HyAC: A Hybrid Structural SAT Based ATPG for Crosstalk

Xiaoliang Bai, Sujit Dey
Dept. of ECE, University of California, San Diego CA 92093
Angela Krstić
Dept. of ECE, University of California, Santa Barbara CA 93106

Abstract

As technology evolves into the deep sub-micron era, signal integrity problems are growing into a major challenge. An important source of signal integrity problems is the crosstalk noise generated by coupling capacitances between wires. Test vectors that activate and propagate crosstalk noise effects are becoming an essential part of design verification and manufacturing test. However, deriving such vectors is a complex task. In this paper, we propose HyAC, a fast yet accurate hybrid ATPG method targeting multiple-aggressor induced crosstalk errors. Given a victim and a set of aggressors, the proposed ATPG method searches for test vectors to activate and propagate a crosstalk error for the victim. Due to logic constraints, it may not be possible to trigger all aggressors simultaneously. Therefore, firstly we use an implication graph (IG) that consists of logic variables and structural information to check for logic conflicts. If the current set of aggressors is not feasible, our algorithm automatically searches for the next-best subset of aggressors (resulting in the largest noise). After a set of feasible aggressors is identified, we use a modified PODEM [21] algorithm to search for test vectors. This hybrid structural SAT-based ATPG method inherits advantages from both Boolean–satisfiabilitity based methods and structural-based methods to achieve flexibility and efficiency. We demonstrate the accuracy, high quality, and run time efficiency of HyAC through experiments conducted on several benchmark circuits as well as a circuit from a commercial processor.

1. Introduction

With dense interconnect, low supply voltage, fast clock frequency and large coupling-to-ground capacitance ratio, crosstalk noises are becoming perilous to ignore in the deep submicron era [1]. Various crosstalk analysis models were proposed [2, 3], at the same time, different noise avoidance and repairing techniques were developed [4, 5, 6]. However, in some situations, crosstalk noise avoidance or repair solutions are expensive and may even cause unnecessary design iterations. For a cost-sensitive design, superfluous over-design to safeguard signal integrity may hurt its performance and profitability. Consequently, simulation-based verification and manufacturing test are crucial for the error-free operation of chips.

Even though designers may be able to perform exhaustive simulation and fine-tuning on the small modules, for a complicated design like System-on-Chip (SoC), module-to-module and global-to-local signal interference become difficult to address. Since the neighboring block and top-level routing information may not be available in the early design cycle, the design needs to be verified after modules are put together. Circuit-level simulators like SPICE provide high accuracy, but suffer from severe capacity limitations and time-consuming performance. Hence, an effective but small set of patterns becomes important for simulation-based verification. An efficient test vector generation tool targeting potential crosstalk errors will help to generate short but high-quality simulation vectors and reduce the verification time.

In deep submicron technologies, process variation exists not only from die-to-die but also across a single die. The high density of interconnects leads to interconnect-related failures. Hence, manufacturing test for crosstalk noise induced errors should be applied to ensure the quality of the manufactured chips.

Therefore, both simulation-based verification and manufacturing test require high-quality vectors to be efficiently generated for large circuits. The quality of the vectors is determined by their ability to activate a large noise effect and propagate the effect to primary outputs (POs) or to memory elements like flip-flops.

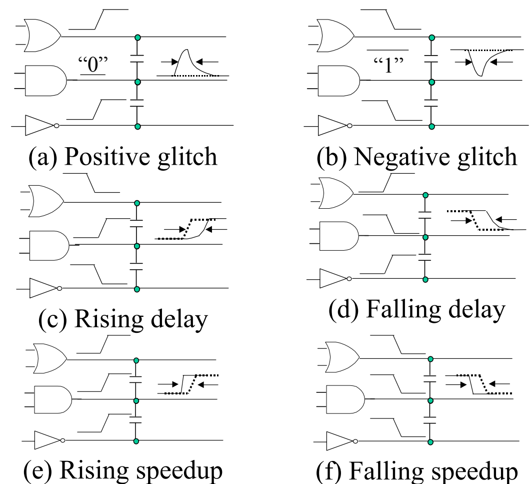

Figure 1: Crosstalk noise effects

Crosstalk noises can cause functional and/or timing errors. Possible error effects include: positive glitch, negative glitch, rising delay, falling delay and speedups, as

[1] This work is supported by MARCO/DARPA Gigascal Silicon Research Center (GSRC)

 0-7803-8106-8/03 $17.00 Copyright 2003 IEEE

shown in Figure 1. A wire whose signal is under concern is a *victim*. An aggressor is a net that has a significant coupling capacitance to the victim net and whose transitions can affect the signal of the victim net.

To trigger these error effects, different test generation methods have been developed. Structural-based ATPG methods for crosstalk noise induced by a single aggressor have been developed in [7, 8, 9]. Recently, timed test pattern generation method for switch failures in domino CMOS has been proposed [10]. In [11], dynamic timing analysis was used to generate test patterns for delay faults in static CMOS circuit. Even though the methods in [10, 11] can consider multiple aggressors, the computationally expensive circuit-level timing simulations hurt the scalability of the proposed methods. In [12], a Genetic Algorithm based test generation for crosstalk induced delay faults has been proposed.

Boolean satisfiability (SAT) based ATPG methods for *stuck-at* faults [13, 14, 15, 17, 18] exhibit advantages such as uniformity and flexibility. A SAT-based crosstalk noise analysis method was proposed in [16] that considers both temporal and functional information. However, the high complexity of general purpose SAT-solver makes this method impractical. On the other hand, when combined with structural information, SAT-based methods can be as efficient as or even faster than structural-based methods, as shown for stuck-at faults [17, 18].

Paper Outline and Contributions

In this paper, we propose HyAC, a **Hy**brid structural SAT-based deterministic **A**TPG method for **C**rosstalk noise. It targets multiple-aggressor induced logic and transition faults. The crosstalk fault library is provided by using other tools such as layout extraction, timing analysis, noise estimation and process variation analysis. For each fault, a victim and a set of aggressors should be identified such that the aggressors' signal transition may cause a large glitch/delay on the victim and lead to an erroneous operation of the circuit under test. However, it is not always possible to trigger all the aggressors. The goal of the ATPG tool is to trigger a set of prominent aggressors and sensitize a propagation path by a sequence of vectors.

