SAT Based ATPG Using Fast Justification and Propagation in the Implication Graph

Paul Tafertshofer Andreas Ganz

Institute for Electronic Design Automation
Technical University of Munich
80290 Munich, Germany
{Paul.Tafertshofer,Andreas.Ganz}@ei.tum.de

Abstract

In this paper we present new methods for fast justification and propagation in the implication graph (IG) which is the core data structure of our SAT based implication engine. As the IG model represents all information on the implemented logic function as well as the topology of a circuit, the proposed techniques inherit all advantages of both general SAT based and structure based approaches to justification, propagation, and implication. These three fundamental Boolean problems are the main tasks to be performed during Automatic Test Pattern Generation (ATPG) such that the proposed algorithms are incorporated into our ATPG tool TIP *which is built on top of the implication engine.*

Working exclusively in the IG, the complex functional operations of justification, propagation, and implication reduce to significantly simpler graph algorithms. They are easily extended to exploit bit-parallel techniques. As the IG is automatically generated for arbitrary logics the algorithms remain applicable independent of the required logic. This allows processing of various fault models using the same engine. That is, the presented IG based methods offer a complete and versatile framework for rapid development of new ATPG tools that target emerging fault models such as cross-talk, delay or bridging faults. TIP *currently handles stuck-at as well as various delay fault models. Furthermore, the proposed methods are used within tools for Boolean equivalence checking, optimization of netlists, timing analysis or retiming (reset state computation).*

In order to demonstrate the performance of IG based ATPG, i.e. justification and propagation in the IG, we provide experimental results for stuck-at and path delay fault models. They show that TIP *outperforms the state-of-the-art in SAT based and structure based ATPG.*

1 Introduction

Automatic Test Pattern Generation (ATPG) primarily has to solve three fundamental Boolean problems: *justification*, *propagation*, and *implication*. In the past, various data structures have been used to tackle these problems with none of them being specifically optimized for all these tasks.

The first set of approaches relies on a structural description (*netlist*) of the circuit to be analyzed [1, 2, 3, 4]. In this model the functionality of the circuit is jointly represented by the netlist and a module library. While the netlist describes the topology of the circuit and the type of each module, the library provides information on the logic function implemented by a given module type. This separation in description complicates algorithms, especially when working with multi-valued logics (e.g. for path delay ATPG).

Contrary to above methods, a second set of approaches uses a *Boolean satisfiability (SAT)* based model that describes the logic functionality of a circuit within a single Boolean formula [5, 6, 7, 8, 9, 10] which is mostly given in terms of a *Conjunctive Normal Form (CNF)*. The SAT model allows a compact problem formulation which is easily adapted to various logics and can be solved by a solver for general SAT problems. This abstraction, however, often impedes development of efficient algorithms as structural information on the circuit is lost. For example, the efficient PODEM based justification cannot be transferred adequately to this model as the notion of a primary input does not exist in an arbitrary SAT problem. Larrabee suggests to solve the task of propagation by extracting a SAT formula from the split circuit model [11] that corresponds to a formula generated by the Boolean difference method [5]. In order to reduce the complexity of solving the resulting formula, structural information on possible propagation paths needs to be added in form of additional clauses *(active clauses)*. Very recently, the SAT based algorithm of [8] has been specialized for solving problems originating from combinational circuits [9]. This is achieved by adapting some of the ideas proposed in [12] such that an additional layer is added to the SAT solver which models the topology of the circuit. As this structural information is only used for justification, the beneficial effects remain limited. In general, this group of approaches is less efficient than structure based methods.

Binary Decision Diagrams (BDD) have also been proposed to tackle justification and propagation [13]. Besides their exponential memory complexity, here, BDDs suffer from their exhaustive nature. That is, when trying to justify a signal assignment, BDD based techniques always compute the complete set of justifications even if a single justification is sufficient. BDD based propagation relies on the split circuit model and the Boolean difference method. In order to constrain the excessive memory requirements, Stanion et al. suggest to consider the possible propagation paths when building the BDDs [13]. In the worst case, however, all propagation paths have to be modeled in a single BDD which is very likely to cause a memory blowup.

So, despite the high importance of justification, propagation, and implication, the data structures used so far have proven to be suboptimal and inflexible in several respects. That is why we propose fast and optimized algorithms for justification, propagation, and implication that are built around a versatile and efficient SAT based *implication engine* [12] as shown in Fig. 1. It inherits the advantages of structure based as well as SAT based techniques as it includes all topological and functional information into a single graph model of the *CNF*, called *implication graph (IG)* [12]. Thus, IG based algorithms combine both the flexibility and elegance of SAT based tech-

0-7803-5832-X /99/$10.00 ©1999 IEEE.

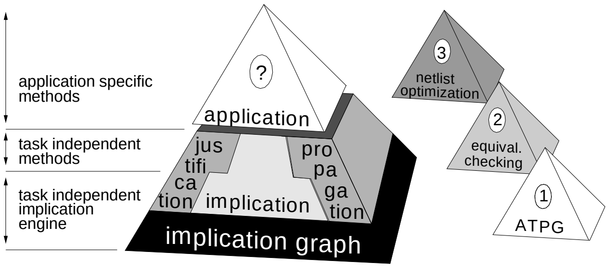


Figure 1: Basic structure of the implication engine

niques and the efficiency of structure based methods. The multitude of heuristics developed for structure based techniques can directly be transfered to the IG. Its memory complexity is only linear in the number of modules in the circuit. As the complex functional operations of justification, propagation, and implication reduce to simple graph algorithms they are easily extended to make use of bit-parallel techniques resulting in a high efficiency. This paper introduces new methods for fast IG based justification and propagation that are included into the implication engine of [12]. Using these algorithms our tool TIP outperforms the state-of-the-art in structure based and SAT based ATPG. Since the IG is automatically generated for an arbitrary logic and the presented algorithms for justification and implication remain applicable independent of the chosen logic, ATPG for various fault models can easily be built on top of the same engine. Tools for path delay, gate delay and stuck-at fault models have already been developed. Additionally, the implication engine has successfully been applied in tools for Boolean equivalence checking [12], netlist optimization [14], and timing analysis [15].

This paper is organized as follows. In Sections 2 and 3, we briefly review the basics introduced in [12]. Sections 4 and 5 discuss justification and propagation in the IG. In order to demonstrate the high efficiency of our IG based ATPG approach, experimental results for stuck-at and path delay ATPG are presented in Section 6. Section 7 concludes the paper.

