SPIRIT: A Highly Robust Combinational Test Generation Algorithm

Emil Gizdarski, *Member, IEEE* and Hideo Fujiwara, *Fellow, IEEE*

***Abstract*—In this paper, an efficient test pattern generation (TPG) algorithm for combinational circuits based on the Boolean satisfiability method (SAT) is presented. The authors propose a new data structure for the complete implication graph that increases the precision of implication process. Next, they examine approaches like a single-cone processing, single path-oriented propagation, and backward justification and show that they are efficient to improve robustness of TPG algorithms. Finally, the authors propose efficient techniques and heuristics for these approaches. The resultant automatic test pattern generation system, called SPIRIT (Satisfiability Problem Implementation for Redundancy Identification and Test generation), combines the flexibility of the SAT-based TPG algorithms with the efficiency of the structural TPG algorithms. Experimental results demonstrate the robustness of the proposed TPG algorithm. Without fault simulation, SPIRIT is able to achieve 100% fault efficiency for a large set of benchmark circuits in a reasonable amount of time.**

***Index Terms*—Boolean satisfiability, static and dynamic learning, stuck-at faults, test generation.**

I. Introduction

In recent years, substantial progress has been achieved in the field of electronic design automation (EDA) using the Boolean satisfiability method (SAT). Originally motivated by the work of Larrabee [1] in test generation, the SAT method has been implemented to many other EDA applications. In general, the SAT-based algorithms have two basic parameters: 1) *performance* associated with the average CPU time per instance and 2) *robustness* associated with the probability that the algorithm finds a solution or proves that solution does not exist within given bounds, number of backtracks, and/or CPU time per instance. In practical automatic test pattern generation (ATPG), the performance and robustness are considered as two contradicting parameters. For example, using costly TPG techniques to reduce the worst case performance of a TPG algorithm may also decrease the average-case performance, while the reverse case may decrease the robustness of the TPG algorithm. As a result, the practical TPG algorithms are incomplete in order to keep the performance of ATPG systems as high as possible. However, the increasing size and complexity of the integrated circuits as well as on going changes in design-for-test technology will require more attention on the robustness of the TPG algorithms in the future. ATPG has become a basic process that determines efficiency of many other processes. For example, the robustness of the TPG algorithms is a critical parameter for input reduction [2]. In this case, just one aborted fault may cause a failure to identify compatibility between two inputs (compatibility classes) that could reduce by 50% the test application time for the counter-based exhaustive built-in self-testing (BIST). This motivates us to work on the robustness of the TPG algorithms. To do so, we study the most successful TPG algorithms: (structural) PODEM [3], FAN [4], TOP [5], SOCRATES [6], [7], ATPG [8], and ATOM [9], (algebraic) Nemesis [1], TRAN [10], TEGUS [11], and TIP [12], [13].

The most important techniques implemented in SPIRIT (Satisfiability Problem Implementation for Redundancy Identification and Test generation) are briefly described as follows:

- 9-V [14] and 16-V [15] algebra for more precise implication process and search space reduction;
- X-path check [3] for early detection of inconsistency during propagation;
- unique sensitization [4] and structural dominators [5]–[7] as efficient dynamic learning techniques during propagation;
- static learning [6] as an efficient technique for deriving new dependencies between signals during preprocessing;
- recursive learning [16] as an efficient dynamic learning technique during propagation and justification;
- single-cone processing [17] and single path-oriented propagation [8], [12] as efficient approaches for search space reduction;
- backward justification [8], [18] as an alternative of forward justification making decisions only on the primary inputs [3], headlines [4], and implication nodes [19];
- Boolean satisfiability method as an elegant model of the TPG problem [1] allowing some powerful learning techniques to be used during the branch and bound search.

The processes that determine the efficiency of TPG algorithms are implication, propagation, justification, fault scheduling and merging [20]–[22], and fault simulation [23]. Fig. 1 represents the structure of the TPG algorithms , the basic part of ATPG systems. Accordingly, the methods are on the top of the pyramid and they have the highest impact on the efficiency of the TPG algorithms. Next, we have a set of approaches that define the top-level strategies for the basic processes. Finally, we have techniques and heuristics that improve the worst case and average-case performances, respectively. Some of these techniques and heuristics may be approach-oriented or have a specific application with different approaches.

