Generating High-Quality Tests for Boolean Circuits by Treating Tests as Proof Encoding*

Eugene Goldberg and Panagiotis Manolios

Northeastern University, USA {eigold,pete}@ccs.neu.edu

Abstract. We consider the problem of test generation for Boolean combinational circuits. We use a novel approach based on the idea of treating tests as a proof encoding rather than as a sample of the search space. In our approach, a set of tests is complete for a circuit N, and a property p, if it "encodes" a formal proof that N satisfies p. For a combinational circuit of k inputs, the cardinality of such a complete set of tests may be exponentially smaller than 2^k. In particular, if there is a short resolution proof, then a small complete set of tests also exists. We show how to use the idea of treating tests as a proof encoding to directly generate high-quality tests. We do this by generating tests that encode mandatory fragments of *any* resolution proof. Preliminary experimental results show the promise of our approach.

1 Introduction

Although formal verification has made significant progress, simulation, due its scalability, is still the main workhorse of functional verification. An obvious drawback of simulation is that it only samples the search space and so may miss some bugs. Making simulation *complete* (i.e guaranteeing the lack of bugs) at the same time keeping the number of tests reasonably small is a very exciting goal.

Recent results [6] show that finding small complete test sets is actually possible. These results are based on the idea of treating a test set as an encoding of a formal proof (that the required property holds) rather than a sample of the search space. We will refer to this concept as Treating Tests as a Proof Encoding (TTPE). In particular, it was shown that to encode a resolution proof of k resolutions one needs at most $2k$ tests. Importantly, a test set encoding a formal proof is *complete* (in the sense that no bug can be missed) and may be very small. Such a test set may be exponentially smaller than a trivial complete test set of 2^n tests (where n is the number of inputs of N).

In this paper, we use TTPE for verification of combinational circuits. The generic problem here is to show that a single-output circuit N always evaluates to 0 or to find a bug, an input assignment $\boldsymbol{x}$ such that $N(\boldsymbol{x})$=1. In principle, TTPE can be used for both proving that $N \equiv 0$ and for showing that N is buggy by generating a sequence of tests until we either encode a proof or find a bug. In more detail, suppose we generated tests $\boldsymbol{x_1}$, ..., $\boldsymbol{x_{k-1}}$, but they do not encode a

* This research was funded in part by NASA Cooperative Agreement NNX08AE37A.

proof and they have not found a bug. We generate a new test, $\boldsymbol{x_k}$, and check if $N(\boldsymbol{x_k}) = 1$. If so, N is buggy. Otherwise, we check if the set $\{\boldsymbol{x_1}, \ldots, \boldsymbol{x_k}\}$ encodes a resolution proof. If it does, then $N \equiv 0$. Otherwise, we continue generating tests. This is a simplified overview of the procedure *ProofByTesting* described in Section 5.

An obvious question is: when does a set of tests encode a resolution proof? The key idea is to use tests to enable certain resolution steps. Thus, a set of tests encodes a resolution proof if the set of resolution steps allowed by the tests includes all the steps of the proof. A full account can be found in Section 4.

Unfortunately, no *efficient* procedure for checking if a set of tests encodes a resolution proof is currently known. However, we develop an efficient variation of *ProofByTesting* meant *only* for showing that N is buggy. Generation of high-quality tests for checking if N has a bug (*i.e.*, evaluates to 1 for some test) is *the problem we address in this paper*. Our approach is based on the idea of generating tests that encode "mandatory" fragments of a resolution proof. Given a CNF formula F, a particular class of complete assignments called boundary points of F specify mandatory resolutions of a proof that F is unsatisfiable [5]. In this paper, we show that boundary points can be used for generation of high quality tests.

The idea of extracting tests from boundary points is the first contribution of our paper. The second contribution is showing that TTPE can be used for building good tests indirectly, *i.e.*, without generating an explicit proof. Our third contribution is in giving experimental evidence that tests extracted from boundary points have high quality.

Given a CNF formula $F(v_1, \ldots, v_n)$, a $lit(v_i)$-boundary point $\boldsymbol{p}$ is an unsatisfying assignment such that all clauses of F falsified by $\boldsymbol{p}$ share the same literal $lit(v_i)$ (where $lit(v_i)$ is v_i or $\overline{v}_i$). Importantly, a $lit(v_i)$-boundary point $\boldsymbol{p}$ *mandates* a resolution on variable v_i [5] (any proof that F is unsatisfiable has to have a resolution on v_i eliminating $\boldsymbol{p}$ as a $lit(v_i)$-boundary point). So one can use boundary points to encode mandatory fragments of a resolution proof that the design property specified by F holds.

It is an open question whether a resolution proof can be encoded by boundary points alone (*i.e.*, whether resolutions mandated by boundary points always constitute a resolution proof). It was shown experimentally in [5] that for well structured proofs, the share of resolutions mandated by boundary points is high (90-100%). This implies that by generating boundary points one has a good chance to encode a proof or a large part of it.

Studying the relation of tests and proofs in propositional logic is important for at least three reasons. First, propositional logic plays an outstanding role in hardware verification. Second, it is also used in software verification. The state-of-the art SMT-solvers are based on propositional SAT-solvers. The latter are also extensively used in software verification systems like Alloy [10], CBMC [11], Java pathfinder [12]. Third, in many cases testing works well in verification of sequential circuits and programs. This implies that the TTPE approach may be applicable to logics more complex than propositional.

Since the method we introduce in this paper is heavily based on the TTPE approach, we describe the latter in Sections 2-5. We use a simple example to explain our definitions. Then in Section 6 we describe how tests are extracted from boundary points. In Section 7, we relate our approach with hardware testing based on the stuck-at fault model and with mutation-based testing. Finally, we give some experimental results and make conclusions.

2 Example

In this section, we introduce an example we will use extensively throughout the paper. Figure 1 shows a circuit called *miter* that is meant for equivalence checking of combinational circuits M' and M''. In a miter, circuits M' and M'' share the same set of inputs. Besides, the outputs of M' and M'' feed an XOR gate (in our example, it is gate G_6). The circuits M' and M'' are functionally inequivalent iff their miter evaluates to 1 (in this case there is an input assignment for which M' and M'' produce different values and so the XOR gate evaluates to 1).