2 Implication graph (IG)

An IG is a directed graph $G = (V,E)$. The set of nodes V divides into *signal nodes* V_S and $\wedge$-*nodes* $V_\wedge$. In this paper, signal nodes are indicated by c_x (c_x^*) where x corresponds to the affiliated signal x in the circuit. $\wedge$-nodes are denoted by Greek letters using the same letter for the three $\wedge$-nodes of a ternary clause; they are depicted by $\wedge$ or a shaded triangle in the figures. While signal nodes represent an encoding bit of a signal (see Table 1 for the encoding of $L_3 = \{0,1,X\}$), $\wedge$-*nodes* denote the conjunction operation needed to model ternary clauses[1]. Every ternary clause has three associated $\wedge$-nodes that uniquely represent the clause in the IG.

$x \in L_3$	encoding	
	c_x	c_x^*
0	0	1
1	1	0
X	0	0
conflict	1	1

Table 1: Encoding of L_3

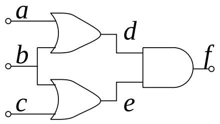
Inconsistent or conflicting signal assignments are easily detected as they are represented by $c_x = 1 \wedge c_x^* = 1$ which is expressed in the following definition:

DEFINITION 1 (*non-conflicting assignment*)
An assignment is called non-conflicting iff $c_x \wedge c_x^* \iff 0$ *holds for all signal variables* x.

Since we require non-conflicting assignments and apply a property based encoding as defined in [16], the complements $\neg c_x$ and $\neg c_x^*$ of literals c_x and c_x^* can be denoted by c_x^* and c_x, respectively.

In order to provide all structural information within the IG the set of edges E, which represent implications, is partitioned into three disjoint subsets. The set of *forward edges* E_F comprises implications from an input to an output signal of a module whereas the set of *backward edges* E_B models the opposite direction. All other implications (e.g. indirect implications) are contained in the set of *other edges* E_O. In the IG these sets are modeled by edge tags f, b, and o (tags denoting other edges are omitted in the figures). Fig. 2 shows a a simple circuit, its *CNF* representation as well as its IG model with respect to L_3. A detailed discussion on how the IG is automatically compiled for an arbitrary combinational circuit and a chosen logic may be found in [12].

- Structural:



- *CNF* for L_3:

$$\begin{array}{ll|l} (\neg c_d^* \vee c_f^*) \wedge (\neg c_e^* \vee c_f^*) \wedge (\neg c_d \vee \neg c_e \vee c_f) & \wedge & f = AND(d,e) \\ (\neg c_a \vee c_d) \wedge (\neg c_b \vee c_d) \wedge (\neg c_a^* \vee \neg c_b^* \vee c_d^*) & \wedge & d = OR(a,b) \\ (\neg c_b \vee c_e) \wedge (\neg c_c \vee c_e) \wedge (\neg c_b^* \vee \neg c_c^* \vee c_e^*) & \wedge & e = OR(b,c) \\ \iff 1 & & \end{array}$$

- Implication graph $G = (V,E)$ for $L_3 = \{0,1,X\}$:

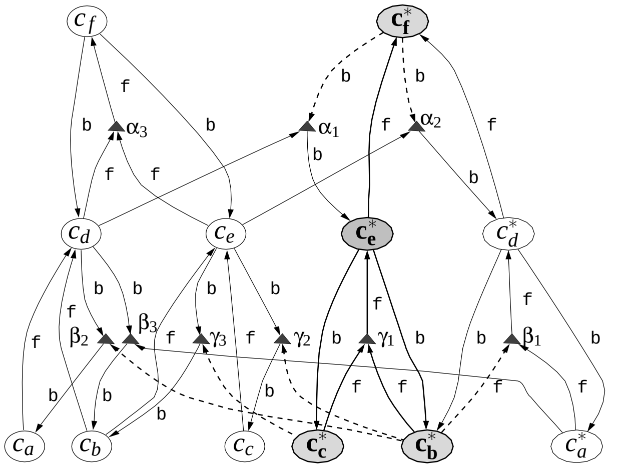


Figure 2: Circuit descriptions: structural — implication graph

3 Implication

IG based implication only requires a partial traversal of the IG. It is performed by an algorithm obeying the following rule.

RULE 1 (*direct implication* [12])
Starting from an initial set $V_I \subseteq V_S$ *of set nodes, all successor nodes* v_j *are set*

- *if node* v_j *is a* $\wedge$*-node and* ***all*** *its predecessors are set.*
- *if node* v_j *is a signal node and* ***at least one*** *predecessor is set.*

This rule is applied until no additional node can be set.

All signal nodes $c_x \in V_S$ that have been set by Rule 1 represent signal values that can be implied from the initial assignment given by V_I.

Let us use the circuit of Fig. 2 for the sake of explanation. Assigning logical value 0 to signal e corresponds to setting node c_e^* in the IG. After running the implication procedure, the following nodes are set: c_b^*, c_c^*, and c_f^*. To finally obtain the implied signal values

---

[1] As shown in [16] any clause system of a higher order can be decomposed into a system of binary and ternary clauses.

with respect to the given logic, the set nodes are decoded, i.e. we determine $b = 0$, $c = 0$, and $f = 0$. As can be seen from this example, implication terminates at $\wedge$-nodes that have only one of their predecessors set, here nodes α_1, α_2, β_1, β_2, γ_2, and γ_3. These nodes represent so-called unjustified ternary clauses that are discussed in the next section.

4 Justification

In the context of ATPG, *justification* denotes the task of finding a value assignment at the primary inputs that forces an internal signal to a required value.

Structure based tools start justification at output signals of gates which are assigned a signal value that is not controlled by its inputs. These signals are referred to as *unjustified lines*. The D-algorithm solves the problem of justification by driving a so-called *J-frontier* towards the primary inputs [1]. In order to reduce the size of the search space, PODEM constrains value assignments to the primary inputs exploiting the fact that in a circuit every internal signal can be controlled by the primary inputs [2]. In PODEM the set of primary inputs, which has to be assigned a value, is found in a *backtracing* step. During backtracing objectives are driven towards the primary inputs. Then, it is decided by implication if the requirements are met.