Given a victim and a set of aggressors, the structural SAT-based ATPG method first checks for potential logic conflicts by using an implication graph (IG). The conflicts are results of logic relation among the victim, the aggressors, as well as mandatory assignments on the propagation path(s). If a logic conflict(s) exists, the proposed algorithm will automatically search for a subset of compatible aggressors based on weights assigned to aggressors. As a result, a set of potentially feasible aggressors will be proposed as objectives for justification. In the next phase, we use a modified PODEM algorithm to search for vectors that can justify the required logic value for the victim, activate the aggressors and sensitize a propagation path. This hybrid structural SAT-based ATPG method combines advantages from both Boolean-satisfiabilitity-based methods and structural-based methods to achieve flexibility and efficiency. SPICE simulations were performed to evaluate the quality of the generated vectors. To demonstrate the scalability of the proposed ATPG methodology, it was also applied to a set of benchmark circuits, and a large logic cone from an industrial processor (Xtensa[TM] from Tensilica Inc.[19]).

The rest of the paper is organized as follows. Section 2 gives a brief background of SAT-based ATPG methods. It also describes our improvements to increase the algorithm's efficiency. Section 3 formulates the problem of pattern generation for crosstalk as a logic-constrained optimization problem. Section 4 explains the details of our structural SAT-based ATPG algorithm. Section 5 shows experimental results. Section 6 concludes this paper.

2. Implication Graph: Background and Improvement

2.1 Conjunctive Normal Form

SAT-based methods transform an ATPG problem into a Boolean Satisfiability problem [13, 14, 17, 18]. Figure 2 shows a circuit under test (CUT) and the logic constraints of its signal variables in conjunctive normal form (CNF). Each basic logic gate has its corresponding CNF formula. For the AND gate in Figure 2, with inputs *a*, *b* and output *c*, the logic relation is $c = a \cdot b$, and the equivalent CNF formula is:

$$(a + \overline{c})(b + \overline{c})(\overline{a} + \overline{b} + c)$$

This formula will evaluate to "1" if and only if the values of *a*, *b* and *c* are consistent with the truth table of AND gate. For each gate in the circuit, the CNF formulae must be independently satisfied, and the conjunction of the CNF formulae of all the gates must also be satisfied. The CNF formula provides a search space for test vectors.

During ATPG, in order to determine whether the error has been activated and propagated to POs, part of the circuit needs to be duplicated. For example, if for the circuit shown in Figure 2, the fault under concern is net *c* stuck-at 1 (s-a-1), then a variable *c'* is added to represent the faulty circuit. The OR gate is duplicated, and another PO *e'* is added to represent the PO in the faulty circuit. The modified circuit and its augmented CNF formula are shown in Figure 3. For a tutorial on this topic, please refer to [13, 20].

$$(a + \overline{c})(b + \overline{c})(\overline{a} + \overline{b} + c)(\overline{d} + e)(\overline{c} + e)(d + c + \overline{e})$$

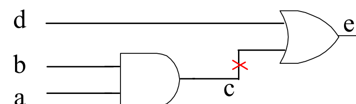

Figure 2: Circuit and logic constraints in CNF format

Compared to structural-based ATPG algorithms, the lack of structural information in the early SAT-based

ATPG algorithms and the inability to use many heuristics used by structural-based algorithms resulted in a relatively slow performance. To overcome this drawback, various methods have been developed. TEGUS [17] uses a variable selection method to mimic the heuristic used by PODEM [21], assigning values to only PIs and deriving all other variables' values by logic implication. The result of TEGUS has shown that the SAT-based algorithm together with structural information can be as fast as the best structural-based algorithm. In [22], an extra layer is added, which provides structural information, and enables utilization of general SAT solver for ATPG problems. In [18, 23], structural information is built together with implication graph, and the experimental results show significant improvement over previous methods.

$$(a+\overline{c})(b+\overline{c})(\overline{a}+\overline{b}+c)(\overline{d}+e)(\overline{c}+e)(d+c+\overline{e})$$
$$(\overline{d}+e')(\overline{c}'+e)(d+c'+\overline{e}')$$
$$(\overline{e}+e'+f)(e+\overline{e}'+f)(\overline{e}+\overline{e}'+\overline{f})(e+e'+\overline{f})$$

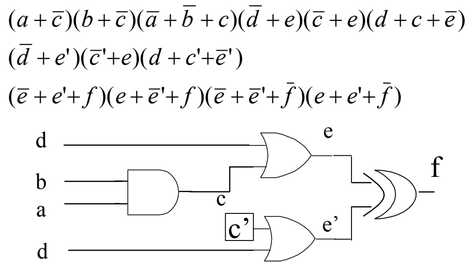

Figure 3: XOR added to combine faulty and good circuit

2.2 Implication Graph

The implication graph is a directed graph. The nodes represent logic variables and the edges represent logic implications. Similar to the method used in [13], binary CNF clauses are generated and directly transformed into implications. For example, the binary clause $(a+\overline{c})$ of an AND gate, is logically equivalent to $(c \to a)\cdot(\overline{a} \to \overline{c})$. Ternary clauses can also be transformed. For example, the ternary clause $(\overline{a}+\overline{b}+c)$ of an AND gate, is transformed into an equivalent logic relation: $((a \cdot b) \to c)\cdot$ $((a \cdot \overline{c}) \to \overline{b})$ $\cdot((b \cdot \overline{c}) \to a)$ [23].

Based on these transformations, implication graph can be constructed. The implication graph contains two kinds of nodes: variable nodes and logic "Λ" (AND) nodes. For example, Figure 4, shows the Implication Graph for a two-input AND gate, with inputs *a* and *b* and output *c*. In IG, according to the physical relation between logic variables, the edges that denote implications are labeled as f (forward), b (backward) and o (other). Because the structural information is embedded in the implication graph, heuristics from structural algorithms can be adopted. Another advantage of implication graph is that algorithms only need to handle two kinds of nodes: the signal nodes and the logic "Λ" nodes. Compared to structural-based methods, which need to check look-up table for different logic gates and logic operations, the IG enables faster logic implication and justification.