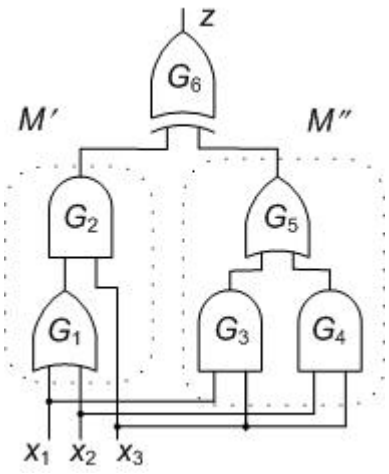

Fig. 1. Miter N of functionally equivalent circuits M' and M''

The miter N shown in Figure 1, checks for equivalence circuits M' and M'' implementing the expressions $(x_1 \vee x_2) \wedge x_3$ and $(x_1 \wedge x_3) \vee (x_2 \wedge x_3)$ respectively. Since M' and M'' are functionally equivalent, N always evaluates to 0. The conventional wisdom is that to prove that N implements constant 0 by simulation one has to generate $2^3 = 8$ tests. As we show later, for the circuit N of Figure 1 there is a *complete* test set of only 5 tests. This test set is complete in the sense it encodes a resolution proof that a CNF formula F specifying the query "Does $N(\boldsymbol{x})$ evaluate to 1 for some $\boldsymbol{x}$?" is unsatisfiable. These 5 tests are extracted from boundary points of F.

3 Some Basic Definitions

In this paper, we specify combinational circuits by CNF formulas. This section gives some relevant definitions.

Definition 1. *A **literal** lit(v_i) of a Boolean variable v_i is either v_i itself (the **positive literal** of v_i) or the negation of v_i denoted as $\overline{v}_i$ (the **negative literal** of v_i).*

Definition 2. *A **clause** is the disjunction of literals where no two (or more) literals of the same variable can appear. A clause consisting of only one literal is*

called ***unit****. **A CNF formula** is the conjunction of clauses. We will also view a CNF formula as **a set of clauses**.*

Definition 3. *Given a CNF formula $F(v_1, \ldots, v_n)$, a **complete assignment** (also called a **point**) is a mapping $\{v_1, \ldots, v_n\} \rightarrow \{0, 1\}$. Given a complete assignment $\boldsymbol{p}$ and a clause C, denote by $C(\boldsymbol{p})$ the value of C when its variables are assigned as in $\boldsymbol{p}$. A clause C is **satisfied** (respectively **falsified**) by a complete assignment $\boldsymbol{p}$, if $C(\boldsymbol{p}) = 1$ (respectively $C(\boldsymbol{p}) = 0$).*

Definition 4. *Given a CNF formula F, a **satisfying assignment $\boldsymbol{p}$** is a complete assignment satisfying every clause of F. The **satisfiability problem (SAT)** is to find a satisfying assignment for F or to prove that it does not exist.*

4 Encoding Resolution Proofs

In this section, we describe how a circuit is represented by a CNF formula and recall the basics of the resolution proof system. Then we describe how one can encode a resolution proof by complete assignments. (This is done as in [6] but without using the machinery of stable sets of points.) Finally, we define the notion of proof encoding in terms of tests.

4.1 CNF representation of a circuit

Typically, finding out if a property of a combinational circuit M holds reduces to checking if a single-output circuit (derived from M) implements constant 0. (For instance, proving that combinational circuits M and M' are functionally equivalent comes down to checking if the miter N of M' and M'' always evaluates to 0, see Section 2).

Let N be a single-output circuit of gates $G_1, \ldots, G_m$ where the output of G_m is also the output of N. Let the inputs of N, the outputs of gates of $G_1, \ldots, G_{m-1}$ and the output of G_m be denoted by $X = \{x_1, \ldots, x_r\}$, $Y = \{y_1, \ldots, y_{m-1}\}$ and z respectively. Let F_N be a CNF formula specifying N that is obtained from F using regular Tseitsin's transformations [9]. Namely, $F_N = F_{G_1} \wedge \ldots \wedge F_{G_m}$ where F_{G_i} is a CNF formula satisfied (respectively falsified) by the consistent (respectively the inconsistent) assignments to the pins of G_i.

Example 1. For the circuit N of Figure 1, $X = \{x_1, x_2, x_3\}$, $Y = \{y_1, \ldots, y_5\}$ (where y_i specifies the output of gate $G_i, i = 1, \ldots, 5$) and variable z specifies the output of G_6. The formula F_N specifying N can be represented as $F_{G_1} \wedge \ldots \wedge F_{G_6}$. Formula F_{G_1}, for instance, specifies the OR gate G_1 and is equal to $(x_1 \vee x_2 \vee \overline{y}_1)$ $\wedge$ $(\overline{x}_1 \vee y_1)$ $\wedge$ $(\overline{x}_2 \vee y_1)$. For example, clause $(x_1 \vee x_2 \vee \overline{y}_1)$ is falsified by the inconsistent assignment $x_1 = 0, x_2 = 0, y_1 = 1$ and so is F_{G_1}.

The problem of finding an input assignment that sets the output of N to 1 comes down to checking the satisfiability of the formula $F(X, Y, z) = F_N \wedge z$. A complete assignment $\boldsymbol{p}$ to the variables of F can be represented as $(\boldsymbol{x},\boldsymbol{y},z^*)$ where

$\boldsymbol{x}$,$\boldsymbol{y}$,z^* are assignments to X, Y and z respectively. The part of $\boldsymbol{p}$ consisting of the assignments to the input variables (*i.e.*, the part $\boldsymbol{x}$) is called a *test*. We will denote it by $inp(\boldsymbol{p})$. If an assignment $\boldsymbol{p}$ satisfies all the clauses of F_N (but may falsify the unit clause z of F), the part ($\boldsymbol{y}$,z^*) of $\boldsymbol{p}$ is called the *correct execution trace* for the test $\boldsymbol{x}$=$inp(\boldsymbol{p})$.

In general, a complete assignment $\boldsymbol{p}$ falsifies clauses of F_N. In such a case, ($\boldsymbol{y}$,z^*) can be interpreted as a *faulty execution trace*. This means that at least one gate G_i of N produces the value that is different from the one implied by the input values of G_i.

Example 2. Let $\boldsymbol{p}$=($x_1=0, x_2=1, x_3=1, y_1=1, y_2=1, y_3=0, y_4=1, y_5=1, z=0$) a complete assignment to the variables of F_N specifying circuit N of Figure 1. The test $\boldsymbol{x}$ corresponding to $\boldsymbol{p}$ is $inp(\boldsymbol{p})$=($x_1=0, x_2=1, x_3=1$). Since $\boldsymbol{p}$ assigns consistent values to all 6 gates of N it satisfies all the clauses of F_N (the entire formula F_N is given below in Example 3). So, in this case, ($y_1=1, y_2=1, y_3=0, y_4=1, y_5=1, z=0$) is the correct execution trace for the test $\boldsymbol{x}$.