Clause based justification is implicitly solved when computing a satisfying assignment for the SAT problem. Since a general SAT solver does not differentiate between internal signals and primary inputs it cannot benefit from constraining optional assignments to the primary inputs. Consequently, a SAT solver has to examine a significantly larger search space. While most SAT based algorithms use a static order for variable assignments during their search for a satisfying assignment [5, 6], TEGUS, tries to mimic PODEM by ordering the clauses in a manner such that optional assignments are first made to primary input signals [7]. CGRASP, a version of the state-of-the-art SAT solver GRASP that is specialized for solving SAT instances from combinational circuits, adds an additional layer for modeling the topology of a circuit [9]. This topological layer allows the concept of a *J-frontier* to be used during justification.

Our *IG based* justification adds the advantages of PODEM to a SAT based approach since all structural information is provided by edge tags. Here, the notion of unjustified lines is replaced by *unjustified clauses* as formulated in Definitions 2 and 3.

DEFINITION 2 (*unjustified clause* [12])
A clause $C = c_1 \vee c_2 \vee \ldots \vee c_n$ is called unjustified iff all literals $c_1, c_2, \ldots, c_n$ do not evaluate to 1 and at least one complement c_i^* of a literal c_i is 1.

DEFINITION 3 (*justification* [12])
Let $c_1, c_2, \ldots, c_m$ be some unspecified literals in a clause $C = c_1 \vee c_2 \vee \ldots \vee c_n$ that is unjustified and let $V_1, V_2, \ldots, V_m$ denote assigned values. Then, the set of non-conflicting assignments $J = \{c_1 = V_1, c_2 = V_2, \ldots, c_m = V_m\}$ is called a justification of clause C, if the value assignments in J make C evaluate to 1.

Unjustified ternary clauses[2] are found in the IG without effort. They are represented by $\wedge$-nodes that have only one of their two predecessors set. A complete set of justifications J_c for an unjustified clause C is easily given by $J_c = \{\{c_1 = 1\}, \{c_2 = 1\}, \ldots, \{c_m = 1\}\}$. As only ternary clauses can be unjustified in our approach, J_c always consists of exactly two justifications.

We will now explain how these two justifications can be derived in the IG with Fig. 3. The given ternary clause $c_x \vee c_y \vee c_z$ is unjustified due to an assignment of $c_x^* = 1$. This is indicated by the two $\wedge$-nodes α_1 and α_2 that have only one predecessor (c_x^*) set. Here, the ternary clause can be justified by setting c_z or c_y to 1. Let us reconsider that the subgraph denoting the ternary clause $c_x \vee c_y \vee c_z$ is a straightforward graphical representation of the formulae: $c_x \vee c_y \vee c_z \Longleftrightarrow c_x^* \wedge c_y^* \rightarrow c_z \Longleftrightarrow c_x^* \wedge c_z^* \rightarrow c_y \Longleftrightarrow c_y^* \wedge c_z^* \rightarrow c_x$ [12]. Then, it becomes apparent that both possible justifications in J_c are found in the consequents of those implications which have the literal making the clause unjustified, i.e. c_x^*, in their antecedent. These consequents correspond to the successors of the two $\wedge$-nodes α_1 and α_2.

In order to realize PODEM based justification in the IG we adopt the concept of unjustified ternary clauses to guide the backtracing process. So as to lead our search towards the primary inputs we only work on a subgraph $G_B = (V, E_B)$ of the IG $G = (V, E)$. Thereby, we extract a *directed acyclic graph (DAG)* from the originally cyclic IG. Please observe, that in our implementation $G_B = (V, E_B)$ is not represented by an additional graph but is implicitly modeled in $G = (V, E)$ by means of the backward tags. This also holds for graph G_F^g needed for propagation in Section 5.

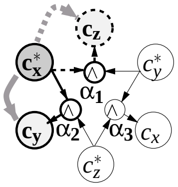
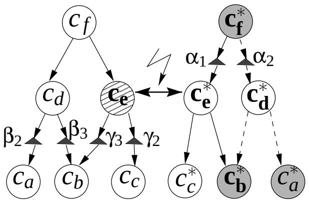


Figure 3: Unjustified ternary clause

For the circuit of Fig. 2, we obtain the DAG G_B shown in Fig. 4. Starting from an initial objective (requirement) o_I, i.e. an internal signal s_I and its associated signal node c_I that is to be forced to a certain logic value, backtracing traverses G_B in a depth first manner towards the primary inputs obeying the following set of rules:



$G_B = (V, E_B)$

Figure 4: Backtracing in G_B

RULE 2 (*backtracing*)
Let the objective o_i be driven to node $v_i \in V$. $suc_S(v_i) \subseteq V_S$ and $suc_\wedge(v_i) \subseteq V_\wedge$ denote the succeeding signal and $\wedge$-nodes in G_B, respectively. Then the objective o_i is driven to the following nodes:

- ***all** signal nodes $v_j \in suc_S(v_i)$.*
- ***one** $\wedge$-node $v_j \in suc_\wedge(v_i)$ which is selected according to a precomputed controllability measure. Nodes v_j, which are succeeded by a signal node c_x whose associated complement node c_x^* is set, are not selected.*

This rule is applied until no further propagation of objectives is possible, that is all objectives have reached a primary input.

As soon as a signal node belonging to a primary input is reached by backtracing, it is set and the implication procedure of Section 3 is invoked. If the unjustified clause becomes justified by implying from the injected assignment we have found a justification. If a conflict is caused during implication this assignment has to be reversed (*backtracking*) by setting its complement node and restarting implication. On the one hand, if all assignments at the PIs cause a conflict even after being reversed it can be deduced that the examined signal

---

[2] If a binary clause is unjustified it reduces to a unary clause. Unary clauses represent necessary assignments (implied signal values).

cannot be forced to the demanded logic value. On the other hand, if the computed non-conflicting assignment does not justify the unjustified clause, backtracing from this clause starts again. Thereby, the search space is implicitly worked off by making assignments only at the primary inputs.

Let us explain backtracing according to Rule 2 with help of graph G_B found in Fig. 4. We assume that signal nodes c_e and c_f^* are set and c_f^* should be justified. Backtracing starts at node c_f^* which makes clause $C_\alpha = c_d^* \vee c_e^* \vee c_f$ unjustified. We drive the objective along the dashed path via $\wedge$-node α_2 to node c_d^*. The alternative path via $\wedge$-node α_1 is not chosen as the complement node c_e of its successor c_e^* is set. From c_d^* the objective is further driven to primary input nodes c_b^* and c_a^*. As can be seen from $G = (V,E)$ in Fig. 2, implication from these nodes sets c_d^* and thereby justifies signal c_f^* and clause C_α, respectively.