Manuscript received September 10, 2001; revised March 19, 2002. This work was supported by the Japan Society for the Promotion of Science under Grant P99747. This paper was recommended by Associate Editor R. Aitken.

E. Gizdarski is with Synopsys Inc, Mountain View, CA 94043 USA (e-mail: emilg@synopsys.com).

H. Fujiwara is with the Nara Institute of Science and Technology, Ikoma, Nara 630-0101, Japan (e-mail: fujiwara@is.aist-nara.ac.jp).

Digital Object Identifier 10.1109/TCAD.2002.804387

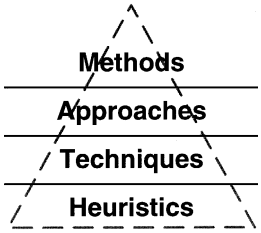


Fig. 1. The structure of TPG algorithms.

The premier SAT-based algorithms [1], [10], [11] translate the TPG problem into a characteristic formula that represents both the *logical* and *structural* constraints for the possible solutions. The formula is usually written in a conjunctive normal form (CNF) where one sum is called a clause. Clauses with one, two, three, or more variables are called unary, binary, ternary, and $k$nary clauses, respectively.

Since a test pattern for a fault is *an input vector* that sensitizes the fault under consideration and propagates the fault effect to a primary output or an observable point, a test pattern is found iff both the fault and a propagation path to an observable point are sensitized and all unjustified lines are justified. If one of these conditions cannot be satisfied, the fault is proved as undetectable. This definition considerably increases the efficiency of the SAT-based TPG algorithms. For example, the premier SAT-based TPG algorithms [1], [10], [11] consider that a test pattern is found when the characteristic CNF formula is satisfied, i.e., all clauses in the formula evaluate to 1. This approach potentially increases the complexity of the TPG problem. The recent SAT-based TPG algorithms [13], [24] also check for an empty J-frontier instead of whether all clauses in the CNF formula are satisfied. In [13], it was shown that SAT-based model also allows an efficient implementation of a forward justification approach typical for the premier structural TPG algorithms [3], [4], [19]. In this way, the SAT-based TPG algorithms demonstrate an ability to incorporate all structural techniques and heuristics. As a result, the main difference between the structural and SAT-based TPG algorithms is in the implication process. The advantages of the SAT-based TPG algorithms are: 1) more elegant and unified model of the TPG problem as well as 2) potentially faster and more precise implication process based on efficient learning techniques [13], [25], [26]. The disadvantage of SAT-based TPG algorithms is higher memory usage because the implication graph in fact duplicates the circuit netlist.

Currently, the practical ATPG systems are structural and the implication process is performed directly on the netlist in order to keep the memory usage as low as possible. In the era of the 32-computers, this was the main restriction because the existing ATPG technology requires processing the whole integrated circuit, having a couple of million gates, at a time. Also, the practical ATPG requires an efficient method for processing tristate logic, incompletely specified blocks (X-sources), and thousands of busses repeatedly, controlling and observing through embedded memories, getting adequate fault coverage, and compressing many faults in a single test pattern while satisfying bus contention and many other restrictions [27], [28]. Clearly, all these requirements increase the complexity of the TPG problem known as NP-complete [29] and the complexity of the TPG model. In this sense, we may consider that a single stuck-at combinational TPG is a simplified model of the practical TPG problem having much lower complexity than the practical TPG problem itself and some related problems like redundancy identification and equivalence checking. Therefore, a failure of a combinational TPG algorithm to achieve complete fault coverage in a reasonable amount of time for relatively small benchmark circuits having a restricted set of primitives indicates that the robustness of this TPG algorithm is not enough. ATPG is involved in many processes where the requirements for the completeness as well as quality of test patterns defined by the maximum number of specified bits and/or detected faults per pattern could be critical. Therefore, ATPG systems have to be able to achieve the required level on completeness in the most efficient way. In this sense, the robustness of ATPG systems will become an important parameter in the future. Then the questions are: 1) whether the SAT method is able to gives an adequate model and efficient solution for the practical ATPG and 2) whether the gap between the existing test technology and test quality requirements will motivate an implementation of a supplementary SAT-based engine in the existing ATPG systems able to achieve higher robustness when this is required. We believe that this work is a step in the right direction.