Now, let $\boldsymbol{p'}$=($x_1=0, x_2=1, x_3=1, y_1=0, y_2=0, y_3=0, y_4=1, y_5=1, z=1$). Point $\boldsymbol{p'}$ assigns consistent values to all the gates of N but gate G_1 ($\boldsymbol{p'}$ falsifies the clause ($\overline{x}_2 \vee y_1$) of F_{G_1} given in Example 1). Point $\boldsymbol{p'}$ specifies the same test $\boldsymbol{x}$ as above and a faulty execution trace ($y_1=0, y_2=0, y_3=0, y_4=1, y_5=1, z=1$).

4.2 Resolution proofs

In this subsection, we recall the basics of the resolution proof system for propositional logic [2].

Resolution is a sound and complete proof system that has only one derivation rule called resolution.

Definition 5. *Let clauses C',C'' have the opposite literals of variable v_i (and no opposite literals of other variables). The* ***resolvent*** *C of C' and C'' on variable v_i is the clause with all the literals of C' and C'' but those of v_i. The clause C is said to be obtained by a* ***resolution operation*** *on v_i. C' and C'' are called the* ***parent clauses****.*

Definition 6. *([2]) Let F be an unsatisfiable formula. Let $\{R_1, \ldots, R_k\}$ be a set of clauses such that*

- *each clause R_i is obtained by a resolution operation where a parent clause is either a clause of F or R_j, $j < i$;*
- *clauses R_i are numbered in the derivation order;*
- *R_k is an empty clause.*

Then the set of k resolutions that produced the resolvents $R_1, \ldots, R_k$ is called a ***resolution proof*** *that F is unsatisfiable.*

Example 3. Here we describe a resolution proof for the unsatisfiable CNF formula $F = F_N \vee z$ where $F_N = F_{G_1} \wedge \ldots \wedge F_{G_6}$ specifies the circuit N of Figure 1. To describe subformulas F_{G_i} one needs clauses $C_i, i = 1, \ldots, 19$ given below. Let C_{20} denote the unit clause z. Then $F = C_1 \wedge \ldots \wedge C_{20}$.

$F_{G_1} = C_1 \wedge C_2 \wedge C_3$, $C_1 = x_1 \vee x_2 \vee \overline{y}_1$, $C_2 = \overline{x}_1 \vee y_1$,$C_3 = \overline{x}_2 \vee y_1$.
$F_{G_2} = C_4 \wedge C_5 \wedge C_6$, $C_4 = \overline{y}_1 \vee \overline{x}_3 \vee y_2$, $C_5 = y_1 \vee \overline{y}_2$, $C_6 = x_3 \vee \overline{y}_2$.
$F_{G_3} = C_7 \wedge C_8 \wedge C_9$, $C_7 = \overline{x}_1 \vee \overline{x}_3 \vee y_3$, $C_8 = x_1 \vee \overline{y}_3$, $C_9 = x_3 \vee \overline{y}_3$.
$F_{G_4} = C_{10} \wedge C_{11} \wedge C_{12}$, $C_{10} = \overline{x}_2 \vee \overline{x}_3 \vee y_4$, $C_{11} = x_2 \vee \overline{y}_4$, $C_{12} = x_3 \vee \overline{y}_4$.
$F_{G_5} = C_{13} \wedge C_{14} \wedge C_{15}$, $C_{13} = y_3 \vee y_4 \vee \overline{y}_5$, $C_{14} = \overline{y}_3 \vee y_5$,$C_{15} = \overline{y}_4 \vee y_5$.
$F_{G_6} = C_{16} \wedge C_{17} \wedge C_{18} \wedge C_{19}$, $C_{16} = y_2 \vee \overline{y}_5 \vee z$, $C_{17} = \overline{y}_2 \vee y_5 \vee z$,
$C_{18} = y_2 \vee y_5 \vee \overline{z}$, $C_{19} = \overline{y}_2 \vee \overline{y}_5 \vee \overline{z}$.

A resolution proof R that F is unsatisfiable is given in Figure 2 as a DAG whose nodes are shown as ovals. Each non-leaf node corresponds to a resolution operation over the parent clauses specified by the preceding nodes. The proof R is obtained by a version of a SAT-solver with conflict driven clause learning [7]. R is partitioned into three chains of resolutions corresponding to the three conflicts (backtracks) that occurred when solving the formula F. Each chain describes derivation of a conflict clause shown in a dotted oval.

Empty ovals correspond to resolvents that are used only once right after they are generated. All the resolvents of the proof R can be easily reproduced by performing the sequence operations specified by the graph of Figure 2. For instance, the first empty oval of the leftmost chain corresponds to resolving clause $C_8 = x_1 \vee \overline{y}_3$ with $C_{13} = y_3 \vee y_4 \vee \overline{y}_5$ (on variable y_3) and producing the resolvent $x_1 \vee y_4 \vee \overline{y}_5$. The latter is then resolved with the clause $C_{11} = x_2 \vee \overline{y}_4$ (on variable y_4) producing the resolvent $x_1 \vee x_2 \vee \overline{y}_5$ corresponding to the next empty oval and so on. Eventually, the conflict clause $C_{21} = y_1 \vee \overline{z}$ is derived. In the middle resolution chain, the conflict clause $C_{22} = x_3 \vee \overline{z}$ is generated. Finally, the empty conflict clause C_{23} is derived in the rightmost resolution chain.

4.3 Proof encodings in terms of points

In this subsection, we explain what it means for a set of points to encode a resolution proof.

Definition 7. *Let C', C'' be two clauses and $\boldsymbol{p'}$ and $\boldsymbol{p''}$ be two complete assignments such that*

- *$\boldsymbol{p'}$ and $\boldsymbol{p''}$ are only different in the value of a variable v_i*
- *$C'(\boldsymbol{p'}) = C''(\boldsymbol{p''})$=0 and $C'(\boldsymbol{p''}) = C''(\boldsymbol{p'})$=1*

*Then $\boldsymbol{p'}$ and $\boldsymbol{p''}$ are said to **legalize** the resolution of $\boldsymbol{C'}$,$\boldsymbol{C''}$ on v_i.*

The two conditions of Definition 7 imply that C',C'' have opposite literals of exactly one variable (which is v_i). This means that clauses C',C'' indeed can be resolved on v_i. Intuitively, the existence of points $\boldsymbol{p'}$ and $\boldsymbol{p''}$ satisfying the conditions of Definition 7 means that a clause (the resolvent of C',C'') is implied by $C' \wedge C''$. In the approach based on the notion of a stable set of points [6], this implication is a theorem (that can be easily proved).