Our approach to justification takes advantage of bit-parallelism in two different ways. First, several justification problems can be solved simultaneously by processing a different justification problem in each bit-slice (*and-parallelism*). This is exploited during fault parallel ATPG for easy-to-detect faults. Second, alternative decisions can be examined simultaneously in different bit-slices (*or-parallelism*). This method is advantageous when dealing with hard-to-detect or redundant faults. It can also be exploited for derivation of *indirect implications* [12].

5 Propagation

Propagation denotes the task of making a signal change at an internal signal observable at at least one of the primary outputs. This is achieved by sensitizing a propagation path and finally justifying the injected sensitizing assignments.

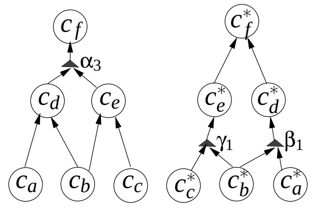
Structure based tools solve the problem of propagation by driving a so-called *D-frontier* towards the primary outputs (*D-drive*) [1]. A first group explicitly considers *multiple-path propagation* and employs a 5-valued logic [1, 2, 3, 4]. Another group relies on a *single-path propagation* strategy that implicitly considers multiple-path propagation [11, 17, 18]. This group applies the 9-valued logic [17] and the *split-circuit model* [11].

Clause based approaches rely on the split circuit model. They translate the D-drive by adding additional clauses (*active clauses*) to the *CNF* which represent structural knowledge about possible propagation paths. This topological information accelerates the solution of the SAT problem but adds complexity to formula extraction. As a different set of active clauses has to be added for every processed fault during ATPG, often the time for extracting the formula surpasses the one needed to solve it [7, 10]. Moreover, due to the lack of topological information available in the *CNF* the heuristics known for structure based approaches are hard to incorporate.

$x \in L_9$	encoding			
	good		faulty	
	c_x	c_x^*	$\hat{c}_x$	$\hat{c}_x^*$
0	0	1	0	1
1	1	0	1	0
X	0	0	0	0
D	1	0	0	1
$\overline{D}$	0	1	1	0
$G0$	0	1	0	0
$G1$	1	0	0	0
$F0$	0	0	0	1
$F1$	0	0	1	0

Table 2: Encoding of L_9

IG based propagation is as efficient as structure based approaches since the IG contains the complete topological information of a circuit. It is also much simpler because of the uniformity of the graph consisting of only two different node types instead of a multitude of gate types. As it relies on the split circuit model, the IG for propagation is simply obtained by duplicating the respective graph for the 3-valued logic L_3. That is, we obtain two disjoint isomorphic graphs $G^g = (V^g, E^g)$ and $G^f = (V^f, E^f)$. While G^g models the good (fault-free) circuit, G^f represents the faulty circuit. Both graphs G^g and G^f are merged such that the composite IG $G = (V^g \cup V^f, E^g \cup E^f)$ is obtained. This graph represents the circuit with respect to the 9-valued logic L_9 which requires four signal nodes for its enconding (see Table 2).



$G_F = (V, E_F)$

Figure 5: Propagation in G_F^g

The pair $c_x \in V^g$ and $c_x^* \in V^g$ encodes the 3-valued logic value of a signal x in the good circuit and the pair $\hat{c}_x \in V^f$ and $\hat{c}_x^* \in V^f$ the corresponding one in the faulty circuit. Again a conflict is indicated by an assignment that sets complementary nodes, i.e. $c_x = 1 \wedge c_x^* = 1$ or $\hat{c}_x = 1 \wedge \hat{c}_x^* = 1$, simultaneously. Similarly to justification, a DAG $G_F^g = (V^g, E_F^g)$, which is obtained by removing all edges except for the forward edges from G^g, is extracted in order to guide propagation. Thus, addition of active clauses becomes unnecessary which increases the efficiency of our approach. Fig. 5 shows $G_F^g = (V^g, E_F^g)$ for the circuit of Fig. 2.

Propagation starts by injecting the logic value D ($\overline{D}$) at an initial signal s_I that should be observed. In the IG $G = (V^g \cup V^f, E^g \cup E^f)$ this is done by setting the nodes c_I and $\hat{c}_I^*$ (c_I^* and $\hat{c}_I$) corresponding to s_I. Then, the propagation procedure traverses G_F^g in a depth first manner obeying the following set of rules:

RULE 3 (*propagation*)
Let the initial signal s_I be observable at signal s_i, i.e. $(s_i = D) \vee (s_i = \overline{D})$ and $(c_i \wedge \hat{c}_i^*) \vee (c_i^* \wedge \hat{c}_i) \iff 1$, respectively. Let $suc_S(v_i) \subseteq V_S^g$ and $suc_\wedge(v_i) \subseteq V_\wedge^g$ denote all succeeding signal and $\wedge$-nodes of a node v_i in $G_F^g = (V^g, E_F^g)$, respectively. Then, signal s_I is made observable at a succeeding signal s_j by:

- *selecting **one** node $v_j \in suc_S(c_i) \cup suc_\wedge(c_i)$ according to a pre-computed observability measure.*
  Nodes $v_j = c_j \in suc_S(c_i)$ whose associated complement node c_j^* is set and nodes $v_j \in suc_\wedge(c_i)$, which are succeeded by a signal node c_x whose associated complement node c_x^* is set, are not selected.
  ***if** $v_j \in suc_S(c_i)$, i.e. v_j denotes a signal node c_j, then set its associated complement node $\hat{c}_j^*$ in G^f.*
  ***if** $v_j \in suc_\wedge(c_i)$ then set its succeeding signal node c_k as well as its associated complement node $\hat{c}_k^*$ in G^f.*
- *implying from all set nodes in G and thereby injecting the sensitizing assignments. If implication results in a conflict, all assignments are reverted and another node $v_j \in suc_S(c_i) \cup suc_\wedge(c_i)$ is selected. If all nodes v_j yield a conflict, backtrack to previous decision.*

This rule is applied until a primary output is reached or all selections of $v_j \in suc_S(c_I) \cup suc_\wedge(c_I)$ result in a conflict.