The rest of the paper is organized as follows. In Section II, a system overview of SPIRIT is provided. In Sections III–V, we discuss implication, propagation, and justification and present techniques and heuristics to improve the efficiency of these processes. Section VI provides experimental results and Section VII concludes the paper.

II. System Overview

Clearly, the performance and robustness for the TPG algorithms are two contradicting parameters; therefore, it is important to find a set of approaches, techniques, and heuristics able to guarantee both high performance and robustness of the TPG algorithms. For example, switching between two approaches for propagation: (1) single path-oriented propagation and (2) propagation and justification for each path segment was very successful in [8]. In [12], it was shown that these two approaches are orthogonal. Also, switching between the basic TPG approaches, whole-circuit and single-cone processing, can be effective to achieve both the high robustness and performance of the TPG algorithms. Obviously, the whole-circuit processing approach has a higher degree of freedom for application of efficient techniques and heuristics for reducing the number of test patterns and improving the performance of TPG algorithm and fault simulation. This approach is also more efficient for easy-to-prove redundant faults because the single-cone processing approach usually needs more than one test session to prove that a fault is undetectable with respect to each primary output where the fault effect can be observed. However, the single-cone processing approach is able to reduce the size of the TPG problem and allows more efficient application of many TPG techniques. As a result, this approach is able to achieve higher robustness than the whole circuit processing approach. Taking this into account, we consider that a two-phase TPG algorithm switching different approaches, techniques, and heuristics will be able to improve

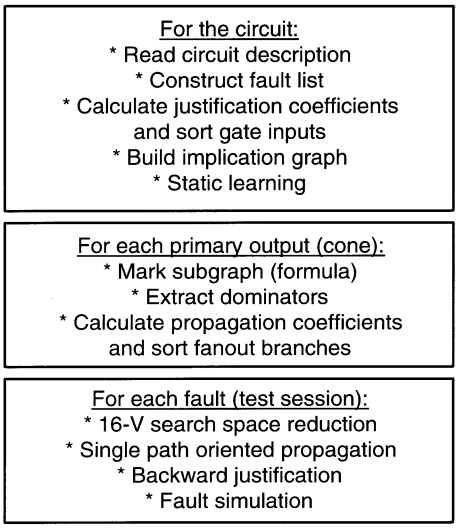


Fig. 2. SPIRIT flowchart.

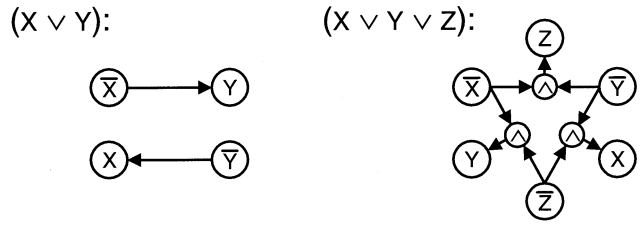


Fig. 3. Implications for binary and ternary clauses.

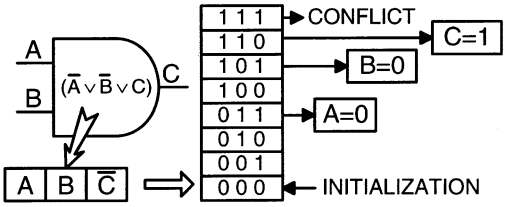


Fig. 4. Representation of direct ∧-implications.

the basic parameters: performance, robustness, and the number of test patterns. In this case, the first phase can be based on the whole circuit processing approach as well as adaptive switching between different approaches for propagation and justification. This phase is oriented to generate test patterns for the most of detectable faults as well as to prove the easy-to-prove redundant faults while keeping performance high and reducing the number of test patterns. The second phase is based on the single-cone processing and single path-oriented propagation. This phase is oriented to increase the robustness of TPG algorithm, i.e., generate complete test set and prove all redundant faults in a reasonable amount of time. In this way, the application of orthogonal strategies (approaches and heuristics) as well as efficient techniques is important to improve the basic parameters of ATPG systems.

In this work, we will focus on the robustness of the TPG algorithms and present some efficient techniques and heuristics for the second phase of the TPG algorithms based on the following approaches: single-cone processing, single path-oriented propagation, and backward justification. We will examine the effectiveness of the proposed techniques and heuristics as well as try to identify which of them are essential to achieve complete test coverage for the processed benchmark circuits. For simplicity, we reduce the number of the implemented techniques but include some techniques that look promising for further research on the robustness of TPG algorithms.

