Verification of Sequential Circuits by Tests-As-Proofs Paradigm

Eugene Goldberg, Mitesh Jain, Panagiotis Manolios
Northeastern University, USA, {eigold,jmitesh,pete}@ccs.neu.edu

***Abstract*—We introduce an algorithm for detection of bugs in sequential circuits. This algorithm is incomplete i.e. its failure to find a bug breaking a property P does not imply that P holds. The appeal of incomplete algorithms is that they scale better than their complete counterparts. However, to make an incomplete algorithm effective one needs to guarantee that the probability of finding a bug is reasonably high. We try to achieve such effectiveness by employing the Test-As-Proofs (TAP) paradigm. In our TAP based approach, a counterexample is built as a sequence of states extracted from proofs that some local variations of property P hold. This increases the probability that a) a representative set of states is examined and that b) the considered states are relevant to property P. We describe an algorithm of test generation based on the TAP paradigm and give preliminary experimental results.**

I. Introduction

Formal methods have lately made impressive progress in verification of sequential circuits. However, these methods still do not scale well enough to handle large designs. So the development of more scalable approaches to sequential verification is an important research direction. One of such approaches is verification by simulation i.e. by applying a set of tests. Simulation is incomplete, which makes it more scalable than formal verification. An obvious downside of simulation though is that it is limited to bug hunting.

To make simulation effective it is crucial to increase the probability that, given a buggy circuit, the part of the search space explored by simulation contains a bug. In the case of sequential verification, making simulation effective is especially challenging for the following reason. Let P be a property of a sequential circuit Φ to be tested. Suppose that Φ is buggy. So there is a sequence $\boldsymbol{s_1}, \ldots, \boldsymbol{s_k}$ of states of Φ such that $\boldsymbol{s_{i+1}}$ is reachable from $\boldsymbol{s_i}$ in one transition, $i = 1, \ldots, k-1$, state $\boldsymbol{s_1}$ is an initial state and $\boldsymbol{s_k}$ falsifies P. Suppose that k is the length of the shortest counterexample breaking property P. This means that no matter how one picks states $\boldsymbol{s_i}$, $i = 1, .., k-1$ they all satisfy property P. To make simulation efficient one has to reduce the set of explored states. But to achieve this goal one must answer the following tough question: *how does one identify the "promising" states if every state reachable from an initial state in less than k steps satisfies P*?

In this paper, we address the challenge above using the Tests-As-Proofs (TAP) paradigm [3], [5]. The essence of TAP is to treat a set of tests not as a sample of the search space but as an encoding of a proof that the property in question holds. So, in a sense, the TAP paradigm reformulates the objective of simulation. Instead of sampling the search space to find a counterexample breaking a property P, a TAP based algorithm looks for a hole in a proof that P holds. A straightforward way of using the TAP paradigm is to generate a set of tests until a counterexample breaking P is found or a test set encoding a proof that P holds is generated. In general, this method is very inefficient because checking if a test set encodes a proof that P holds is computationally hard. There are, however, more practical ways to use TAP. For example, to generate tests for checking if property P holds, one can first prove that a simpler property derived from P holds and then use the tests encoding the obtained proof to verify P itself.

In this paper, we describe a TAP based algorithm called *TapSeq* meant for generation of tests for sequential circuits. Let $P^o(\boldsymbol{s})$ denote the property that every state reachable from state $\boldsymbol{s}$ in one transition satisfies property P of a sequential circuit Φ. (The superscript 'o' stands for 'one'.) *TapSeq* explores only traces $\boldsymbol{s_1}, \ldots, \boldsymbol{s_k}$ where state $\boldsymbol{s_i}$ is extracted from an encoding of a proof that property $P^o(\boldsymbol{s_{i-1}})$ holds. That is *TapSeq* uses local properties $P^o(\boldsymbol{s})$ for building a counterexample breaking the global property P. The idea here is that, on the one hand, these properties are related to property P and on the other hand, they are much easier to prove than P. Importantly, a set of states encoding a proof that the property $P^o(\boldsymbol{s})$ holds is typically a very small subset of all states reachable from $\boldsymbol{s}$ in one transition. So, in a sense, instead of achieving effectiveness of testing by finding "promising" states reachable from $\boldsymbol{s}$ in one transition, *TapSeq* looks for a *representative* subset of states reachable from $\boldsymbol{s}$ in one transition.

This paper is structured as follows. The TAP paradigm is recalled in Section II. In Section III, our algorithm for generation of tests for sequential circuits is described. Finally, Section IV gives some preliminary experimental results.