The method in [18] shows significant improvement over previous ATPG algorithms. However, this method still needs to duplicate the whole implication graph. This imposes a large memory requirement. To eliminate the overhead of the duplication and large memory requirement, we introduce a simple D frontier label in the implication graph.

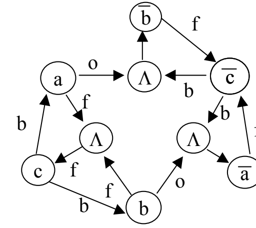

Figure 4: Implication graph for 2-input AND gate

2.3 D frontier in Implication Graph

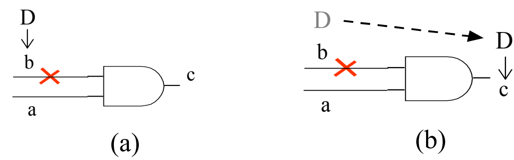

Figure 5: D frontier (a) circuit under test
(b) D frontier propagate though AND gate

Previous SAT-based ATPG algorithms either duplicate part of the circuit (from the faulty site to POs)[13], or add active clauses to represent possible propagation paths. After test generation for each fault, all the extra variables have to be removed. The duplication and clean-up operation are repeated for each fault. This overhead could be expensive when the number of faults is large. In structural ATPG algorithm, D frontier is used to denote the different behavior of the good circuit and the faulty circuit. By simply adding one extra label for each variable node to symbolize whether it is a D frontier, we can perform fast D frontier propagation on the implication graph without the duplication overhead.

When a fault is activated, the first D frontier will be introduced. For example, consider the circuit shown in Figure 5 (a). Suppose fault *b* s-a-0 is activated. Signal *b* should be 1 in the good circuit and 0 in the faulty circuit. Therefore, *b* will be assigned as the first D frontier. In order to further propagate D frontier though the AND gate to *c*, signal *a* has to be logic 1. Backtrace is performed from signal *a* to PIs, and forward implication is used to try to justify the objective. Suppose *a*=1 is successfully justified. To determine whether the D frontier can propagate though the AND gate, simulation is performed on the implication graph. For the good circuit, the b should be 1 and in the corresponding IG shown in Figure 6 (b) node *b* is marked, and node *c* is implied. For the faulty circuit, *b* should be 0. Hence, node $\overline{b}$ is marked and node $\overline{c}$ is implied in Figure 6 (c). The result nodes are node *c* and its complement $\overline{c}$ for the good circuit and the faulty circuit, respectively. Hence the D frontier can propagate

though the AND gate and node *c* will be labeled as the new D frontier as shown in Figure 6 (d).

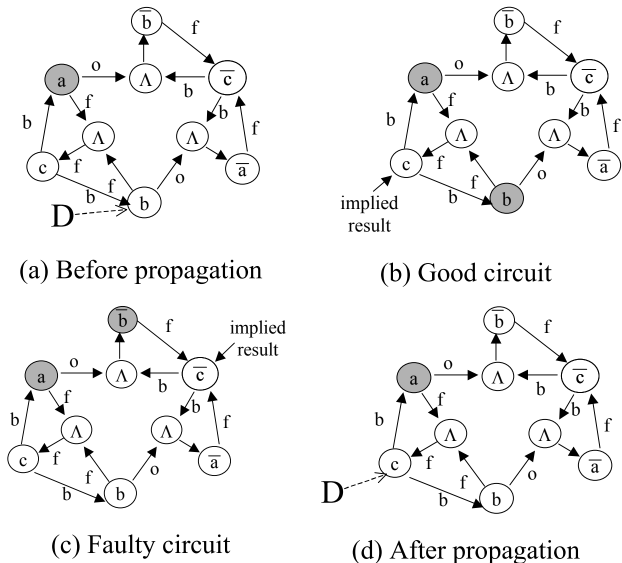

Figure 6: D frontier propagate though AND gate in IG

By introducing the D frontier in the implication graph, we avoid the overhead of the duplication operation. The improved implication graph inherits advantages from both structural-based algorithms and SAT-based algorithms. Thus it enables faster logic conflict checking and logic implication operations.

3. Problem Formulation

The goal of ATPG for crosstalk is to activate a crosstalk error, propagate the error effect to POs and trigger as many aggressors as possible to attack the victim. However, the logic relations among the aggressors, the victim and potential propagation paths pose complicated constraints on the ATPG problem. Figure 7 shows an example with multiple aggressors. The target error is a positive glitch on wire *v*, and the potential aggressors are *a*, *b*, *c*, and *d*. Dotted lines show the preferred transitions on the aggressors and the bold waveform shows the noise effect we intend to generate on the victim. Because of logic relation of the NAND gate, signals *a*, *b* and *c* cannot have rising transitions simultaneously. Also, in order for the noise effect to propagate though the NOR gate and fit into the sampling windows at POs, d has to be logic 0 (we use a single fault assumption).

Two vectors are needed to generate the desired transitions. According to their sequence relation to the transition, they can be denoted as *before-switching vector* and *after-switching vector*. We denote the logic value of aggressor *i* for *before-switching* as A_i^{bs}, and the logic value of aggressor *i* for *after-switching* as A_i^{as}. Therefore, for a rising transition on aggressor *i*, we have, A_i^{bs} = 0 and A_i^{as}=1.