The flow chart of SPIRIT is shown in Fig. 2. In contrast to the premier SAT-based TPG algorithms [1], [10], [11], SPIRIT builds an implication graph and performs static learning once for the whole circuit since these are prominent and time-consuming steps. Using single path-oriented propagation, we avoid extracting the specific structural constraints for propagation of each fault [1], [10], [11]. Using single-cone processing, we apply a "divide-and-conquer" strategy and keep the size of the TPG problem as small as possible.

III. Implication Process

The first step in satisfying a CNF formula is to construct an implication graph. More formally, each variable X is represented by two nodes X and $\overline{X}$. Each binary clause $(X \vee Y)$ is represented by two implications $(\overline{X} \rightarrow Y)$ and $(\overline{Y} \rightarrow X)$. This is shown in Fig. 3(a). In this case, the implication graph represents only the binary clauses and the formula can be easily manipulated since a binding procedure requires only a partial traversal of the implication graph and checking the k-nary clauses [1].

In [12] and [13], an efficient data structure representing all clauses of the CNF formula has been presented. The resultant implication graph is called *complete* and contains two types of nodes. While the first type of nodes represents the variables, the second type of nodes, called ∧-*nodes* here, symbolize an conjunction operation or simply *a direct* ∧-*implication*. In the complete implication graph, each ternary clause is uniquely represented by three direct ∧-implications; see Fig. 3(b). The advantage of this approach is that it allows more than one value assignment to be performed simultaneously using bit-parallelism. The disadvantage of this approach is that it requires dedicated transformation of the k-nary clauses into ternary in order to be included into the complete implication graph.

A. New Data Structure for the Complete Implication Graph

Fig. 4 depicts the proposed data structure of the complete implication graph for two-input AND gate. To represent a k-nary clause, this data structure has 2^k∧-nodes organized as a one-dimensional array and a k-bit key dynamically calculated by the binding procedure. Each bit of the k-bit key corresponds to one variable in the k-nary clause. A bit is set to 1, if the corresponding variable is specified and the k-nary clause is still unsatisfied. A conflict occurs when all the variables in a k-nary clause are specified and the clause still evaluates to 0. In the proposed data structure, the following conventions are used: 1) we represent a gate instead of a k-nary clause. In this case, a modification of this structure is necessary to be able to represent gates having more than one k-nary clause, for example, XOR gates and tristate buffers. 2) The less significant bit(s) of the k-bit key corresponds to the gate output. In this way, we easily identify the unjustified lines and the type of ∧-implications – forward or backward. 3) We need one extra bit in the k-bit key to represent

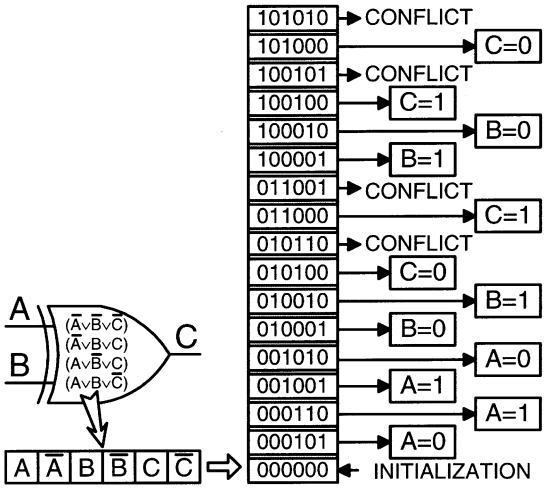


Fig. 5. Representation of direct $\wedge$-implications for two-input XOR gate.

that a gate is justified. This is the most significant bit of the k-bit key. In this way, the k-bit key becomes a negative integer when the gate is justified. Fig. 5 shows an example for two-input XOR gate. Since XOR gate has four ternary clauses and each variable is represented as itself and its complement in the CNF formula therefore the k-bit key has six bits to be able to represent all possible assignments for the variables involved. Accordingly, if variables A and B are set to 1 and 0, then k-bit key is $\langle 100\,100 \rangle$ and variable C must be set to 1. Also, if variables A and C are set to 0 and 1, then k-bit key is $\langle 010\,010 \rangle$ and variable B must be set to 1.