Example 4. Let us consider the first resolution operation of the proof R described in Example 3 (*i.e.*, resolution over $C_8 = x_1 \vee \overline{y}_3$ and $C_{13} = y_3 \vee y_4 \vee \overline{y}_5$). Let point $\boldsymbol{p'}$ be $(x_1 = 0, .., y_3 = 1, y_4 = 0, y_5 = 1, ..)$ (the values of the missing variables can be chosen arbitrarily). Let $\boldsymbol{p''}$ be obtained from $\boldsymbol{p'}$ by flipping the value of y_3. Then $C_8(\boldsymbol{p'})$=$C_{13}(\boldsymbol{p''})$=0 and $C_8(\boldsymbol{p''})$=$C_{13}(\boldsymbol{p'})$=1. So $\boldsymbol{p'}$ and $\boldsymbol{p''}$ legalize the resolution over C_8 and C_{13}.

Definition 8. *Let F be an unsatisfiable CNF formula and P be a set of points. Let $R = \{R_1, \ldots, R_k\}$ be a resolution proof. We will say that P* ***encodes proof*** *R if each of k resolutions of R is legalized by some points $\boldsymbol{p'}$ and $\boldsymbol{p''}$ of P.*

Informally, the fact that P encodes a resolution proof means that the former is large enough to legalize resolutions comprising a proof that F is unsatisfiable. Definitions 7 and 8 imply that to encode a resolution proof $R = \{R_1, \ldots, R_k\}$ one needs a set P of at most $2k$ points (two points for each resolution). In reality, this number may be smaller because the same point of P may participate in legalization of more than one resolution operation.

Definition 9. *Let N be a single-output circuit and T be a set of tests $\{\boldsymbol{x_1}, \ldots, \boldsymbol{x_s}\}$. We will say that T encodes a resolution proof R that $F_N \wedge z$ is unsatisfiable if there is a set of points $P = \{\boldsymbol{p_1}, \ldots, \boldsymbol{p_m}\}$ such that P encodes R and each test $\boldsymbol{x_i}$ of T is the input part of a point $\boldsymbol{p_j}$ of P.*

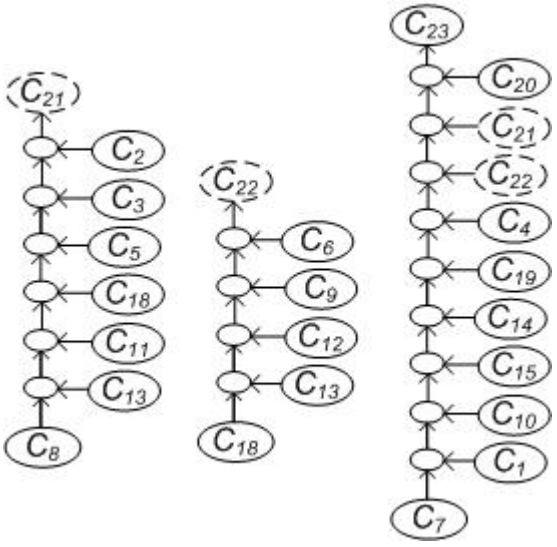

Fig. 2. A resolution proof that $F = F_N \wedge z$ is unsatisfiable

Definitions 8 and 9 imply that there is always a test set encoding a resolution proof R that is at most two times the size of R.

Example 5. The proof R of Figure 2 consists of 19 resolutions. So there is a set of at most 38 points encoding it. (In this small example, the size of the proof is larger than 8, which is the total number of assignments to 3 inputs. However, it is not unusual to have a proof, say, of 10^6 resolutions for a circuit with 1000 inputs.) Since different points $\boldsymbol{p'}$ $\boldsymbol{p''}$ may have the same input part one may need less than 8 tests to encode R.

For instance, it can be shown that the following set

of 5 tests $\boldsymbol{x_1}$=(x_1=0, x_2=1, x_3=0), $\boldsymbol{x_2}$=(x_1=0, x_2=0, x_3=1), $\boldsymbol{x_3}$=(x_1=1, x_2=1, x_3=0), $\boldsymbol{x_4}$=(x_1=1, x_2=0, x_3=1), $\boldsymbol{x_5}$=(x_1=0, x_2=1, x_3=1) encodes the proof R. That there is a set of points P legalizing the 19 resolutions of R such that the number of different input parts of points from P is 5.

5 Test Generation by Proof Encoding

In this section, we describe a procedure (called *ProofByTesting*) that, given a single-output combinational circuit N checks if $N \equiv 0$. Our objective here is threefold. First, we want to illustrate the point that the idea of TTPE is applicable to both buggy and correct circuits. (N is buggy when $N(\boldsymbol{x})$=1 for some input assignment $\boldsymbol{x}$.) Second, even though *ProofByTesting* is not practical, it illustrates the point that one can have a complete test set that is smaller than 2^n (where n is the number of inputs of N). Third, the TCBP procedure described in Subsection 6.3 is a variation of *ProofByTesting* made efficient by removing checks that a set of points encodes a proof. (So this variation is limited to finding bugs in N).

```
c the procedure checks if N ≡ 0
ProofByTesting(N)
{ P = ∅, F = F_N ∧ z;
while(true)
  {p = gen_pnt(F);
  x = inp(p);
  if (N(x) == 1) return(no);
  P = P ∪ {p};
  while (true)
    {(C,Exst)= new_legal_res(P,F);
    if (Exst==false) break;
    if (C == empty) return(yes);
    F = F ∪ {C}; }}}
```

Fig. 3. A procedure for generating tests in the process of encoding a proof

Pseudocode of the *ProofByTesting* procedure is shown in Figure 3. First, it builds CNF formula F as described in Subsection 4.1. In the outer loop, *ProofByTesting* generates a point $\boldsymbol{p}$ and extracts its input part $\boldsymbol{x}$. (We assume here that the same point is not generated more than once). If $\boldsymbol{x}$ is a counterexample, the procedure returns *no*. Otherwise, $\boldsymbol{p}$ is added to the set of points P and the inner loop begins. In this loop *ProofByTesting* checks if P encodes a resolution proof.