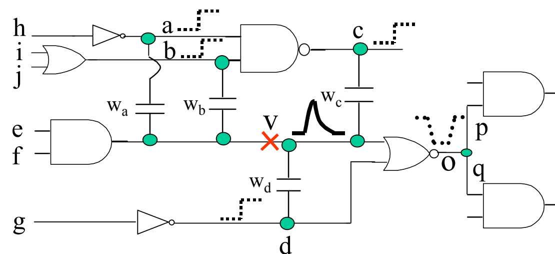

Figure 7: Aggressors cannot be triggered simultaneously

Because of logic constraints, it is not always possible to trigger all the aggressors. If that happens, the ATPG algorithm should choose to activate a subset of most "important" aggressors to trigger a large noise effect. Using extracted layout information, and the crosstalk noise estimation tool from [24], we assign a normalized weighting factor to each aggressor. This factor is used as a measure of how much the given aggressor can potentially contribute for the noise generation. For example, the contribution of aggressor *i* with weighting factor w_i to a positive glitch noise is $w_i(A_i^{as} - A_i^{bs})$. Therefore, the contribution is equal to w_i for a rising transition on the aggressor *i*, it is equal to $-w_i$ for falling transition on *i*, and it is equal 0 for the case of a quiet aggressor *i*.

Therefore, the ATPG problem can be formulated as an optimization problem. The objective is to activate the largest possible error effect subject to constraints including fault activation, fault propagation, and logic constraints of the circuit. Consider the positive glitch fault on victim *v* as an example. The problem can be formulated as:

$$\text{Maximize} \sum_{i \in aggressors} w_i(A_i^{as} - A_i^{bs}) \qquad (1)$$

Subject to:

v^{as}=0, v^{bs}=0; (fault activation)

Sensitized propagation path from *v* to PO

$$\left.\begin{aligned} &A_i^{bs} + X_j^{bs} + X_k^{bs} + \overline{X}_l^{bs} = 1 \\ &\overline{A}_{i+2}^{bs} + \overline{X}_n^{bs} + X_k^{bs} = 1 \\ &\ldots \end{aligned}\right\} \text{Logic constraints (\textit{before-switching})}$$

$$\left.\begin{aligned} &A_i^{as} + X_j^{as} + X_k^{as} + \overline{X}_l^{as} = 1 \\ &\overline{A}_{i+2}^{as} + \overline{X}_n^{as} + X_k^{as} = 1 \\ &\ldots \end{aligned}\right\} \text{Logic constraints (\textit{after-switching})}$$

$A_i, X, v \in \{0,1\}$; *A, X, v* are signals of the CUT

The patterns generated by ATPG should justify the appropriate logic values for the victim wire and propagate the error effect to PO(s). For a positive glitch on the victim *v* in the circuit shown in Figure 7, v^{bs}=0 and v^{as}=0 are implied. When the *after-switching vector* is applied, the

signal of wire v in the good circuit will be 0 and in the faulty circuit with a positive glitch, the signal of wire v should be 1 for some period and then fall back to 0. Similar to handling stuck-at faults, D frontier can be used to denote the different behavior between the good circuit and the faulty circuit. The fault propagation can be achieved by propagating the D frontier for an equivalent stuck-at-1 fault, from the faulty site to PO by the *after-switching* vector.

The ATPG problem is a 0-1 integer-programming problem and it is NP hard. In order to minimize the complexity, we use zero-delay model and assume that the circuit is hazard free. To further speedup the search process, the original optimization problem can be divided into two parts: *before-switching* and *after-switching*. For example for a positive glitch, we want to trigger rising transitions on aggressors. The *before-switching* vector needs to justify 0 for the victim wire and to justify as many 0's on the aggressors as possible. The *after-switching* vector need to justify a 0 for the victim, sensitize a path from the victim to PO(s), and to justify as many 1's on the aggressors as possible. Hence the original problem in (1) is divided into two parts with different constraints:

(a) *Before-switching*:

$$\text{Minimize} \sum_{i \in aggressors} w_i A_i^{bs} = \text{Maximize} \sum_{i \in aggressors} w_i \overline{A}_i^{bs} \quad (2)$$

s.t.
Victim wire logic value (v^b=0 for positive glitch);
Logic constraints;

and

(b) *After-switching*:

$$\text{Maximize} \sum_{i \in aggressors} w_i A_i^{as} \quad (3)$$

s.t.
Victim wire logic value (v^a=0 for positive glitch);
Propagation path from v to PO;
Logic constraints;

Note that the original objective function is transformed into two parts:

$$\text{Maximize} \sum_{i \in aggressors} w_i (A_i^{as} - A_i^{bs}) \text{ equivalent to}$$

$$(\text{Maximize} \sum_{i \in aggressors} w_i A_i^{as}) \text{ and } (\text{Maximize} \sum_{i \in aggressors} w_i \overline{A}_i^{bs})$$

and each part is an optimization problem, with different constraints. In these equivalent optimization problems, if a rising transition occurs on aggressor A_i, the total objective function value will be $2w_i$; if a falling transition occurs, the total obj. function value will be 0; and if a stable 0 or stable 1, the obj. function value will be w_i. Hence it encourages rising transitions on as many "important" aggressors as possible. If we assume the two input vectors at PIs are not correlated, then we can solve these two parts separately. Next, we will introduce an algorithm to efficiently search for solutions for the above optimization problems using a single framework.

4. Test Generation Algorithm

If we could afford to evaluate all vectors satisfying the constraints in Equation (1), then we could compare them and select the one with the maximum objective function value. Obviously, this is not feasible for large circuits. Our ATPG algorithm is divided into 2 phases. Given a crosstalk target fault consisting of a victim line and a set of potential aggressors, in the first phase we check if the set of aggressors is feasible. If not, we identify a subset of feasible aggressors. The outcome of the first phase is a set of objectives that need to be justified in the second phase. We use a modified PODEM algorithm to search for vectors justifying the multiple objectives. If the proposed set of aggressors fails in the second phase, the algorithm can iterate and select another set of relaxed objectives. In the following, we describe the details of the two phases.

4.1 Phase I: Search for Feasible Objectives

Mandatory Assignments.
For both *before-switching* and *after-switching* vectors, there are several constraints that have to be satisfied by any pattern. These constraints result in mandatory assignments. First, the logic value of the victim should be justified. Second, if all the potential propagation paths share a common segment, this segment of the propagation path has to be sensitized, i.e., the corresponding side inputs of the segment need to be assigned and justified. For example, for the positive glitch on victim v in Figure 7, v^{as}=0 and v^{bs}=0 are mandatory assignments. Since the NOR gate is on the only possible propagation path, the signal value of the side input d needs to be considered. Even though a late rising transition on d may allow the noise to propagate though the NOR gate, because of our single fault assumption we do not consider this case as possible to allow the error to be sampled at POs. Hence, d^{as} =0 is a mandatory assignment and consequently, g^{as} =1 is implied. Initially, a D frontier is added for v. After assigning d^{as} =0, the D frontier propagates though the NOR gate, and variables p and q will be added to D frontier list. Next, since multiple potential propagation paths exist (p and q), no more mandatory assignments will be performed for the D frontier propagation.