II. Test-As-Proofs Paradigm

This section is structured as follows. In Subsection II-A, we recall the notions of a resolution proof and a boundary point [6], [4]. The notion of encoding a resolution proof by a set of points [3], [5] is explained in Subsection II-B. Subsection II-C recalls the Tests-As-Proofs paradigm [3], [5] by the example of testing combinational circuits.

A. Resolution and Boundary Points

Definition 1: A **literal** of a Boolean variable v is v itself (positive literal) or the negation of v (negative literal). A **clause** C is a disjunction of literals. We will assume that a clause C

cannot have two literals of the same variable. A **Conjunctive-Normal Form (CNF)** F is a conjunction of clauses. We will also consider F as just a set of clauses. So, for instance, the CNF formula $F \wedge G$ can also be represented as $F \cup G$.

Definition 2: Let X be a set of Boolean variables. An **assignment** $\boldsymbol{q}$ to variables of X is a mapping $Z \rightarrow \{0,1\}$ where $Z \subseteq X$. We will also consider $\boldsymbol{q}$ as a set of value assignments to the individual variables of Z. If $Z = X$, the assignment $\boldsymbol{q}$ is called **complete**. We will also refer to a complete assignment as a **point**.

Definition 3: Let F be a CNF formula and C be a clause. Denote by $\boldsymbol{Vars(F)}$ (respectively $\boldsymbol{Vars(C)}$) the set of variables of F (respectively C). Let $\boldsymbol{q}$ be an assignment. We denote the set of variables assigned in $\boldsymbol{q}$ as $\boldsymbol{Vars(q)}$.

Definition 4: Let v be a Boolean variable. A literal of v is said to be satisfied (falsified) by an assignment to v if it evaluates to 1 (respectively to 0) by this assignment. A clause C is said to be **satisfied** (respectively **falsified**) by an assignment $\boldsymbol{q}$ if a literal of C is satisfied by $\boldsymbol{q}$ (respectively all literals of C are falsified by $\boldsymbol{q}$). A CNF formula F is **satisfied** (respectively **falsified**) by an assignment $\boldsymbol{q}$ if every clause of F is satisfied by $\boldsymbol{q}$ (respectively at least one clause of F is falsified by $\boldsymbol{q}$).

Definition 5: Let $C' \vee v$ and $C'' \vee \overline{v}$ be two clauses such that no variable of $Vars(C') \cap Vars(C'')$ has opposite literals in C' and C''. The clause $C' \vee C''$ is called the **resolvent** of the **parent clauses** $C' \vee v$ and $C'' \vee \overline{v}$. This resolvent is said to be obtained by **resolution** of the parent clauses on v. Clauses $C' \vee v$ and $C'' \vee \overline{v}$ are called **resolvable** on v.

Definition 6: Let F be a CNF formula. A clause C is said to be **derived from** $\boldsymbol{F}$ by a set of resolutions $r_1, \ldots, r_k$ if

- the resolvent of resolution r_k is clause C,
- the parent clauses of resolution r_i, $i = 1, \ldots, k$ are either clauses of F or resolvents of resolutions r_j where $j < i$.

We will call the sequence $r_1, \ldots, r_k$ a **resolution derivation** of clause C from F.

Proposition 1: The resolution proof system based on the operation of resolution is complete in the following sense. Given a CNF formula F and a clause C such that $F \rightarrow C$, there is a resolution derivation of clause C' from F such that $C' \rightarrow C$. In particular, if F is unsatisfiable, one can always derive an **empty clause** from F i.e. a clause that has no literals and so cannot be satisfied. Derivation of an empty clause from F is called a **resolution proof** that F is unsatisfiable.

Definition 7: Let F be a CNF formula and $\boldsymbol{p}$ be a complete assignment to $Vars(F)$. Point $\boldsymbol{p}$ is called a $\boldsymbol{v}$**-boundary point** of F if

- $\boldsymbol{p}$ falsifies F,
- every clause of F falsified by $\boldsymbol{p}$ has variable v.

Proposition 2 below shows that boundary points characterize "mandatory" fragments of resolution proofs.

Proposition 2: Let F be an unsatisfiable formula and $\boldsymbol{p}$ be a v-boundary point of F. Then any resolution proof that F is unsatisfiable contains a resolution r such that

- r is a resolution on variable v,
- the resolvent produced by r is falsified by $\boldsymbol{p}$.

B. Set of Points Encoding a Resolution Proof

Definition 8: Let X be a set of Boolean variables and P be a set of points i.e. complete assignments to X. Let C' and C'' be two clauses such that

- $(Vars(C') \cup Vars(C'')) \subseteq X$,
- C' and C'' are resolvable on variable v.