First, *ProofByTesting* arbitrarily picks a new resolvent C obtained by a resolution legalized by points of P. (We assume here that no resolvent is generated if it is implied by an existing clause of F.) If all legal resolvents have been generated, *ProofByTesting* leaves the inner loop to generate a new point $\boldsymbol{p}$. Otherwise, it checks if C is an empty clause. If it is, the answer *yes* is returned. (The set of resolvents added to F contains a resolution proof and so $N \equiv 0$. This proof is encoded by points of P. The input parts of points from P form a test set encoding a resolution proof.) If C is not empty, it is added to F and a new iteration of the inner loop begins.

6 Extracting Tests from Boundary Points

In this section, we recall the definition and some properties of boundary points and describe the idea of using such points for proof encoding. Then we give a simple algorithm (called *TCBP*) for generation of tests extracted from boundary points. *TCBP* is a variation of *ProofByTesting* described in the previous section. Instead of running inefficient checks if a set of points encodes a resolution proof, *TCBP* generates points that encode mandatory parts of a resolution proof. (Since *TCBP* does not encode a complete proof, it can be used only for finding bugs.)

6.1 Definition of boundary points and some useful properties

Definition 10. *Denote by Unsat($\boldsymbol{p}$,F) the set of clauses of a CNF formula F falsified by a complete assignment $\boldsymbol{p}$.*

Definition 11. *Given a CNF formula F, a complete assignment $\boldsymbol{p}$ is called a lit(v_i)-boundary point, if Unsat($\boldsymbol{p}$,F) $\neq \emptyset$ and every clause of Unsat($\boldsymbol{p}$,F) contains literal lit(v_i).*

Example 6. The point $\boldsymbol{p}$=($x_1=0, x_2=1, x_3=0, y_1=1, y_2=0, y_3=1, y_4=0, y_5=1, z=1$) falsifies only the clauses $C_8 = x_1 \vee \overline{y}_3$ and $C_9 = x_3 \vee \overline{y}_3$ of the formula F of Example 3. These two clauses share literal $\overline{y}_3$. So $\boldsymbol{p}$ is a $\overline{y}_3$-boundary point.

Definition 12. *Denote by Bnd_pnts(F) the set of all boundary points of F.*

Definition 13. *Let $\boldsymbol{p}$ be a complete assignment. Denote by flip($\boldsymbol{p}$,v_i) the point obtained from $\boldsymbol{p}$ by flipping the value of v_i.*

The proposition below explains why studying boundary points is important.

Proposition 1. *([5]) If Bnd_pnts(F) = $\emptyset$, then F is unsatisfiable.*

Proposition 1 implies that for a satisfiable formula F, *Bnd_pnts*(F) $\neq \emptyset$. In particular, it is not hard to show [5] that if $F(\boldsymbol{p'})$=0, $F(\boldsymbol{p''})$=1 and $\boldsymbol{p''}$= *flip*($\boldsymbol{p'}$,v_i), then $\boldsymbol{p'}$ is a *lit*(v_i)-boundary point. (This explains the name "boundary point".) Another interesting fact is that if $\boldsymbol{p'}$ is a v_i-boundary point, the point $\boldsymbol{p''}$= *flip*($\boldsymbol{p'}$,v_i) is either a satisfying assignment or a $\overline{v}_i$-boundary point [5]. So for an unsatisfiable formula all boundary points come in pairs. We will refer to $\boldsymbol{p'}$ and $\boldsymbol{p''}$ as *symmetric* v_i-boundary and $\overline{v}_i$-boundary points.

Let $\boldsymbol{x}$ be the test corresponding to a *lit*(v_i)-boundary point $\boldsymbol{p}$ (*i.e.*, $\boldsymbol{x}$ = *inp*($\boldsymbol{p}$)) where v_i is not variable z. Then the part ($\boldsymbol{y}$,z) of $\boldsymbol{p}$ specifies a faulty execution trace for test $\boldsymbol{x}$. Namely, at least one gate of N whose output/input variable is specified by v_i produces the wrong output value (which is the negation of the value implied by the input values of this gate).

Example 7. Consider the $\overline{y}_3$-boundary point $\boldsymbol{p}$=($x_1=0, x_2=1, x_3=0, y_1=1, y_2=0, y_3=1, y_4=0, y_5=1, z=1$) of Example 6. It specifies test $\boldsymbol{x}$=($x_1=0, x_2=1, x_3=0$) and the execution trace ($y_1=1, y_2=0, y_3=1, y_4=0, y_5=1, z=1$). In this trace, the AND gate G_3 (whose output is described by y_3) produces the wrong output value 1 for the input values $x_1=0$, $x_3=0$. (All the other gates produce output values implied by the input values of these gates.)

6.2 Encoding resolutions proofs by boundary points

Let $\boldsymbol{p'}$ and $\boldsymbol{p''}$ be symmetric v_i-boundary and $\overline{v}_i$-boundary points of F. It is not hard to show [5] that any clause C' falsified by $\boldsymbol{p'}$ can be resolved on variable v_i with any clause C'' falsified by $\boldsymbol{p''}$. This resolution produces a clause that is falsified by both $\boldsymbol{p'}$ and $\boldsymbol{p''}$ and does not have variable v_i. Then $\boldsymbol{p'}$ and $\boldsymbol{p''}$ are not $lit(v_i)$- boundary points of $F \wedge C$. Adding a clause to F can only eliminate some boundary points (but cannot produce new ones). So $Bnd\_pnts(F \wedge C) \subset Bnd\_pnts(F)$. We will refer to this process of removing boundary points by adding clauses implied by F as *boundary point elimination*. (Note that adding C to F may also eliminate $lit(v_i)$-boundary points different from $\boldsymbol{p'}$ and $\boldsymbol{p''}$.)

Let $R_1, \ldots, R_k$ be a resolution proof where R_k is an empty clause. Note that $Bnd\_pnts(F \wedge R_k) = \emptyset$. (By definition, if p is a $lit(v_i)$-boundary point, every clause falsified by $\boldsymbol{p}$ has to have at least one literal, *i.e.*, $lit(v_i)$.) This means that every $lit(v_i)$-boundary point $\boldsymbol{p}$ of the initial formula F is eventually eliminated. Then there is a resolvent R_{m+1} such that $\boldsymbol{p}$ is a $lit(v_i)$-boundary point for $F \wedge R_1 \wedge \ldots \wedge R_m$ but not for $F \wedge R_1 \wedge \ldots \wedge R_{m+1}$. It was shown in [5] that a $lit(v_i)$-boundary point $\boldsymbol{p}$ is eliminated in the proof only by a resolution on variable v_i. In other words, a $lit(v_i)$-boundary point *mandates* a resolution on v_i. This fact is the foundation for using boundary points to encode resolution proofs.