The assignment of mandatory values and their implications are performed on the IG. Aggressors that have been assigned and/or implied mandatory values in this step will be removed from the set of aggressors under consideration for the next step.

Objective Relaxation.
After the assignment of mandatory values, we need to further check the feasibility of the current set of

aggressors. For easier understanding of our algorithm, we will use the search for *after-switching* vector for a positive glitch as example. Because aggressor d was assigned a mandatory value 0 in the first step, we only need to consider signals a, b and c. Without loss of generality, let us assume that the weights of a, b and c are 6, 5 and 4, respectively. The implication graph is used to check for logic conflicts.

Because each aggressor can be either logic 0 or 1, a BDD tree can be constructed to fully explore all possible aggressor assignment combinations. Figure 8 (a) shows such a tree. The leaves of the BDD tree represent the total weights obtained by assigning aggressors according to the branches along the path from the root to leaves. Because of logic constraints, some branches of the tree are trimmed (e.g., a=0, b=0 and c=0). The trimmed branches and leaves are shown in dotted lines. The potentially feasible combinations can be sorted according to the weight of each leaf. Then, the combination with the highest value is chosen and the set of aggressor assignments act as objectives, which need to be justified in the next phase of the ATPG algorithm. If it turns out that no vector can justify these objectives, the process will be repeated with selecting the leaf with the second highest weight. This way, the search process keeps relaxing the objective function value until a solution is found or the weights of the remaining leaves are less than a given threshold value.

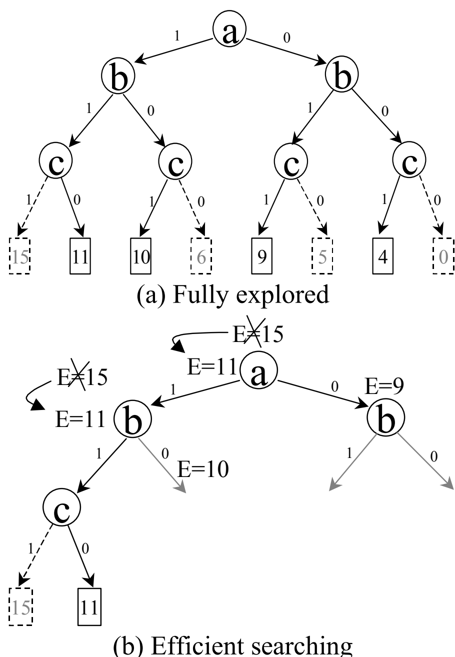

Figure 8: BDD Tree in searching for feasible aggressors

However, since the size of the BDD tree increases exponentially with the number of aggressors, it may not be affordable to fully explore the BDD tree. Building the whole tree and checking for logic conflicts for all possible combinations is impossible for some circuits. In addition, each implication and unimplication operation could affect hundreds of variables on IG. Hence, each exploring step on the BDD tree may need expensive operations on IG. Therefore, in order to make the search more efficient, only the necessary implication and unimplication operations should be performed.

We develop a simple search algorithm to achieve this goal. Before exploring a node, we assume that the best combination of aggressors in the subtree of both branches is feasible. An expected value is calculated for the unexplored branches. The algorithm compares two branches of current node to decide which branch should be chosen. When conflict occurs or node in the downstream is implied, the expected values will be updated. For example, in Figure 8 (b), we begin with node a. Since for branch a=0, the estimated value is 9, and for branch a=1 the estimated value is 15, we choose a=1. Implication on the IG will imply $\bar{h}$ and there is no conflict. Then algorithm will move on to node b, this time branch b=1 will be chosen. Again implication operation will be performed to check for conflicts.

During the implication operation, the objective list and D frontier list will be built and updated. Implication graph for the NAND and part of the OR gate of our example is shown in Figure 9. Suppose during the implication operation for node b=1, the backward implication begins from variable b and it stops at the logic nodes before reaching variables i and j. Variable b will be added to the objective list (to be justified later). The forward implication begins from variable b. Since variable a has already been marked, $\bar{c}$ will be implied. The assigned nodes are marked in dark color, and the implication operations are shown in bold arrows in Figure 9.

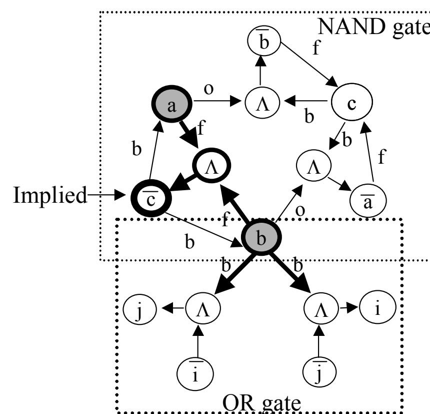

Figure 9: Example of backward and forward implication

Because $\bar{c}$ is implied, branch c=1 will be trimmed on the BDD tree. The estimated values for all branches from the current node to the root will be updated. If the current node does not have the best estimated value anymore, unimplication will be performed, and the search process will move upward until it reaches a node containing a branch with the highest estimated value. To reduce the risk of jumping among different subtrees, the nodes should be

sorted. In this example, the updated estimated value is 11 for nodes *a* and *b* (Figure 8(b)) meaning that the current branch still has the best estimated solution. The search process will continue till it reaches a leaf (*a*=1, *b*=1 and *c*=0). Then modified PODEM will be used to search for a vector that can justify the objectives and to propagate the D frontier(s).