Resolving C' and C'' on v is said to be **legal** with respect to P if there are points $\boldsymbol{p'}, \boldsymbol{p''} \in P$ such that

- $\boldsymbol{p'}$ falsifies C' and $\boldsymbol{p''}$ falsifies C'',
- $\boldsymbol{p'}$ and $\boldsymbol{p''}$ are different only in the value of v.

Proposition 3: Let clause C be obtained by resolving clauses C' and C'' on variable v. Then points $\boldsymbol{p'}$ and $\boldsymbol{p''}$ make this resolution legal iff both $\boldsymbol{p'}$ and $\boldsymbol{p''}$ falsify C and are different only in variable v.

Definition 9: Let F be an unsatisfiable CNF formula and P be a set of complete assignments to $Vars(F)$. Suppose, there is a resolution proof $R = r_1, \ldots, r_k$ that F is unsatisfiable such that every resolution $r_i, i = 1, \ldots, k$ is legal with respect to P. We will say then that the set of points P **encodes proof** $\boldsymbol{R}$. More generally, we will say that a set of points P encodes an unspecified resolution proof that F is unsatisfiable if there is a resolution proof of unsatisfiability of F encoded by P.

There is a simple but very inefficient procedure [5] for checking if a set of points P encodes a resolution proof that a CNF formula F is unsatisfiable. This procedure starts by making sure that every point of P falsifies F. If not, then F is satisfiable. Otherwise, all resolution operations that are legal with respect to set of points P are performed. If an empty clause is derived then P encodes a proof that F is unsatisfiable. Otherwise, P is too small and needs to be expanded to either include an assignment satisfying F or to encode a proof that F is unsatisfiable.

Obviously, the procedure above is impractical. Unfortunately, no efficient procedure for checking if a set of points encodes a resolution proof is known. On the contrary, the *reverse* procedure of finding a set P encoding a given resolution proof $r_1, \ldots, r_k$ is trivial. The idea of this procedure is to start with an empty set of points P and then add points that makes resolutions of the proof legal. Let r_i be a resolution in which clauses C' and C'' are resolved on variable v producing resolvent C. From Proposition 3 it follows that to make r_i legal one just needs to add to P points $\boldsymbol{p'}$ and $\boldsymbol{p''}$ that falsify C and are different only in value of v. So the upper bound on the size of P is $2 * k$ because one needs two points per resolution. In reality, the size of P may be much smaller because two-point sets legalizing different resolutions r_i and r_j may overlap.

C. Test-as-Proofs Paradigm

In this subsection, we introduce the Tests-As-Proofs (TAP) paradigm by showing how one can use tests to encode a proof of a property of a combinational circuit. Let $N(X, Y, z)$ be a single-output combinational circuit. Here X and Y denote input and internal variables of N respectively and z denotes the output of N. We will assume that the fact the N evaluates

only to 0 means that a combinational property holds. (For instance, N can be the miter of two combinational circuits M', M'' checked for equivalence. Then the fact that N always evaluates to 0 means that M' and M'' are functionally equivalent.) If N evaluates to 1 for some input assignment $\boldsymbol{x}$, then property specified by N does not hold and $\boldsymbol{x}$ is a counterexample.

Let $F_N(X, Y, z)$ be a CNF formula specifying circuit N, i.e. a satisfying assignment of F_N corresponds to a consistent assignment to gates of N and vice versa. Let F denote the formula $F_N \wedge z$. The satisfiability of F means that, for some input assignment, N evaluates to 1 and so there is a bug.

Suppose that F is unsatisfiable and $\Psi = \{r_1, \ldots, r_k\}$ is a resolution proof of that. Let $\boldsymbol{p}$ be a complete assignment to $Vars(F)$. Denote by $inp(\boldsymbol{p})$ be the projection of $\boldsymbol{p}$ onto the set of input variables X. Let $E = \{\boldsymbol{p_1}, \ldots, \boldsymbol{p_m}\}$ be a set of points encoding Ψ. Let $inp(E)$ denote $E = \{inp(\boldsymbol{p_1}), \ldots, inp(\boldsymbol{p_m})\}$. Notice that $inp(\boldsymbol{p_i})$ may be equal to $inp(\boldsymbol{p_j})$ for two different points $\boldsymbol{p_i}$,$\boldsymbol{p_j}$ of E. We will assume that $inp(E)$ does not contain duplicates. We will say that the **set of tests** $T = \{\boldsymbol{x_1}, \ldots, \boldsymbol{x_d}\}$ **encodes proof** Ψ if there is a set of points E encoding Ψ such that $T = inp(E)$. Similarly, set T encodes an unspecified resolution proof if there is a set of points E encoding a resolution proof such that $T = inp(E)$.