Propagation according to Rule 3 is related to the method for single-

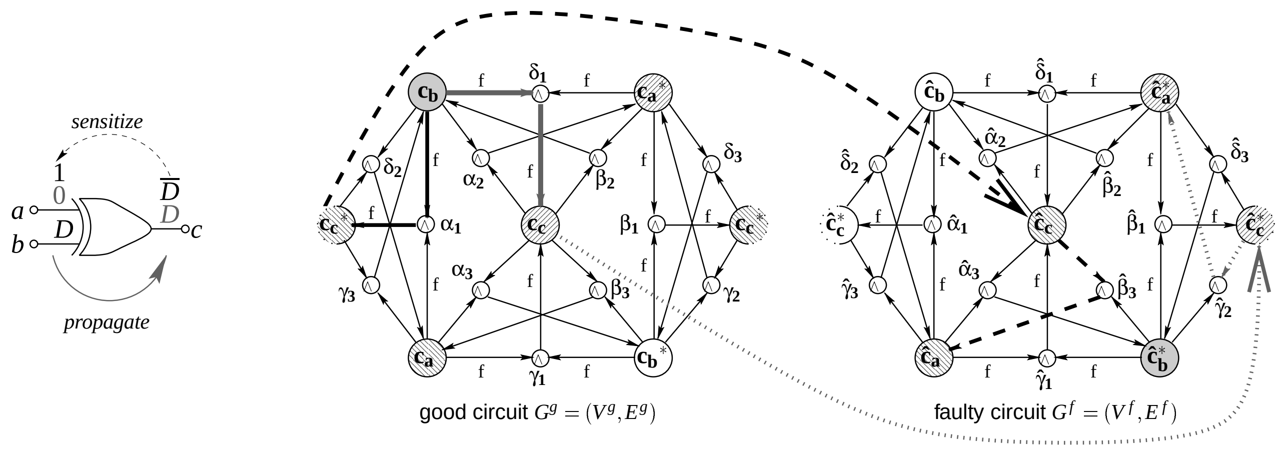


Figure 6: Propagation over an XOR-gate

path propagation proposed in [18] as it implicitly generates the presented necessary and sufficient sensitizing conditions for the gate model. In the IG model, the sensitization of gates corresponds to justification of unjustified ternary clause and subsequent implication. That is, if we propagate via an $\wedge$-node we thereby justify the corresponding unjustified clause. Implication from this justification yields the value assignment necessary to “sensitize” the $\wedge$-node. Please observe, that fanout nodes in G_F^g may be caused either by fanout signals in the circuit or by the logic function of a gate such as the XOR-gate discussed next.

Let us now explain how a signal change is propagated according to Rule 3 with help of Fig. 6 showing the IG $G = (V^g \cup V^f, E^g \cup E^f)$ for an XOR-gate with respect to L_9. (The logic function of an XOR-gate with respect to L_3 is represented by four ternary clauses and no binary clause.) Nodes c_c^* ($\hat{c}_c^*$) are drawn twice in Fig. 6 in order to provide a clearer representation of G. To the human reader the IG of Fig. 6 may appear more complicated than the gate level representation of an XOR-gate. Yet, the IG model is optimal for being worked on efficiently by a computer.

Let us assume that a change from logical 1 to 0 at signal b should be propagated. We start by setting nodes c_b and $\hat{c}_b^*$ in the IG which corresponds to assigning logic value D to b. As $suc_S(c_b) = \emptyset$, first a node $v \in suc_\wedge(c_b) = \{\alpha_1, \delta_1\}$ is selected. If we select $\wedge$-node α_1 we follow the path $c_b - \alpha_1 - c_c^*$ in G_F^g, which is indicated by bold black arrows, and set node c_c^*. Thereby, clause $C_\alpha = c_a^* \vee c_b^* \vee c_c^*$ is justified. Next, we have to set the associated complement node of c_c^* in G^f, that is node $\hat{c}_c$. Finally, nodes c_a and $\hat{c}_a$ are set by implication. So, after propagation along $c_b - \alpha_1 - c_c^*$ signal c is assigned $\overline{D}$ ($c_c^* \wedge \hat{c}_c \iff 1$). The required sensitizing assignment logical 1 ($c_a \wedge \hat{c}_a \iff 1$) was automatically injected at signal a by calling the implication procedure. The alternative propagation along path $c_b - \delta_1 - c_c$ is marked by the bold grey arrows. It assigns logic value D to signal c ($c_c \wedge \hat{c}_c^* \iff 1$) and sensitizes the path by setting a to logical 0 ($c_a^* \wedge \hat{c}_a^* \iff 1$).

This example shows how both ways to sensitize an XOR-gate are modeled by selecting a different propagation path in the IG. Thus, alternative ways to propagate a signal change that originate from the logic function of a gate are dealt with in the same manner as choosing different propagation paths at a fanout stem in a circuit. As a consequence, our approach does not have to consider different sensitization conditions for different module types as structure based methods do. The resulting uniformity of our graph algorithm for propagation allows effective exploitation of bit-parallelism in two ways. First, different possible propagation paths for a fault effect can be simultaneously investigated in different bit-slices (*or-parallelism*). Second, several independent propagation problems may be solved at the same time (*and-parallelism*). These techniques can also be exploited for derivation of so-called *D-implications* [19].

6 Experimental results

Fast justification and propagation in the IG have been included into the implication engine of [12]. So as to validate the effectiveness of the proposed methods, they are incorporated into our ATPG tool TIP [20, 21] that is capable of handling various fault models. All experiments were run on a Digital Alpha 4100 5/533 (SPECint95base 15.0) using ISCAS85/89 benchmark circuits. There were no aborted faults unless explicitly stated and no random patterns were used.