4.2 Phase II: Justifying the Objectives

To efficiently justify objectives and propagate D frontier(s) to PO(s), we use the same strategy as PODEM [21]: making decisions only on PIs to minimize the search space. First, the objectives are extracted from the objective list. BACKTRACE will search and return an unassigned PI variable. Then forward implication is performed. Objectives will also be generated by D frontiers to guide the modified PODEM algorithm to sensitize propagation path(s). After each forward implication, D simulation is performed to propagate D frontiers and check whether D frontier reached PO(s). Figure 10 shows the pseudo code for the modified PODEM.

```
Modifed_PODEM()
Begin
  If(D at PO and objective list is empty)
          return SUCCESS
  If(D frontier list is empty and not reach PO)
          return FAILED
  (k,v_k)=OBJECTIVE()
  (j,v_j)=BACKTRACE(k,v_k) /* j is a PI */
  if(IMPLY(j,v_j))
     {
      D_Simulation()
      if(Modified_PODEM()==SUCCESS)
          return SUCCESS
     }
  UNIMPLY(j,v_j)
  /*reverse PI assignment*/
  if(IMPLY(j,v̄_j))
     {
      D_Simulation()
      If (Modified_PODEM()==SUCCESS)
          return SUCCESS
     }
  UNIMPLY(j, v̄_j)
  Return FAILED
END
```

Figure 10: Modified PODEM

Procedures used in the modified PODEM include BACKTRACE for backtracing from a given variable to primary inputs; and IMPLY for forward implications from PIs and checking for logic conflicts. These procedures are similar to those in [25]. The D frontier propagation is performed by D simulation as explained in section 2.3.

The search for potential feasible objectives for the *before-switching* vector is similar to the *after-switching* vector, but the algorithm does not need to consider D frontier propagation and the preferred logic value is 0 for aggressors (since we are considering a positive glitch on the victim). Hence, the weights are assigned to $\bar{a}$, $\bar{b}$ and $\bar{c}$. The same framework for solving the ATPG problem as an optimization problem is used again.

5. Experimental Results

To evaluate the proposed ATPG method, we have implemented the ATPG algorithm in C++ and applied it to benchmark circuits. In these experiments, we used the positive glitch as an example. First, SPICE simulation is used to evaluate the quality of the ATPG vectors for two circuits, comparing with an exhaustive set of vector pairs. However, for large circuits exhaustive simulation becomes very time-consuming. Hence, we will report results on two large circuits by comparing randomly generated vectors against the ATPG vectors. Finally, the efficiency of the ATPG methodology is assessed. We present experimental results on ISCAS85 circuits and a large logic cone from the Xtensa processor [19]. All these experiments were performed on a Sun Ultra 5 workstation with 512MB memory.

To assess the quality of the ATPG vectors, we performed exhaustive transistor-level simulations for two small circuits. The first circuit is shown in Figure 11. It has 3 inputs and 2 outputs, with six logic levels. It has 11 potential crosstalk candidates. For each crosstalk candidate, two SPICE netlists were used. One in which coupling defects were injected (faulty circuit) and another without the coupling defects (good circuit). We simulated all possible input-vector sequences. Crosstalk-induced noises were calculated by comparing the simulation results of the good circuit with the simulation results of the faulty circuit. In our simulation results, the ATPG vectors triggered the largest noises among all vector pairs.

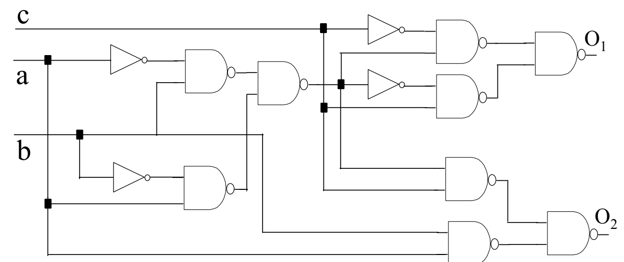

Figure 11: Circuit1, with shallow logic levels

The second test circuit that we examined is shown in Figure 12. It has 5 inputs, 3 outputs and deeper logic levels. There were 25 potential crosstalk fault candidates identified. Exhaustive simulations were performed for all the potential crosstalk candidates as described above.

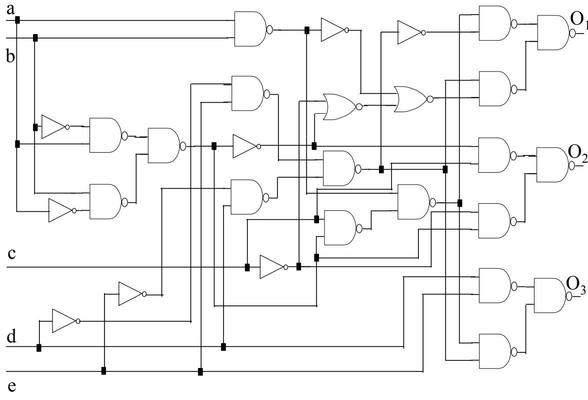

Figure 12: Circuit2 with deeper logic levels

Out of the 25 potential crosstalk fault candidates, only 9 of them showed noticeable positive glitch noise in these simulations. This is because some aggressors cannot be triggered due to logic relations between aggressors and the victim. Table 1 shows the noise magnitude (in volt) for each fault that has noticeable noise triggered in these simulations. Among all the vectors, the ATPG vectors are identified and compared with all the other vectors (exhaustive vectors excluding the set of ATPG vectors). The ATPG vectors successfully generated the largest noise effects for all the faults except fault 11. Even for this fault, the ATPG vectors still triggered a large noise, whose magnitude was 96% of the worst-case noise magnitude.