If $\boldsymbol{p'}$ and $\boldsymbol{p''}$ are symmetric v_i-boundary and $\overline{v}_i$-boundary points and they are eliminated by adding to F the resolvent of C' and C'' on v_i, then $\boldsymbol{p'}$ and $\boldsymbol{p''}$ *legalize* this resolution (because $\boldsymbol{p'}$, $\boldsymbol{p''}$ and C' and C'' satisfy both conditions of Definition 7). If $\boldsymbol{p'}$ and $\boldsymbol{p''}$ are symmetric boundary points legalizing a resolution on v_i, *every proof* has to contain a resolution on v_i (but not necessarily the resolution of C' and C'') In the general case, *i.e.*, when $\boldsymbol{p'}$ and $\boldsymbol{p''}$ are not symmetric boundary points, they may legalize a resolution on variable v_i even if there are proofs that have no resolutions on v_i. So proof encodings by boundary points are much closer related to proofs than encodings by arbitrary points.

It is not clear yet if one can encode an entire resolution proof using only boundary points. (It may be the case that some resolution operations of a proof can be legalized only by non-boundary points). However, the experimental study of [5] showed that for well structured proofs the ratio of resolutions that could be legalized by boundary points was close to 100%. This implies that (at least for the formulas of [5]) using boundary points one can encode an entire proof or a large part thereof.

Example 8. Every resolution operation of the proof R described in Example 3 eliminates a boundary point (that has not been eliminated by previous resolutions). The set of 5 tests given in Example 5 was actually built by a program that extracted tests from boundary points eliminated by resolutions of R. For example, the $\overline{y}_3$-boundary point $\boldsymbol{p'}$=$(x_1 = 0, x_2 = 1, x_3 = 0, y_1 = 1, y_2 = 0, y_3 = 1, y_4 = 0, y_5 = 1, z = 1)$ introduced in Example 6 and the symmetric y_3-boundary point $\boldsymbol{p''}$=$flip(\boldsymbol{p'}, y_3)$ are eliminated by the first resolution of R. (The latter resolves clauses $C_8 = x_1 \vee \overline{y}_3$ and $C_{13} = y_3 \vee y_4 \vee \overline{y}_5$ on variable y_3.) Points $\boldsymbol{p'}$ and $\boldsymbol{p''}$ legalize this resolution.

6.3 Extraction of tests from boundary points

```
c F = F_N ∧ z
TCBP(F,T)
{ count=0; T = ∅;
while (count < thresh)
  {v_i= pick_var(F);
  (ans,p) = BndPnt(F,v_i,lim);
  if (ans < success) continue;
  else count++;
  x=inp(p); T = T∪ {x};
  if (simulate(x,F) == yes)
   return (found);
  eliminate(p,F);}
return(not_found);}
```

Fig. 4. *TCBP* procedure

The procedure for extraction of tests from boundary points called *TCBP* (Testing based on Computation of Boundary Points) is shown in Figure 4. Given a single-output circuit N, specified by a CNF formula F_N, the *TCBP* procedure generates a set of tests to check if N evaluates to 1. (This comes down to checking the satisfiability of CNF formula $F = F_N \wedge z$.) *TCBP* terminates if a test is found for which N evaluates to 1 or if the number of generated tests exceeded a threshold. *TCBP* records the set of all generated tests. (The reason is as follows. Our experiments showed that due the high quality of tests generated by *TCBP*, even if a test is unsuccessful for N it may detect a bug in a modified version N. This modification may correspond, for example, to a wrong design change.)

The main work is done in the 'while' loop. First a variable v_i of F is picked randomly. Then the procedure *BndPnt* is called to find a $lit(v_i)$-boundary point. *lim*. If no boundary point is found by *BndPnts* within time limit *lim*, a new iteration of the 'while' loop is started. (In our experiments, *lim* was set to 10 sec.) Otherwise, a test $\boldsymbol{x}$ is constructed as the input part of the boundary point $\boldsymbol{p}$ found by *BndPnt*. This test is used for simulation. In terms of SAT, simulation comes down to adding the set U of unit clauses encoding test $\boldsymbol{x}$ (*i.e.*, satisfied by the assignments of $\boldsymbol{x}$) to the formula F and running Boolean Constraint Propagation (BCP).

```
BndPnt(F,v_i,lim)
{G = F \ (F_{v_i} ∪ F_{v̄_i}).
(ans,p)=sat(G,lim);
return((ans,p));}
```

Fig. 5. *BndPnt* procedure

If the circuit N is deterministic, then BCP over the formula $F = F_N \wedge U \wedge z$ results in assigning a value to the output variable z. If z=1 (respectively z= 0), the test $\boldsymbol{x}$ is a counterexample (respectively not a counterexample). In experiments, we used faults that may make the behavior of a gate of N non-deterministic. Then, variable z may not get assigned after BCP is over. If this is the case, a SAT-solver was used to finish the instance (*i.e.*, to prove that F was unsatisfiable if its input variables were assigned according to $\boldsymbol{x}$ or to find a satisfying assignment). If such an assignment was found, this meant that $\boldsymbol{x}$ detected the fault that the gate with non-deterministic behavior produced the wrong output value (*i.e.*, produced the negation of the value implied by the input assignments for the gate without the fault).

If $\boldsymbol{x}$ fails to detect a fault, the $lit(v_i)$-boundary point $\boldsymbol{p}$ from which $\boldsymbol{x}$ was extracted is eliminated by adding a resolvent on variable v_i. (No particular heuristic

was used to pick the pair of clauses to be resolved). The test $\boldsymbol{x}$ is added to the set of tests T and a new iteration of the 'while' loop starts.

The procedure for generation of a $lit(v_i)$-boundary point (called *BndPnt*) is given in Figure 5. First, the CNF formula G is obtained from F by removing the clauses having literal v_i (denoted by F_{v_i}) and $\overline{v}_i$ (denoted by $F_{\overline{v}_i}$). Then a SAT-solver *sat* is called to check if G is satisfiable. If a satisfying assignment $\boldsymbol{p}$ is found, it means that one of the following three possibilities occurred: a) $F_{v_i}(\boldsymbol{p}) = 0$ and hence $\boldsymbol{p}$ is a v_i-boundary point (because it falsifies only clauses of F that have literal v_i); b) $F_{\overline{v}_i}(\boldsymbol{p})=0$ and $\boldsymbol{p}$ is a $\overline{v}_i$-boundary point; c) $F(\boldsymbol{p})=1$ and $\boldsymbol{p}$ is a satisfying assignment (never happened in our experiments).