	TEGUS[7] time in [s]					TIP time in [s]		
circuit	total	total - FSIM	SAT	CNF	FSIM	total	ATPG	IG
c432	0.61	0.52	0.14	0.38	0.08	0.02	0.02	0.00
c499	0.72	0.58	0.12	0.46	0.11	0.05	0.03	0.02
c880	0.83	0.65	0.13	0.52	0.14	0.02	0.02	0.00
c1355	2.06	1.49	0.28	1.21	0.54	0.23	0.20	0.03
c1908	2.98	2.37	0.63	1.74	0.57	0.42	0.37	0.05
c2670	9.76	8.87	3.37	5.50	0.76	0.43	0.38	0.05
c3540	26.10	23.80	14.21	9.59	2.20	1.13	0.85	0.28
c5315	13.39	10.59	1.69	8.90	2.56	0.52	0.40	0.12
c6288	66.45	57.90	40.62	17.28	8.41	0.18	0.17	0.02
c7552	20.76	16.21	3.75	12.46	4.23	1.80	1.75	0.05
s1269	1.60	1.18	0.21	0.97	0.38	0.03	0.03	0.00
s3271	3.29	2.03	0.26	1.77	1.13	0.12	0.10	0.02
s4863	7.84	3.76	0.64	3.12	3.96	0.27	0.22	0.05
s5378	7.15	5.16	0.50	4.66	1.79	0.43	0.40	0.03
s9234	47.42	36.26	9.87	26.39	10.70	7.65	6.18	1.47
s13207	74.46	37.92	2.59	35.33	35.76	5.13	5.05	0.08
s15850	209.58	80.20	8.01	72.19	128.42	3.37	3.25	0.12
s35932	674.73	253.92	2.43	251.49	418.03	29.34	29.11	0.23
s38417	755.98	267.16	9.04	258.12	486.00	31.12	30.74	0.38
s38584	896.69	294.04	3.97	290.07	599.82	33.31	33.02	0.28
geo. av.	15.89	10.20				0.66		

Table 3: Stuck-at ATPG running fault simulation every 64 patterns

Tables 3 and 4 present results for combinational stuck-at ATPG. In a first experiment, ATPG was run in combination with fault simulation; that is, every 64 patterns, which were generated by ATPG,

	TEGUS[7] time in [s]			CGRASP[9] time in [s]			TIP time in [s]		
circuit	total	SAT	CNF	total	SAT	CNF	total	ATPG	IG
c432	2.05	0.53	1.48	3.70	1.48	2.21	0.07	0.05	0.02
c499	5.44	1.27	4.08	5.56	2.06	3.49	0.17	0.17	0.00
c880	2.16	0.41	1.69	5.48	2.13	3.34	0.07	0.07	0.00
c1355	11.73	2.49	9.12	31.98	12.90	19.08	0.82	0.82	0.00
c1908	18.75	3.77	14.82	41.79	22.95	18.84	1.67	1.62	0.05
c2670	27.88	8.65	18.83	32.93	23.38	9.55	1.57	1.52	0.05
c3540	94.94	37.47	57.06	102.53	57.14	45.39	6.55	6.27	0.28
c5315	48.90	7.94	40.28	77.26	54.85	22.41	5.32	5.29	0.03
c6288	473.61	244.02	228.87	566.82	319.37	247.44	39.44	39.40	0.03
c7552	104.93	20.63	83.16	214.04	169.48	44.55	13.94	13.91	0.03
s1269	5.21	0.95	4.16				0.25	0.25	0.00
s3271	9.55	1.34	7.92				1.00	0.98	0.02
s4863	63.61	18.47	44.68				6.60	6.49	0.12
s5378	25.84	3.08	22.16				2.80	2.77	0.03
s9234	215.05	134.63	79.41				19.21	17.84	1.37
s13207	137.01	10.51	124.63				33.31	33.23	0.08
s15850	282.60	32.62	247.63				28.33	28.20	0.13
s35932	749.79	10.05	732.84				238.55	238.35	0.20
s38417	1035.19	42.88	984.26				175.10	174.80	0.30
s38584	920.57	20.73	892.09				341.08	340.81	0.27
geo. av.	51.40			36.97			4.79		

Table 4: Stuck-at ATPG without running fault simulation

fault simulation was started. The achieved results for TIP are found in columns 7 to 9 of Table 3. While column 7 provides the total time for both ATPG and construction of the IG, columns 8 and 9 give the time for each individual step. The time for IG construction includes the time required for deriving some indirect implications. So as to prove the robustness of our approach we conducted a second experiment. Here, ATPG was run for every fault in a circuit (after fault collapsing) without using fault simulation. The corresponding results for TIP are given in columns 8 to 10 of Table 4. The geometric average of total run times may be found in the last row of Tables 3 and 4.

In order to demonstrate the quality of IG based ATPG, we compare the obtained results with the SAT based approaches TEGUS [7] and CGRASP [9] that mark the state-of-the-art. So as to allow a fair comparison we compiled the version of TEGUS that comes with the synthesis tool SIS [22] using the same compiler settings and machine as for TIP. The results for CGRASP have been taken from [9]. They are scaled to execution times on a Digital Alpha 4100 5/533 using SPECint95base ratios as the experiments in [9] have been carried out on a Pentium-II/266 machine (SPECint95base 10.8). The superiority of our approach can be seen from the experimental data shown in Tables 3 and 4. While column 2 of Table 3 gives the total run time for TEGUS, columns 4, 5, and 6 provide the times for solving the SAT formulae, extracting the *CNF* from the circuit, and running fault simulation, respectively. Since the time needed for fault simulation in TEGUS is quite substantial, while it is negligible in TIP, we also give the total run time without fault simulation in column 3. As can be seen from the data our approach is one order of magnitude faster than TEGUS. In Table 4, columns 2 to 4 and columns 5 to 7 provide the corresponding data for TEGUS and CGRASP, respectively, when running ATPG without fault simulation. Again, a comparison with the results for TIP in columns 8 to 10 demonstrates the high effectiveness of IG based implication, justification, and propagation.

In case of stuck-at ATPG the time for graph construction in TIP (columns IG) may be considered as being corresponding to the time needed for *CNF* extraction in TEGUS and CGRASP (columns *CNF*). The time required by justification, propagation, and implication in TIP (columns ATPG) corresponds to the time needed for solving the extracted SAT formulae in TEGUS and CGRASP (columns SAT). As can be seen from Tables 3 and 4, the proposed IG based approach provides significantly better performance compared to general SAT solvers even if the latter are specialized for combinational circuits. Furthermore, the experimental data gives evidence that often the time needed for *CNF* extraction is prohibitively high in [7, 9].

Since TEGUS has been proposed as a benchmark program for ATPG tools, an extensive comparison with ATPG tools that mark the state-of-the-art is made in [7]. It is shown that TEGUS is faster and more robust than previously published approaches. Therefore, the experimental results in Tables 3 and 4 establish that TIP also beats these tools in terms of speed and robustness.