As we mentioned in Subsection II-B, the size of a set of points E encoding a proof Ψ is bounded by $2 * |\Psi|$ where $|\Psi|$ is the number of resolutions in Ψ. Since $|inp(E)| \leq |E|$, the same applies to the size of a set of tests encoding Ψ. In reality, as we mentioned above, $|inp(E)|$ may be drastically smaller than $|E|$ because different points of E may have identical projections onto the set of input variables.

The relation between tests and proofs implies that testing can be viewed as finding an encoding of a proof that the property in question holds rather than sampling the search space. We will refer to such a point of view at the **Tests-As-Proofs (TAP) paradigm**. There are numerous ways to use the TAP paradigm in practice. One of them is to build a test set encoding a proof that a property of a circuit holds and apply it in a different situation. (For instance, this set of tests can be used to check if this circuit still has the same property after a modification.)

In Subsection II-B, we outlined a trivial procedure of building a set of points E encoding a known proof Ψ that F is unsatisfiable. However, this procedure cannot guarantee that the set of tests $inp(E)$ extracted from E has high quality. To produce a test set of high-quality one needs to extract them from a set of points E forming a **tight encoding** of Ψ. The intuition here is that the closer a set of points E encoding Ψ to Ψ, the higher the quality of tests $inp(E)$. By proximity of E to Ψ we mean that E makes legal the smallest possible set of resolutions that are not in Ψ.

Informally, building a tight proof encoding means that when looking for points $\boldsymbol{p'}$,$\boldsymbol{p''}$ legalizing resolution of clauses C' and C'' one needs to make $\boldsymbol{p'}$,$\boldsymbol{p''}$ satisfy as many clauses of F as possible. (In particular, if a clause C of F is satisfied by every point of E, then C is redundant in a proof encoded by E. This is because any resolution involving C is illegal with respect to E.) One way to build a tight proof encoding is to require that $\boldsymbol{p'}$,$\boldsymbol{p''}$ are v-boundary points of F where v is the variable on which C' and C'' are resolved. The high quality of tests extracted from boundary points has been confirmed in [5].

III. TAP Based Generation Of Tests For Sequential Circuits

In this section, we describe an algorithm based on the TAP paradigm meant for testing sequential circuits. We will refer to this algorithm as *TapSeq*. This section is structured as follows. In Subsection III-A, some basic definitions of sequential verification are listed. A high-level view of *TapSeq* is given in Subsection III-B. Subsection III-C describes *TapSeq* in more detail.

A. Some Definitions

Definition 10: A sequential circuit Φ is specified by a pair of predicates (I, T) over Boolean variables. Here $T(S, S', Z)$ is the **transition relation** of Φ where S, S' are the sets of present and next state variables respectively, and Z is the set of combinational variables. Predicate $I(S)$ specifies the set of initial states of Φ. We will denote the **input variables** of Φ by X where $X \subseteq Z$.

Definition 11: Let pair $(I(S)$,$T(S, S', Z))$ specify a circuit Φ. A complete assignment $\boldsymbol{s}$ to variables of S (respectively S') is called **a state** (respectively **next state**) of Φ.

Definition 12: Let Φ be a circuit specified by pair (I, T). A sequence of states $\boldsymbol{s_1}, \ldots, \boldsymbol{s_k}$ is called a **trace** if $I(\boldsymbol{s_1}) = 1$ and $\exists Z[T(\boldsymbol{s_i}, \boldsymbol{s_{i+1}}, Z)]$=1 for every i where $1 \leq i \leq k - 1$.

Definition 13: Let Φ be a circuit specified by pair (I, T). The state $\boldsymbol{s}$ is called **reachable** by Φ if there is a trace ending in state $\boldsymbol{s}$. Denote by $\boldsymbol{R(S)}$ a predicate specifying the set of **all reachable states** of Φ. That is $R(\boldsymbol{s}) = 1$ if and only if state $\boldsymbol{s}$ is reachable.

Definition 14: In this paper, we consider the problem of property checking. Let Φ be a circuit specified by pair (I, T). A property of Φ is specified by a predicate $P(S)$ describing the set of states where this property holds (i.e. the set of **good states**). So the predicate $\overline{P}$ specifies the set of **bad states**. For the sake of simplicity, we will refer to the property specified by P as **property** $\boldsymbol{P}$. We will say that property P holds for Φ if $R \wedge \overline{P} \equiv 0$.

Definition 15: Let Φ be a circuit specified by pair (I, T). Let P be a property of Φ and $\boldsymbol{s}$ be a state of Φ. Denote by $\boldsymbol{R^o(s)}$ the set of all states of Φ that are reachable from $\boldsymbol{s}$ in one transition. Denote by $\boldsymbol{P^o(s)}$ the property that holds iff the property P holds for every state of $R^o(\boldsymbol{s})$.