If the SAT-solver *sat* fails to find a boundary point it may mean that the run time exceeded *lim* or that formula G (and hence F) is unsatisfiable. (The latter never happened in our experiments because we considered only satisfiable formulas F there).

7 Some Background

Methods of combining test generation and formal methods have been studied in many papers (e.g. [4, 8] to name a few). In this section, for the lack of space, we only mention the work directly related to generation of tests extracted from boundary points (namely, generation of tests detecting hardware faults [1] and software mutations [3]).

Identification of defective chips is usually done by running tests that detect faults of a particular fault model [1]. In many cases, such a fault model may have little to do with what really happens on a defective chip. It is just used because tests detecting faults of this model are good at finding real faults. The most popular fault model of that kind is the stuck-at model. It describes the situation when a line of a gate is stuck at a constant value 0 or 1.

There is a tight relation between tests detecting stuck-at faults in a circuit M' and boundary points of a CNF formula $F = F_N \wedge z$. Here N is the miter of circuits M' and M'' (like the one shown in Figure 1) where M'' is a copy of circuit M'. Variable z specifies the output of N.

Consider, for example, a stuck-at-0 fault ϕ on the output line of AND gate G' of M'. Let y_i,y_j be the input variables of G' and y_k be its output variable. Denote by M'_ϕ the circuit M' with fault ϕ. A test $\boldsymbol{x_\phi}$ detecting ϕ (*i.e.*, detecting that $y_k \equiv 0$) has to assign y_i and y_j to 1. Then the gate G' of M'_ϕ and its counterpart G'' in M'' produce different output values (0 and 1 respectively).

Let $\boldsymbol{p}$=($\boldsymbol{x_\phi}$,$\boldsymbol{y}$,1) be the point where ($\boldsymbol{y}$,1) is the correct execution trace for test $\boldsymbol{x_\phi}$ for the miter of M'_ϕ and M''. Then $\boldsymbol{p}$ is an y_k-boundary point for the formula F. Indeed, since $y_i = 1$,$y_j = 1$,$y_k = 0$ in $\boldsymbol{p}$, the latter falsifies clause $C = \overline{y}_i \vee \overline{y}_j \vee y_k$ of the formula $F_{G'}$ specifying gate G'. Since ($\boldsymbol{y}$,1) is the *correct* execution trace for the miter of M'_ϕ and M'', all the other gates of the miter N of M' and M'' are assigned correctly. It means that $\boldsymbol{p}$ satisfies all the clauses of the formula F but C.

Summarizing, one can view introduction of stuck-at faults as a way to generate boundary points of formula F (describing the miter of two *correct* copies of the same circuit). Importantly, the stuck-at fault model has been successfully used in industry for three decades, which serves as an indirect evidence of the quality of tests extracted from boundary points (at least for hardware testing).

The method of introducing mutations into a program has a lot of similarity with identification of defective chips based on fault models. In particular, the mutation operator replacing a logical subexpression with constant 'true' or 'false' introduces a "stuck-at fault" into a program. Another interesting similarity is that fault injection into a circuit (respectively introduction of a mutation into a program) can be used to identify redundancy in the circuit (respectively in the program). It is not hard to show that the absence of $lit(v_i)$- boundary points in a CNF formula F means that the clauses of F with variable v_i can be removed from F without changing its satisfiability. One can conjecture that similarly to hardware manufacturing testing, using mutations is just a way to produce tests that could have been extracted from some sort of boundary points of a formula describing the "correct" program (*i.e.*, the one without a mutation).

8 Experimental Results

In this section, we give some preliminary experimental results. Our intention here is just to check the idea of using tests extracted from boundary points by considering a simple example that mimics a hard real-life problem. This problem is verification of arithmetic devices embedded into control logic. (The famous Pentium bug was caused by failing to solve an instance of this problem.) In the experiments, we tested the miter N of a faulty and correct circuits denoted by M_f and M respectively (an example of a miter is shown in Figure 1). The circuit M contained a large arithmetic component. The circuit M_f was a copy of M with a fault introduced into the arithmetic component.

In the experiments we wanted to demonstrate the following two properties of tests built by TCBP (*i.e.*, extracted from boundary points). First, even though generation of such tests takes time their quality is much higher than that of random tests. Second, even unsuccessful TCBP tests meant for verification of N (*i.e.*, tests for which N evaluates to 0) have a high chance to find a bug in a modified version of N if this modification is small. (In particular, we extracted tests from boundary points of an *unsatisfiable* formula F describing the miter N of two *identical* copies of circuit M. These tests were very effective in finding bugs when one copy of M was replaced in N with a faulty circuit M_f. We do not report this part of experiment for the lack of space.) From a practical point of view this means that the cost of tests generated by TCBP can be amortized over many design steps (due to reusing tests generated at previous design steps).

As a fault, we considered adding a literal to a clause of the CNF formula F_M specifying circuit M. (We found the bugs of this kind to be very easy to introduce and very hard to detect.) The resulting formula F_{M_f} specified the faulty circuit M_f. Adding a literal to a clause describing a gate, makes the latter behave non-

deterministically. (Consider the clause $C = \overline{y}_i \vee \overline{y}_j \vee y_k$ of F_M requiring that when inputs y_i and y_j of an AND gate G_k are set to 1, the output of G_k specified by y_k is 1 too. After adding literal $\overline{y}_m$ to C, where y_m specifies the output of gate G_m, the output of G_k can take an arbitrary value, when $y_i = y_j = 1$ and $y_m = 0$.) A test $\boldsymbol{x}$ is considered to have detected a fault in gate G_k of M_f, if the miter of M_f and M evaluates to 1 when G_k produces the *wrong* value, *i.e.*, the negation of the value implied by the values of its inputs.

We compared *TCBP* with two extremes of functional verification: random testing (an instance of pure simulation) and checking the satisfiability of the miter N by a SAT-solver (an instance of pure formal verification). For testing the satisfiability of F in one SAT check we used SAT-solvers *Satzilla*, *Precosat* and *Mxc* that won the first, second and third places in the SAT-2009 competition [13] in the industrial category for satisfiable formulas. We also used *Precosat* in *TCBP* for finding boundary points.

We ran two experiments that were meant to probe the regions that are good and bad for random testing. In the first experiment, the circuit M consisted only of an arithmetic component (good for random testing). In the second experiment, M consisted of an arithmetic component feeding an input of a multi-input AND gate (bad for random testing).