Next in Tables 5 and 6, we provide results for ATPG targeting nonrobust and robust path delay faults. When dealing with path delay faults our tool TIP uses the IG for fast implication and justification. Explicit propagation of fault effects is not required in path delay ATPG as it is inherent in the fault model.

Columns 9 to 11 in Table 5 provide the number of detected faults, the number of faults that are proven untestable, and the required run time, respectively, when running TIP for nonrobust path delay ATPG using a 3-valued logic. The total number of faults in a circuit is given in column 2. Again, no faults were aborted. A comparison of the results with TRAN (columns 3 to 5) and TSUNAMI-D (columns 6 to 9) shows that TIP clearly outperforms the SAT based TRAN but is slower than the BDD based TSUNAMI-D.[3] TSUNAMI-D, however, cannot process the circuits having the most paths as it suffers from the excessive memory requirements of its BDDs.

In Table 6 you find the corresponding results for robust path delay ATPG. Here, the results of TIP found in columns 13 to 16 are obtained using an IG for a 10-valued logic. The comparison with the SAT based approach of [10] (columns 3 to 5), TRAN (columns 6 to 9), and TSUNAMI-D (columns 10 to 12) show again that TIP is the fastest approach that can process all circuits.[3] As TRAN and TIP aborted some faults they are listed in columns 8 and 15, respectively.

---

[3] The results for [10], TRAN and TSUNAMI-D are scaled to execution times on a Digital Alpha 4100 5/533 using SPECint95base ratios.

		Tran[23]			Tsunami-D[24]			Tip		
circuit	faults	detected	untestable	time in [s]	detected	untestable	time in [s]	detected	untestable	time in [s]
s510	738	738	0	2.22	738	0	0.03	738	0	0.10
s382	800	734	66	0.55	704	96	0.02	734	66	0.02
s526	820	720	100	2.04	708	112	0.02	720	100	0.07
s820	984	984	0	5.04	984	0	0.05	984	0	0.27
s832	1012	996	16	5.08	996	16	0.05	996	16	0.30
s1488	1924	1916	8	14.13	1916	8	0.12	1916	8	0.93
s1494	1952	1927	25	13.82	1926	26	0.12	1927	25	1.02
s953	2312	2312	0	10.14	2266	0	0.13	2312	0	0.35
s641	3488	2270	1218	15.11	2096	1392	0.30	2270	1218	0.13
s1196	6196	3759	2437	44.84	3708	2486	0.36	3759	2437	0.80
s1238	7118	3684	3434	47.76	3663	3453	0.38	3684	3434	0.93
c880	17284							16652	632	0.82
s5378	27084				19413	7671	2.60	21928	5156	3.10
s3271	38388							19292	19096	1.75
s3384	39582							31966	7616	3.30
s713	43624				2066	41558	0.83	4922	38702	0.22
s1269	79140							33382	45758	3.03
s1423	89452				33981	55471	17.69	45198	44254	2.48
s35932	394282				38372	355910	6.94	58657	335625	40.52
s9234	489708				38621	451087	16.08	59854	429854	12.65
c432	583652							15855	567797	2.20
c499	795776							367744	428032	27.07
c2670	1359920							130626	1229294	11.35
c7552	1452988							277244	1175744	570.38
c1908	1458114							355168	1102946	27.69
s38584	2161446				170291	1991151	60.40	334927	1826519	613.29
c5315	2682610							342117	2340493	132.48
s13207	2690738				162798	2527840	68.88	476145	2214593	293.54
s38417	2783158							1138194	1644964	752.87
c1355	8346432							1110304	7236128	42.69
c3540	57353342							1202584	56150758	1762.70
s15850	329476092							10782994	318693098	5791.82

Table 5: ATPG for nonrobust path delay faults

circuit	nonrobust	robust
s713	6.67	4.41
s838	2.31	3.22
s938	4.46	8.91
s991	7.16	1.36
s1269	3.16	1.76
s1423	4.36	8.41
s3271	2.46	4.08
s5378	5.80	4.53
s9234	3.85	2.13
s13207	0.43	2.11
s15850	5.07	2.14
average	4.16	3.91

Table 7: Speedup $t_{\text{single}}/t_{\text{parallel}}$ due to bit-parallel justification

In a final experiment, we investigated the speedup that can be achieved by exploiting bit-parallelism in justification and propagation. Table 7 gives the obtained speedup factor $t_{\text{single}}/t_{\text{parallel}}$ when running nonrobust and robust path delay ATPG. Here, t_{single} denotes the time required for justification when using only one bit, whereas t_{parallel} represents the corresponding time when exploiting full 64 bit words. The results show that the exploitation of and-parallel as well as or-parallel methods in Tip yields an average speedup of 4.

7 Conclusion

We have proposed fast IG based justification and propagation. Working in the IG model, the complex functional operations of justification and propagation could be reduced to significantly simpler graph algorithms. It has been shown how the uniformity of graph operations in the IG allows efficient and effective exploitation of bit-parallel techniques. Experimental results for stuck-at and path delay ATPG confirm the effectiveness of our approach. The proposed techniques, which are currently integrated into a new object-oriented framework for logic synthesis and verification, can also be applied to Boolean equivalence checking [12], optimization of netlists [12], timing analysis or retiming (reset state computation).

Acknowledgements

The authors are very grateful to Prof. Kurt J. Antreich for many valuable discussions and his advice. They like to thank Hannes Wittmann and Manfred Henftling for developing the early versions of the ATPG tool.

References

[1] J. P. Roth, W. G. Bouricius, and P. R. Schneider, “Programmed algorithms to compute tests to detect and distinguish between failures in logic circuits,” *IEEE Transactions on Electronic Computers*, vol. 16, pp. 567–580, Oct. 1967.

[2] P. Goel, “An implicit enumeration algorithm to generate tests for combinational logic circuits,” *IEEE Transactions on Computers*, vol. 30, pp. 215–222, Mar. 1981.

[3] H. Fujiwara, “FAN: A fanout-oriented test pattern generation algorithm,” in *IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 671–674, June 1985.

[4] M. H. Schulz, E. Trischler, and T. M. Sarfert, “SOCRATES: A highly efficient automatic test pattern generation system,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 7, pp. 126–137, Jan. 1988.