B. High-level View of TapSeq

Let Φ be a sequential circuit specified by pair (I, T). Let P be a property of Φ to be verified. The pseudocode of *TapSeq* is given in Figure 1. *TapSeq* is incomplete i.e. it can build a counterexample breaking P but cannot prove that P holds.

```
// TapSeq returns bug if a reachable bad state is found
// Otherwise TapSeq returns no_bug_found
//
TapSeq(I, T, P){
1  if (I ∧ ¬P ≢ ∅) return(bug);
2  All_states := {init_state(I)};
3  Act_states := All_states;
4  while (Act_states ≠ ∅){
5     Curr_state := pick_state(Act_states);
6     Act_states := Act_states \ {Curr_state};
7     sat := enc_proof(All_states, Act_states,
                       Curr_state, T, P);
8     if (sat) return(bug); }
9  return(no_bug_found);}
```

Fig. 1. *Pseudocode of TapSeq*

For the sake of simplicity we will assume that there is only one state $\boldsymbol{s_1}$ satisfying I i.e. Φ has only one initial state.

First, *TapSeq* checks if property $P^o(\boldsymbol{s_1})$ holds. If not, then there is a bad state $\boldsymbol{s_2} \in R^o(\boldsymbol{s_1})$ and $\boldsymbol{s_1}$,$\boldsymbol{s_2}$ form a counterexample. Otherwise, a resolution proof is generated stating that $P^o(\boldsymbol{s_1})$ holds and a set of states $E^o(\boldsymbol{s_1})$ is extracted from an encoding of this proof. Here $E^o(\boldsymbol{s_1})$ is a subset of $R^o(\boldsymbol{s_1})$. Then the same procedure repeats for the states of $R^o(\boldsymbol{s_1})$. That is for every state $\boldsymbol{s} \in R^o(\boldsymbol{s_1})$, *TapSeq* checks the property $P^o(\boldsymbol{s})$. If it does not hold, then a state $\boldsymbol{s^*} \in R^o(\boldsymbol{s})$ breaks P and $\boldsymbol{s_1}$,$\boldsymbol{s}$,$\boldsymbol{s^*}$ form a counterexample. Otherwise, new states $E^o(\boldsymbol{s})$ are extracted from an encoding of a proof that $R^o(\boldsymbol{s})$ holds.

TapSeq maintains the set *All_states* of all visited states. This allows one to avoid visiting the same state more than once. *TapSeq* terminates in two cases.

- A bad state is reached (property P does not hold).
- No new states are extracted from encodings of proofs of properties $P^o(\boldsymbol{s})$, $\boldsymbol{s} \in All\_states$. In this case, we will say that *TapSeq* reached a **convergence point**.

C. More Detailed Description of TapSeq

TapSeq starts by checking if the initial state breaks property P (line 1 of Figure 1). If it does, then *TapSeq* terminates reporting a bug. Otherwise, variables All_states and Act_states are initialized with the initial state. As we mentioned above, All_states specifies the set of all visited states. Act_states is a subset of All_states. A state $\boldsymbol{s}$ remains in Act_states until the validity of property $P^o(\boldsymbol{s})$ is established.

```
enc_proof(All_states, Act_states, Curr_state, T, P){
1  F = cnf(Curr_state) ∧ T ∧ ¬P';
2  (Ψ, sat) := gen_proof(F);
3  if (sat) return(true);
4  enc_resol(All_states, Act_states, Ψ, F);
5  return(false); }
```

Fig. 2. *Pseudocode of enc_proof*

The main work is done by *TapSeq* in a 'while' loop (lines 4-8). First, *TapSeq* picks a state from Act_states and removes the former from the latter. This state is assigned to variable $Curr\_state$ that is used to specify the state currently processed

```
enc_resol(All_states, Act_states, Ψ, F, Curr_state, T){
1  while (Ψ ≠ ∅) {
2     (C, v) := extract_resolution(Ψ)
3     Ψ := Ψ \ {(C, v)};
4     p := enc_clause(F, C, v, Curr_state);
5     if (p = nil) continue;
6     update_states(All_states, p, T); }}
```

Fig. 3. *Pseudocode of* $encode\_resol$

```
enc_clause(F, C, v, Curr_state){
1  p := find_sat_assgn((F ∪ ¬C) \ F^{v});
2  if (p = nil) return(nil);
3  p := assign_var(p, v, Curr_state);
4  return(p); }
```

Fig. 4. *Pseudocode of enc_clause*

by *TapSeq*. Notice that every state assigned to $Curr\_state$ is reachable from the initial state. Then *TapSeq* checks if property $P^o(Curr\_state)$ holds (line 7). If not, then *TapSeq* reports the presence of a bug. Otherwise, a proof that $P^o(Curr\_state)$ holds is generated. This proof is encoded and new states (if any) are added to All_states and Act_states by procedure *enc_proof*. Then a new iteration begins. Iterations go on as long as Act_states is not empty. Once a convergence point is reached (i.e. Act_states becomes empty), *TapSeq* terminates reporting that no bug was found.