Fault	Noise Mag. (ATPG)	Noise Mag. (other)
3	1.3364	1.3122
8	1.9979	1.9016
9	2.3330	2.2152
10	3.2878	3.0536
11	2.3240	2.4143
15	0.5588	0.5372
16	3.6992	2.8920
18	1.1664	1.0962
19	0.6559	0.5735

Table 1: Experimental result of circuit2

For a circuit with relative large number of PIs, exhaustive simulation becomes too expensive. Hence, random vectors are widely used. However, random vectors frequently fail to trigger and propagate crosstalk errors. Targeting crosstalk fault candidates, HyAC enables efficient high-quality test vector generation. To illustrate this, we performed experiments on an ALU of the PARWAN CPU [26], and a 32-bit full-adder in a RISC core. In these experiments, we selected crosstalk fault candidates using layout-extracted physical information and the noise estimation tool [24]. Due to the large number of test vectors, exhaustive simulations cannot finish in a reasonable time. Even for the relatively small ALU, exhaustive simulation requires $2^{18} \times (2^{19} - 1)$ vector pairs. Hence, we generated random test vectors. For each fault candidate, we generated 128 random vectors by perturbing test vectors for stuck-at faults on the victim of the crosstalk fault. Using SPICE simulation, the quality of the ATPG generated vectors was compared with the quality of the randomly generated vectors. Table 2 shows the experimental results of test generation and SPICE simulations for positive glitches. The third column shows the average magnitude of crosstalk noise triggered by the best random vector pairs for all fault candidates. The fourth column shows the average magnitude of crosstalk noise triggered by the ATPG generated vectors. In all the simulated cases, the ATPG-generated vector pairs successfully triggered the largest noise effect.

CUT	Total # of Faults	Avg Noise (random)	Avg. noise (ATPG)	# faults with infeasible aggressor set
ALU	114	0.27	2.763	48
32-bit Adder	277	0.418	1.19	126

Table 2: Results on large circuits with layout information

In Table 2, the fifth column shows the number of crosstalk candidates for which the original set of aggressors extracted from layout cannot be triggered simultaneously due to logic constraints. For many potential crosstalk candidates, the worst-case situation of assuming all the aggressors attacking the victim will not happen. Therefore, a static noise estimation tool that assume all the aggressors can be triggered will be too pessimistic. The proposed ATPG method can help to improve the noise estimation accuracy by eliminating some false crosstalk error reports.

CUT	T_1(sec)	T_2(sec)	T_3(sec)	# of abort
C432	<1	0.03	0.01	6
C499	<1	0.03	0.02	3
C1908	1	0.43	0.02	12
C2670	1	0.06	0.02	2
C3540	1	0.49	0.16	9
C5315	1	1.99	0.16	19
C6288	1	1.98	0.32	12
C7552	1	1.31	0.21	10

T1: time spent on building implication graph
T2: average time spent on ATPG per fault (all faults)
T3: average time spent on ATPG per fault (exclude faults with aborted aggressor set)

Table 3: Experimental results on ISCAS85 circuits

To evaluate the efficiency of the proposed ATPG method, experiments are also conducted on ISCAS85 circuits and an industrial circuit. Table 3 shows the results on ISCAS85 benchmark circuits. Due to the lack of layout information for ISCAS85 circuits, we randomly selected

victims, aggressors and randomly injected coupling capacitances between the victim and its corresponding aggressors. We used the coupling capacitances as weights for aggressors. For each benchmark circuit, we randomly generated 200 test cases. We set 4000 as the backtrack limit for the modified PODEM algorithm.

In Table 3, columns 2, 3 and 4 show the time used for building the implication graph, the average time spent on ATPG per fault and the average time spent on ATPG per fault excluding faults with aborted aggressor sets, respectively. Comparing the third column with the fourth column of Table 3, it is clear that a lot of time was consumed by trying to generate vectors for faults with aborted aggressor sets. Since the modified PODEM algorithm exhaustively explores all possible ways to satisfy a set of objectives, it can become very costly to prove that a set of aggressors are impossible to be justified (equivalent to redundant fault in the stuck-at fault model). Hence, efficient detection of such redundancy can tremendously improve the speed of the ATPG for these cases. Column 5 shows the number of victims for which one or more sets of aggressors were aborted due to the backtrack limit in PODEM. Note that this abort means that a set of aggressors cannot be justified within the given backtrack limit and the set of aggressors will be given up. However, for the same crosstalk victim, a relaxed set of aggressors will be proposed and justified.

We also applied the proposed ATPG method on the Xtensa processor from Tensilica Inc. [19]. The Xtensa processor is a commercial configurable and extensible RISC processor with 5 pipeline stages and 81 core instructions. A typical configuration of Xtensa processor was synthesized using Leonardo Spectrum™ [27]. Experiments were conducted on a large module, called EX1, which is the logic cone leading to the result of the execution stage in the pipeline. The logic cone has one 32-bit output port and 81 input ports corresponding to 335 bits. Ex1 contains the basic building blocks for completing arithmetic/relational/logical operations. It also contains logics for pipeline forwarding. A layout of EX1 was generated using IC Station® [28] and parasitic parameters were extracted by xCalibre® [29]. 998 potential crosstalk-fault candidates were identified using the noise estimation tool [24].

	Total # of faults	Total Run time (sec)	# of aborted faults
Ex1	998	1006	38

Table 4: ATPG result on Ex1 in Xtensa Processor

Assuming the PIs and POs of Ex1 are fully controllable and observable by using scan chains, we performed ATPG for the potential crosstalk fault candidates. Table 4 shows the total ATPG run time on Ex1 and the number of aborted faults. The above experiments demonstrate that the proposed ATPG methodology can successfully generate high-quality vectors.

6. Conclusions and Future Work

Based on Hybrid structural SAT, an efficient deterministic ATPG method, HyAC, has been developed for crosstalk induced logic and transition faults. HyAC efficiently explores logic conflicts among the victim, the set of aggressors and the propagation path(s). It automatically selects a set of potentially feasible aggressors, searches for vectors to trigger those aggressors and sensitizes a propagation path(s). The ATPG method generates high-quality test vectors, enabling efficient simulation-based verification and manufacturing test for crosstalk errors.

Our current ATPG flow assumes scan-based DFT is employed, hence all vectors can be applied at the PIs of the CUT and there is no correlation between the consecutive test vectors. However, many vectors cannot be generated in functional mode because the functional mode imposes extra constraints on the PIs and POs and correlations may exist between consecutive vectors. Further study will be conducted to address the challenge of ATPG considering functional mode imposed constraints.

