z$. Formula H was also different in every run of *GenPCT* due to randomization of SAT-calls invoked in *GenCls* (line 2 of Fig. 5).

References

[1] N. Beckman, A. Nori, S. Rajamani, R. Simmons, S. Tetali, and A. Thakur. Proofs from tests. *IEEE Transactions on Software Engineering*, 36(4):495–508, July 2010.

[2] A. Biere, A. Cimatti, E. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using sat procedures instead of bdds. In *DAC*, pages 317–320, 1999.

[3] A. R. Bradley. Sat-based model checking without unrolling. In *VMCAI*, pages 70–87, 2011.

[4] R. Bryant. Symbolic simulation—techniques and applications. In *DAC-90*, pages 517–521, 1990.

[5] E. Dantsin, A. Goerdt, E. Hirsch, R. Kannan, J. Kleinberg, C. Papadimitriou, P. Raghavan, and U. Schöning. A deterministic (22/(k+1))n algorithm for k-sat based on local search. *Theoretical Computer Science*, 289(1):69 – 83, 2002.

[6] N. Eén and N. Sörensson. An extensible sat-solver. In *SAT*, pages 502–518, Santa Margherita Ligure, Italy, 2003.

[7] C. Engel and R. Hähnle. Generating unit tests from formal proofs. In *TAP*, pages 169–188, 2007.

[8] P. Godefroid and N. Klarlund. Software model checking: Searching for computations in the abstract or the concrete. In *Integrated Formal Methods*, pages 20–32, 2005.

[9] E. Goldberg. Testing satisfiability of cnf formulas by computing a stable set of points. In *Proc. of CADE-02*, pages 161–180, 2002.

[10] E. Goldberg. On bridging simulation and formal verification. In *VMCAI-08*, pages 127–141, 2008.

[11] E. Goldberg. Complete test sets and their approximations. Technical Report arXiv:1808.05750 [cs.LO], 2018.

[12] E. Goldberg. Generation of complete test sets. Technical Report arXiv:1804.00073 [cs.LO], 2018.

[13] N. Kitchen and A.Kuehlmann. Stimulus generation for constrained random simulation. In *ICCAD-07*, pages 258–265, 2007.

[14] C. H. Papadimitriou. On selecting a satisfying truth assignment. In *32nd Annual Symposium of Foundations of Computer Science*, pages 163–169, Oct 1991.

[15] Berkeley Logic Synthesis and Verification Group. ABC: A system for sequential synthesis and verification, 2017. http://www.eecs.berkeley.edu/∼alanmi/abc.

[16] S. Tasiran and K. Keutzer. Coverage metrics for functional validation of hardware designs. *IEEE Design Test of Computers*, 18(4):36–45, Jul 2001.

[17] Minisat2.0. http://minisat.se/MiniSat.html.