The pseudocode of the *enc_proof* procedure is shown in Figure 2. First, a CNF formula F is formed (line 1) that is satisfiable iff property $P^o(Curr\_state)$ does not hold. The satisfiability of F is checked in line 2. If F is satisfiable, then *enc_proof* terminates (line 3). Otherwise, a proof Ψ of unsatisfiability of F is generated. Resolutions of Ψ are encoded by *enc_resol* procedure shown in Figure 3.

Procedure *enc_resol* loops over resolutions of proof Ψ. First, it extracts a new resolution (C, v) of Ψ and removes it from the latter. Here C is the resolvent and v is the variable on which the parent clauses of C were resolved. Then, a v-boundary point $\boldsymbol{p}$ of F falsifying C is generated by procedure *enc_clause*. From Proposition 3 it follows, that $\boldsymbol{p}$ and the point obtained from $\boldsymbol{p}$ by flipping the value of v legalize the resolution specified by C and v. We want $\boldsymbol{p}$ to be a v-boundary point to make our proof encoding *tight*. If $\boldsymbol{p}$ does not exist, *enc_resolutions* starts a new iteration. Otherwise, procedure *update_states* is called to update sets All_states and Act_states.

The pseudocode of procedure *enc_clause* is shown in Figure 4. This procedure computes a v-boundary point of formula F that falsifies a resolvent clause C. This is done by finding an assignment satisfying formula $F \cup \overline{C} \setminus F^v$ where F^v is the set of clauses of F containing variable v. Notice that if $\boldsymbol{p}$ satisfies $F \cup \overline{C} \setminus F^v$ then it satisfies all the clauses of F but some clauses containing variable v. In other words, $\boldsymbol{p}$ is a v-boundary point of F. After computing $\boldsymbol{p}$, the value of variable v is set in $\boldsymbol{p}$ (line 3). If $v \notin S$, then the value of v is set arbitrarily. Otherwise, v is assigned the same value as

```
update_states(All_states, p, T){
1  (s,x) := extract_state_input(p);
2  s* := find_next_state(s, x, T);
3  if (s* ∈ All_states) return;
4  All_states := All_states ∪ {s*};
5  Act_states := Act_states ∪ {s*}; }
```

Fig. 5. *Pseudocode of update_states*

```
RandAlg(I, T, P, max_tries, max_length){
1  if (I ∧ ¬P ≢ ∅) return(bug);
2  Curr_state := set_init_state(I);
3  length := 0; tries := 0;
4  while (tries ≤ max_tries) {
5    if (length > max_length) {
6      length := 0; tries++;
7      Curr_state := set_init_state(I);
8      continue;}
9    F := cnf(Curr_state) ∧ T ∧ ¬P;
10   if (satisf(F)) return(bug);
11   x := gen_rand_input(X);
12   Curr_state := next_state(T, x, Curr_state);
13   length++; }
14 return(no_bug_found); }
```

Fig. 6. *Algorithm for generation of counterexamples randomly*

in $Curr\_state$. This is done to guarantee that the new states generated by *update_states* are reachable from $Curr\_state$ in one transition.

The fact that one uses only v-boundary points that agree with the values of $Curr\_state$ means that proof Ψ is encoded only partially. Namely, this encoding does not legalize resolutions on variables of S. This is done to simplify *TapSeq*. We are going to fix this problem in future versions of *TapSeq*.

Figure 5 shows the pseudocode of procedure *update_states*. First, the assignments ($\boldsymbol{s}$,$\boldsymbol{x}$) to variables of S and X (i.e. present state and input variables) are extracted from a v-boundary point found by procedure *enc_clause*. Then the transition relation T is used to compute the state $\boldsymbol{s}^*$ to which circuit Φ switches from state $\boldsymbol{s}$ under the input assignment $\boldsymbol{x}$. If $\boldsymbol{s}^*$ is a new state, it is added to All_states and Act_states.

IV. Experimental Results

In this section, we describe two experiments conducted to evaluate the performance of *TapSeq*. This section is structured as follows. In Subsection IV-A, we describe an algorithm of random test generation that we compared with *TapSeq*. Some details of the implementation of *TapSeq* we used in experiments are given in Subsection IV-B. The first and second experiments are described in Subsections IV-C and IV-D respectively.