Crosstalk noise induced errors are very complex and ATPG is extremely hard. The main complexity comes from the requirement for specific logic and timing conditions that need to be satisfied to activate the largest crosstalk noise impact. However, usually, considering timing considerably hurts the scalability of the ATPG. Our current framework does not include timing information in the ATPG. Instead, crosstalk fault candidates are selected by using other tools, considering timing, noise estimation and potential process variations. Our experiments show HyAC can generate high-quality test vectors. This is especially true for circuits with small number of logic levels such as high-speed and highly pipelined circuits. Experiments were also conducted on ISCAS85 circuits and an industrial circuit to demonstrate the scalability of the proposed ATPG method. By leveraging the advantages of both Boolean-satisfiability based methods and structural based methods, HyAC is fast and effective in generating test vectors for multiple-aggressor induced crosstalk errors.

References

[1] C. Werner, R. Göttsche, A. Wörner and U. Ramacher, "Crosstalk Noise in Future Digital CMOS Circuits," *Design, Automation and Test in Europe Conference*, pp.331-335, 2001.

[2] H. Kawaguchi and T. Sakurai, "Delay and Noise Formulas for Capacitively Coupled Distributed RC Lines," *Proc. the Asian and South Pacific Design Automation Conference*, pp.35-43, 1998.

[3] K. Rahmat J. Neves, J. Lee, "Methods for calculating coupling noise in early design: a comparative analysis," *Proc. International Conference on Computer Design VLSI in Computers and Processors,* pp.76-81, 1998.

[4] Hsiao-Ping Tseng, Scheffer, L., Sechen, C. “Timing- and crosstalk-driven area routing,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol.20 pp 528-544, April 2001.
[5] I. H-R Jiang, Y-W Chang and J-Y Jou, “Crosstalk-Driven Interconnect Optimization by Simultaneous Gate and Wire Sizing,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and systems, Vol. 19, pp. 999-1010, Sept. 2000.
[6] K. M. Lepak, I. Luwandi, and L. He, "Shield insertion and net ordering under explicit RLC noise constraint", *Proc. of Design Automation Conference*, June 2001.
[7] K. T. Lee, C. Nordquist and J. A. Abraham, “Automatic test pattern generation for crosstalk glitches in digital circuits,” *Proc. of VLSI Test Symposium*, pp. 34-39, 1998.
[8] W. Y. Chen, S. K. Gupta and M.A. breuer, “Test generation in VLSI circuits for crosstalk noise,*” Proc. Int’l Test Conf.*, pp. 641-650, 1998.
[9] W. Y. Chen, S. K. Gupta and M. A. Breuer, “Test generation for crosstalk-induced delay in integrated circuits,*” Proc. Int’l Test Conf.*, pp. 191-200, 1999.
[10] R. Kundu and R. D. Blanton, “Timed Test Generation for Crosstalk Switch Failures in Domino CMOS Circuits,” *Proc. of VLSI Test Symposium*, pp. 379~385 2001.
[11] B. C Paul, K. Roy, “Testing Cross-Talk Induced Delay Faults in static CMOS circuit Though Dynamic Timing Analysis,” *International Test Conference*, pp. 384-390, Oct. 2002.
[12] A. Krstic, J.-J. Liou, Y.-M. Jiang and K.-T. Cheng, “Delay Testing Considering Crosstalk-Induced Effects,” Proc. of International Test Conference, pp.558-567, Oct. 2001.
[13] T. Larrabee, “Test pattern generation using Boolean Satisfiability,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, pp.4-15, Jan. 1992.
[14] J. Marques-Silva and K. A. Sakallah, “GRASP- A New Search Algorithm for Satisfiability,” in *proc. of the International Conference on Computer-Aided Design*, pp. 220-227, Nov. 1996.
[15] Schulz M., Trischler E., Sarfert T.,”SOCRATES: A highly efficient automatic test pattern generation system,” *Proc. Intl. Test Conf.*, pp.1016-1026, 1987.
[16] P. Chen and K. Keutzer, “Towards True Crosstalk Noise Analysis,” *proc. International Conference on Computer-Aided Design* (ICCAD), pp. 132-137 Nov. 1999.
[17] P. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, “Combinational Test Generation Using Satisfiability,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1167-1176, Sept. 1996.
[18] P. Tafertshofer, A. Ganz, “SAT based ATPG Using Fast Justification and Propagation in the Implication Graph,” *IEEE International Conference on Computer-Aided Design* (ICCAD), pp. 139-146, 1999.
[19] Xtensa[TM] Microprocessor Overview Handbook, http://www.tensilica.com/xtensa_overview_handbook.pdf, Tensilica Inc, Aug. 2001.
[20] J. P. Marques-Silva, K. A. Sakallah, “Boolean Satisfiability in Electronic Design Automation,” in *Proceedings of the Design Automation Conference*, June 2000.
[21] P. Goel, “An implicit enumeration algorithm to generate tests for combinational logic circuits,” *IEEE Trans. Comput.*, Vol. C-30, pp. 215-222, Mar. 1981.
[22] L. G. e Silva, L. M. Silveira, and J. Marques-Silva, “Algorithms for solving Boolean satisfiability in combinational circuits,” in *Design, Automation and Test in Europe* (DATE), pp. 526-530, Mar. 1999.
[23] P. Tafertshofer, A. Ganz and M. Henftling, “A SAT-Based Implication Engine for Efficient ATPG, Equivalence Checking, and Optimization of Netlists,” *IEEE International Conference on Computer-Aided Design* (ICCAD), pp. 648-655, Nov. 1997.
[24] X. Bai, R. Chandra, S. Dey and P.V. *Srinivas, “Noise-Aware Driver Modeling,” Proc*. of *International Symposium on Quality Electronic Design*, pp.177-182, 2003.
[25] M. Abramovici, M. A. Breuer, A. D. Friedman, *Digital Systems Testing and Testable Design*, IEEE Press, pp. 182-208, 1990.
[26] Z. Navabi, VHDL: *Analysis and modeling of digital systems*, New York, McGraw-Hill 1993.
[27] LeonardoSpectrum[TM], Mentor Graphics Corp.
[28] IC *Station*®, Mentor Graphics Corp.
[29] xCalibre®, Mentor Graphics Corp.