		[10]			Tran[23]				Tsunami-D[24]			Tip			
circuit	faults	tested	untest.	t in [s]	tested	untest.	abr.	t in [s]	tested	untest.	t in [s]	tested	untest.	abr.	t in [s]
s510	738	729	9	0.53	729	9	0	2.65	729	9	0.05	729	9	0	0.40
s382	800	667	133	0.23	667	133	0	1.14	667	133	0.03	667	133	0	0,10
s526	820	694	126	0.30	694	126	0	2.21	694	126	0.03	694	126	0	0.22
s820	984	980	4	0.76	980	4	0	6.45	980	4	0.06	980	4	0	1.20
s832	1012	984	28	0.83	984	28	0	6.63	984	28	0.07	984	28	0	1.32
s444	1070	586	484	0.31	586	484	0	0.73	586	484	0.04	586	484	0	0.12
s1488	1924	1875	49	1.73	1875	49	0	16.67	1875	49	0.16	1875	49	0	4.58
s1494	1952	1882	70	1.76	1882	70	0	16.09	1882	70	0.16	1882	70	0	4.77
s953	2312	2302	10	2.11	2302	10	0	13.56	2256	10	0.22	2302	10	0	1.53
s641	3488	1979	1509	3.00	1979	1509	0	14.56	1979	1509	0.45	1979	1509	0	0.38
s1196	6196	3581	2615	12.73	3581	2614	1	60.56	3579	2615	0.73	3581	2615	0	4.20
s1238	7118	3589	3529	15.60	3589	3529	0	66.18	3587	3529	0.74	3589	3529	0	4.32
c880	17284											16083	1201	0	4.24
s5378	27084	18656	8428	44.93					18656	8428	5.18	18656	8428	0	12.18
s3271	38388											7707	30681	0	17.20
s3384	39582											16766	22724	92	97.22
s713	43624	1184	42440	12.04					1184	42440	1.16	1184	42440	0	0.27
s1269	79140											10182	68958	0	25.97
s1423	89452	28696	60756	110.02					28696	60756	23.05	28696	60756	0	8.95
s35932	394282								21783	372499	16.31	21783	372499	0	480.78
s9234	489708	21389	468319	808.62					21389	468319	25.43	21389	468319	0	52.27
c432	583652											3730	579922	0	36.62
c499	795776											133395	571634	90747	4255.85
c2670	1359920											15370	1344550	0	16.40
c7552	1452988											86251	1366411	326	4086.13
c1908	1458114											97588	1308584	51942	5335.95
s38584	2161446								92235	2069207	138.11	92239	2069207	0	773.88
c5315	2682610											81435	2600249	926	6821.10
s13207	2690738								27503	2663135	139.91	27603	2663135	0	108.11
s38417	2783158											598062	2185096	0	3487.41
c1355	8346432											22784	8323648	0	42.74
c3540	57353342											88408	57264453	481	6887.76
s15850	329476092											182673	329293419	0	646.04

Table 6: ATPG for robust path delay faults

[5] T. Larrabee, “Test pattern generation using Boolean satisfiability,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 11, pp. 4–15, Jan. 1992.

[6] S. T. Chakradhar, V. D. Agrawal, and S. G. Rothweiler, “A transitive closure algorithm for test generation,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, pp. 1015–1028, July 1993.

[7] 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*, vol. 15, pp. 1167–1176, Sept. 1996.

[8] J. P. M. Silva and K. A. Sakallah, “GRASP — a new search algorithm for satisfiability,” in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 220–227, Nov. 1996.

[9] 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.

[10] C.-A. Chen and S. K. Gupta, “A satisfiability-based test generator for path delay faults in combinational circuits,” in *ACM/IEEE Design Automation Conference (DAC)*, pp. 209–214, June 1996.

[11] W.-T. Cheng, “Split circuit model for test generation,” in *ACM/IEEE Design Automation Conference (DAC)*, vol. 25, pp. 96–101, June 1988.

[12] P. Tafertshofer, A. Ganz, and M. Henftling, “A SAT-based implication engine for efficient atpg, equivalence checking, and optimization of netlists,” in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 648–655, Nov. 1997.

[13] R. T. Stanion, D. Bhattacharya, and C. Sechen, “An efficient method for generating exhaustive test sets,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, pp. 1516–1525, Dec. 1995.

[14] B. Rohfleisch, B. Wurth, and K. Antreich, “Logic clause analysis for delay optimization,” in *ACM/IEEE Design Automation Conference (DAC)*, (San Francisco), pp. 668–672, June 1995.

[15] A. Ganz and P. Tafertshofer, “An efficient framework for functional path analysis,” in *ACM/IEEE Int. Workshop on Timing Issues in the Spec. and Syn. of Dig. Systems*, Mar. 1999.

[16] P. Tafertshofer, A. Ganz, and M. Henftling, “A SAT-based implication engine,” Tech. Rep. TUM-LRE-97-2, Technical University of Munich, Apr. 1997.

[17] P. Muth, “A nine-valued circuit model for test generation,” *IEEE Transactions on Computers*, vol. 25, pp. 630–636, June 1976.

[18] M. Henftling, H. C. Wittmann, and K. J. Antreich, “A single-path-oriented fault-effect propagation in digital circuits considering multiple-path sensitization,” in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, (San Jose, California), pp. 304–309, Nov. 1995.

[19] W. Kunz and D. K. Pradhan, “Recursive learning; a new implication technique for efficient solutions to cad problems — test, verification, and optimization,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, pp. 1143–1158, Sept. 1994.

[20] M. Henftling, H. Wittmann, and K. J. Antreich, “A formal non-heuristic atpg approach,” in *European Design Automation Conference with EURO-VHDL (EURO-DAC)*, pp. 248–253, Sept. 1995.

[21] M. Henftling and H. Wittmann, “Bit parallel test pattern generation for path delay faults,” in *European Design and Test Conference (ED&TC)*, (Paris), pp. 521–525, Mar. 1995.

[22] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli, “SIS: A system for sequential circuit synthesis,” Memorandum UCB/ERL M92/41, Electronics Research Laboratory, University of California, Berkeley, CA 94720, May 1992.

[23] S. T. Chakradhar, M. A. Iyer, and V. D. Agrawal, “Energy models for dealy testing,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, pp. 728–739, June 1995.

[24] D. Bhattacharya, P. Agrawal, and V. D. Agrawal, “Test generation for path delay faults using binary decision diagrams,” *IEEE Transactions on Computers*, vol. 44, pp. 434–447, Mar. 1995.