A. Random Algorithm We Used in Experiments

In this subsection, we describe an algorithm of random test generation we used in the first experiment. We will refer to this algorithm as $RandAlg$. The pseudocode of $RandAlg$ is shown in Figure 6. The set of counterexamples generated by $RandAlg$ is controlled by parameters *max_tries* and *max_length*. The value of *max_tries* limits the number of generated counterexamples while *max_length* sets the limit to the number of states in a counterexample. The length of the current counterexample and the number of counterexamples generated so far are specified by variables *length* and *tries* respectively.

$RandAlg$ maintains variable $Curr\_state$ specifying a state reachable from the initial state that is currently processed by $RandAlg$. At the beginning, $Curr\_state$ is set to the initial state (line 2). The main work is done in the 'while' loop (lines 4-13). If the value of *length* exceeds *max_length*, a new counterexample is started and the value of *tries* is incremented (lines 5-8). Otherwise, $RandAlg$ checks if $Curr\_state$ satisfies property P. If not, then $RandAlg$ returns value *bug*. Otherwise, $RandAlg$ randomly generates an assignment $\boldsymbol{x}$ to input variables X (line 11). Then $\boldsymbol{x}$ is used to generate a new state that is the state to which the circuit switches from state $Curr\_state$ under input assignment $\boldsymbol{x}$ (line 12). After that, the length of the current counterexample is incremented and a new iteration begins.

B. Implementation of TapSeq

In the pseudocode of *TapSeq* given in Figure 1, we did not clarify in what order states were extracted from Act_states in the 'while' loop. The two extremes are depth-first and breadth-first orders. The depth-first order is to first process the state of Act_states the is the farthest from the initial state (in terms of transitions). On the contrary, the breadth-first order, is to first process the state that is the closest to the initial state. In the breadth-first variant of *TapSeq*, states are processed one time frame after another. We assume here that i-th time frame consists of the states of All_states that can be reached from the initial state in i transitions. That is, in the breadth-first variant, a state of Act_states of i-th time frame is processed only after every state of every j-th time frame where $j < i$ has been processed and removed from Act_states. Obviously, by imposing a particular order of extracting states from Act_states one can also have modifications of *TapSeq* that are different from the two extremes above. *In this paper, we report results of a breadth-first implementation of TapSeq.*

In the experiments, we ran two versions of *TapSeq*: randomized and non-randomized. The difference between these versions is in finding boundary points used to encode proofs. In the randomized version, the internal SAT-solver called to find boundary points had some randomization in its decision making. Namely, the phase of every 10-th decision assignment was chosen randomly. The reason for such randomization is explained in Subsection IV-C.

C. First Experiment: Comparison of TapSeq with $RandAlg$

The objective of the first experiment was to compare *TapSeq* with $RandAlg$. In this comparison we used 314 buggy benchmarks of the HWMCC-10 competition. 78 benchmarks of this set were trivial: the initial state did not satisfy the property to be verified. We excluded them from consideration.

The results of the experiment on non-trivial benchmarks are summarized in Table I.

TABLE I

Solving non-trivial buggy HWMCC-10 benchmarks. Maximum number of visited states is limited to 1,000,000 for $RandAlg$ and 40,000 for TapSeq

number of benchmarks	$RandAlg$ solved	*TapSeq* unrandomized		*TapSeq* randomized		*TapSeq* total solved
		solved.	converg.	solved	converg.	
236	**43**	35	94	59	8	**69**

The first column of Table I shows the number of non-trivial benchmarks used in the first experiment. The second column gives the number of benchmarks solved by $RandAlg$. The parameters max_tries and max_length of $RandAlg$ were set to 10,000 and 100 respectively. That is $RandAlg$ generated up to 10,000 counterexamples of length 100. (So the total number of visited states was limited by 1,000,000. The counterexample length of 100 was large enough to solve any benchmark solved by *TapSeq*.) For every benchmark, the time limit for $RandAlg$ was set to 900 seconds.

The next four columns show results of unrandomized and randomized versions of *TapSeq*. For both versions, the number of visited states (i.e. the size of All_states) was limited by 40,000 and the time limit was set to 180 seconds. For either version, we report the number of solved benchmarks and the number of benchmarks where a convergence point was reached. (Recall that a convergence point is reached by *TapSeq* when the set Act_states becomes empty before a bug is found.) The last column gives the number of benchmarks solved by at least one version of *TapSeq*.