The results of the first experiment are described in Table 1. In this experiment, circuit M implemented the functionality of a medium bit of a 128-bit multiplier. (The size of the CNF formula F specifying the miter of M and M_f was about 114,000 variables and 437,000 clauses. The formulas of Table 2 had about the same size.) We cherry picked 5 faults that were easy for random testing and hard for SAT-solving. The runtimes of SAT-solvers are shown in columns 2,3,4. (The timeout was set to 1 hour.) When using random testing, for every fault we generated 10 sets of random tests, the two columns of "random_testing" showing the average run time and average test set size.

The last three columns of Table 1 show the results of *TCBP*. To mitigate the influence of chance in picking boundary points, *TCBP* was given 10 tries to find tests for the set of 5 faults. Before generating tests for fault number i (i=2,..,5), the tests generated for faults 1,..i-1 (*in the same try*) were applied. If an old test detected the fault no new tests were generated. In such a case, the number of tests generated for this fault in this try was set to 0 and only the time taken by simulation of old tests was charged. The "old tst." column shows the number of times a fault was detected by an old test $\boldsymbol{x}$ generated to detect a previous fault. (To be tried for a fault number i, test $\boldsymbol{x}$ did not have to be successful for a fault number j, $j < i$.). For example, value 10 for fault 3 means that in every try (out of 10), fault 3 was detected by a test generated before to detect fault 1 or 2. In the last two columns, for each fault, the average run time and average number of tests are given. The latter is computed over the tries where no old test was successful and new tests had to be generated to detect this fault. (For that reason no number of tests is given for faults 3 and 5 that were detected by old tests in all 10 tries.)

The results of Table 1 show that *TCBP* performed much better than SAT-solvers (but worse than random testing). It was able to reuse tests generated for previous faults and it was faster even without reusing tests (e.g. for fault 1).

Table 1. Testing a circuit derived from a 128 bit multiplier

Flt.	*Satzilla*	*Precosat*	*Mxc*	*random testing*		*TCBP*		
num.		s.	s.	times.	#tsts	old tst.	time (s.)	#tsts
1	>1h	>1h	>1h	4.8	334	0	346	87
2	>1h	>1h	>1h	7.8	656	1	328	83
3	>1h	2,450	>1h	0.2	15	10	0.4	n/a
4	>1h	931	>1h	2.8	212	7	50	43
5	>1h	1,596	52	4.6	205	10	0.4	n/a

Table 2 shows the results of testing the miter N of M and M_f when M was made up of the circuit implementing a medium bit of a 128-bit multiplier which fed an input of a 24-input AND gate. The output of this AND gate was the output of M. (The other 23 inputs of this AND gate were primary inputs of M.) The idea was to make it much harder for random tests to propagate faults to the output. The experiments indeed showed that *random testing failed on every fault* we introduced (with the threshold of 10^6 tests per fault). Table 2 contains the performance of *Precosat* and *TCBP* on a subset of 17 faults (we discarded the faults that were easy for both *Precosat* and *TCBP*). The time limit was set to 5 hours. (We did not use *Satzilla* and *Mxc* because they performed much worse than *Precosat* not being able to detect the majority of faults within the time limit while *Precosat* found all faults.)

Table 2. Testing a 17-fault set for a circuit with arithmetic and logic components

	Precosat	*TCBP*	
	time (s)	time (s)	#tests
total	54,115	2,919	562
average	3,183	172	33
median	935	8	3

As before, we used 10 tries to generate tests for the 17-fault set. So the total, median and average values of Table 2 were computed for the averages over 10 tries. When testing fault number i, i=2,..17 we also (as in Experiment 1) checked if this fault can be detected by a test generated for a fault 1,.., i-1 *in the same try*. (Typically, one had to generate tests only for 3-5 faults out of 17. The other faults were detected by old tests.)

The results of Table 2 show that on the set of hard faults we used, *TCBP* was almost 20 times faster (even though it employed the same version of *Precosat* to generate boundary points). Again, this can be attributed to a) reusing old tests; b) finding a counterexample faster even when *TCBP* had to generate new tests. Reusing of old tests explains the small median values of *TCBP* for the run time and for the number of tests. For many faults, *TCBP* generated new tests in a small number of tries (if any).

9 Conclusions

We introduced a test generation procedure based on the TTPE framework (Treating Tests as a Proof Encoding). Given a circuit and a property, this procedure finds tests that encode mandatory fragments of *any* resolution proof that the circuit satisfies the property. These tests are extracted from boundary points, and this process does not require a proof. The successful demonstration of these ideas implies that the study of proof systems more complex than resolution and for logics more expressive than propositional logic, may lead to new methods for generating high-quality tests for both hardware and software verification.

References

1. Abramovici, M., Breuer, M., Friedman, D.: Digital Systems Testing and Testable Design. John Wiley & Sons, (1994)
2. Bachmair, L., Ganzinger, H.: Resolution theorem proving. In Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. I, chapt. 2, pp. 19-99. North-Holland (2001)
3. Budd, T.A., DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In: 7th ACM SIGPLAN- SIGACT Symposium on Principles of Programming Languages, pp. 220-233. Las Vegas, Nevada, (1980)
4. Engel, C., Hähnle, R.: Generating unit tests from formal proofs. In: Gurevich, Y., Meyer, B. (eds.), TAP-2007. LNCS, vol. 4454 , pp. 169-188. Springer, Heidelberg (2007)
5. Goldberg, E.: Boundary points and resolution. In: Kullmann, O. (eds.), SAT-2009. LNCS, vol. 5584 , pp. 147-160. Springer, Heidelberg (2009)
6. Goldberg, E.: On bridging simulation and formal verification. In: Logozzo, F., Peled, D., Zuck, L.D. (eds.), VMCAI-2009. LNCS, vol. 4905 , pp. 127-141. Springer, Heidelberg (2009)
7. Marques-Silva, J., Sakallah, K.: Grasp - a new search algorithm for satisfiability. In: International conference on computer-aided design, pp. 220-227, Washington, DC, USA, (1996)
8. Satpathy, M., Butler, M.J., Leuschel, M., Ramesh, S.: Automatic testing from formal specifications. In: Gurevich, Y., Meyer, B. (eds.), TAP-2007. LNCS, vol. 4454 , pp. 95-113. Springer, Heidelberg (2007)
9. Tseitin, G.S.: On the complexity of derivation in the propositional calculus. In: Zapiski nauchnykh seminarov LOMI, vol. 8, pp. 234-259 (1968)
10. Alloy system, http://alloy.mit.edu/community
11. CProver, http://www.cprover.org/cbmc
12. JavaPathfinder, http://babelfish.arc.nasa.gov/trac/jpf
13. SAT 2009 competition, http://www.satcompetition.org/2009