The results of Table I show that for many benchmarks the unrandomized version of *TapSeq* reached a convergence point. This means that *TapSeq*, in its current form, needs some way to escape early convergence. In this experiment, we achieved this goal by randomizing *TapSeq* as described in Subsection IV-B. The randomized version of *TapSeq* solved more benchmarks and reached a convergence point only for 8 benchmarks. Overall, the experiment showed that *TapSeq* outperformed $RandAlg$ solving more benchmarks (69 versus 43) with much stricter limit on the number of visited states.

D. Second Experiment: Bounded Model Checking and TapSeq

The objective of the second experiment was to show that some benchmarks solved by *TapSeq* were hard for Bounded Model Checking (BMC) [2]. In this experiment, we used a BMC tool built on top of the Aiger package [7] and Picosat [1], a well-known SAT-algorithm. In general, BMC is good at detecting shallow bugs but struggles to find deeper bugs even if these bugs are easy to detect. This point is illustrated by results of the second experiment shown in Table II. Notice that we do not claim that the current implementation of *TapSeq* outperforms BMC. The latter performed extremely well on shallow benchmarks of the set we used in the first experiment while *TapSeq* could not solve many of them. We just want to emphasize the promise of *TapSeq* in finding deep bugs.

The first column of Table II gives benchmark names. The next two columns show the time taken by the BMC tool to find a counterexample and the length of this counterexample. The last two columns provide the same information for *TapSeq*. The examples of Table II have the largest counterexample length among the benchmarks solved by *TapSeq*. These are also the examples (among those solved by *TapSeq*) where the BMC tool had the longest run time. *TapSeq* significantly outperforms the BMC tool on these examples. Interestingly, the first five benchmarks were also easy for $RandAlg$ (but $RandAlg$ failed to solve the 'visbakery' benchmark).

TABLE II

Some benchmarks that are hard for BMC and easy for TapSeq

benchmarks	*BMC*		*TapSeq*	
	time (s.)	cex length	time (s.)	cex length
pdtswvroz10x6p0	118	58	1.2	88
pdtswvsam6x8p0	116	48	7.7	48
pdtswvtma6x6p0	95	57	0.8	57
pdtswvtma6x4p0	70	57	0.9	57
pdtswvroz8x8p0	65	48	1.1	72
visbakery	925	59	28	61

V. Conclusions

In this paper, we introduce *TapSeq*, a new algorithm for generation of tests for sequential circuits based on the Tests-As-Proofs (TAP) paradigm. *TapSeq* forms a counterexample from encodings of proofs of local properties that are versions of the property to be verified. The preliminary experimental results allows one to conclude that

- *TapSeq* convincingly outperforms a random algorithm;
- *TapSeq* significantly outperforms a BMC tool on some benchmarks with non-shallow bugs.

These results suggest that algorithms based on the TAP paradigm can be used for finding deep bugs.

Our future research will be focused in the following directions.

1) In this paper, we consider an algorithm mimicking forward model checking. That is one generates a set of states reachable from an initial state trying to find a state violating the property in question. Instead, one can try to mimic a backward model checking algorithm building a set of states from which a bad state is reachable. The objective here is to reach an initial state. Moreover, one can try to design an algorithm that combines forward and backward model checking. Intuitively, such an algorithm can be much more effective in finding a bug because a counterexample is built from both initial and bad states.

2) The other important direction for research is to find a better way to avoid reaching a convergence point i.e. the situation where no new states are generated. In this paper, we achieved this goal by randomizing the part of *TapSeq* that performed proof encoding. This solution is not quite satisfactory because it leads to generating too many states per time frame and hence makes it much harder for *TapSeq* to find a deep bug. (In particular, the benchmarks with non-shallow bugs shown in Table II were solved by the unrandomized version of *TapSeq*.)

VI. Acknowledgments

This work was supported in part by C-FAR, one of six centers of STARnet, an SRC program sponsored by MARCO

and DARPA. It was also partially funded by NSF grant CCF-1117184.

REFERENCES

[1] A. Biere. Picosat essentials. *JSAT*, 4(2-4):75–97, 2008.
[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] E. Goldberg. On bridging simulation and formal verification. In *VMCAI-08*, pages 127–141, 2008.
[4] E. Goldberg. Boundary points and resolution. In *Proc. of SAT*, pages 147–160. Springer-Verlag, 2009.
[5] E. Goldberg and P. Manolios. Generating high-quality tests for boolean circuits by treating tests as proof encoding. In *TAP-10*, pages 101–116. Springer-Verlag, 2010.
[6] E. Goldberg, M. Prasad, and R. Brayton. Using problem symmetry in search based satisfiability algorithms. In *DATE '02*, pages 134–141, Paris, France, 2002.
[7] AIGER package,http://fmv.jku.at/aiger/.