Moreover, the proposed data structure can be viewed as a virtual truth table that does not distinguished between input and output signals such that each signal can be in logic value {0,1, X and Z}. In this way, this data structure can effectively represent complex tristate logic primitives defined by incompletely specified logic functions so that all direct implications and $\wedge$-implications can be easily derived. Fig. 6 depicts an example of this. Fig. 6(a) and (b) shows a truth table and corresponding k-nary clauses for a primitive having four signals. Next, Fig. 6(c) shows for each k-nary clause the number implications and $\wedge$-implications found using simulation and the proposed data structure where the extra implications derived by the proposed data structure are shown in brackets. In this case, we consider that an $\wedge$-implication can be found by simulation only when $k-1$ signals of the primitive are specified, i.e., a forward (or backward) $\wedge$-implication can be found if only the output signal (or only one input signal) is unspecified. This example shows that the precision of implication process can be considerably improved using the proposed data structure for the complete implication graph.

In few words, the advantages of this data structure are that it increases the precision of implication process and allows a unified representation of all primitives and implications. First, all direct implications and $\wedge$-implications are included into the complete implication graph so that they can be easily found without extra calculations during the branch and bound search. Next, all indirect $\wedge$-implications derived during static learning can be also included into the complete implication graph in order to increase the precision of the implication process [25]. In this case, each primitive has its own virtual truth table because the virtual truth tables of the primitives of one type maybe different, i.e, the memory usage for this data structure may increase significantly.

The disadvantage of the proposed data structure is that it can be used for relatively small primitives, i.e., the number of signals of each primitive k is restricted. For example, when a virtual truth table is built for each type of primitives, i.e., the indirect $\wedge$-implications are not included into the complete implication graph, then the memory required is 2^{2k} bytes per type (for the virtual truth table) and $5k+2$ bytes per primitive when $k \leq 7$. A reduction of the size of the virtual truth tables can be achieved using appropriate hash functions. In this case, the virtual truth table includes a restricted set of $\wedge$-nodes, for example, all combinations of specified signals plus all combinations of one unspecified signal. In this way, the virtual truth table of a complex primitive having k signals will be $(k+1)2^k$ bytes.

B. Static and Dynamic Learning

Deriving new relations between the signals is known as "learning," and it is a basic technique for many SAT-based algorithms. The learning itself can be performed either as a preprocessing step, called "static learning," or during the search process, called "dynamic learning." In fact, if all implications are derived at each level of branch and bound search then each instance can be solved without backtracking. Since the deriving of all implications at each level of branch and bound search could be costly, the deriving of as many implications as possible and keeping the complexity of learning as low as possible is an important problem. The premier learning procedures [4], [6] derive only indirect implications, while the recent learning procedures derive new clauses [26] and indirect $\wedge$-implications [25]. The new relations (indirect implications, $\wedge$-implications, and clauses) added into a CNF formula that are always valid are called static, while the relations validated at a certain level of the branch and bound search are called dynamic. Although it is a good idea to perform both learning and search processes simultaneously [26], for simplicity, we suppose here that all static relations between the signals are derived by static learning during preprocessing. This approach helps us to clarify both the complexity and precision of the existing learning techniques as well as find more an effective approach to avoid some negative effects of learning during test generation.

The precision of static learning in [1] and [6] strongly depends on the order of value assignments since some new relations between the signals can be found if certain other relations have already been included into the formula. To avoid this dependency, we assume an iterative computation of the indirect implications [11], i.e., the iterative static learning procedure performs both 0 and 1 value assignments through the variables until one full iteration produces no new implications.

C. Contradiction (Learning Rule 1)

In [6], static learning is based on the contrapositive law, $(X \rightarrow Y) \Leftrightarrow (\overline{Y} \rightarrow \overline{X})$, called *learning rule 1* here. Clearly, the 2CNF portion of a formula (only the binary clauses) fulfills the contrapositive law. This is not true when the k-nary clauses are also included. For example, it is possible that a

A	B	C	D
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	Z
1	0	0	X
1	0	1	X
1	1	0	1
1	1	1	Z

(a) truth table

A	B	C	D
0	0	X	1
0	1	0	0
1	1	0	1
X	1	1	Z

(b) k-nary clauses

backward implications	backward $\wedge$-implications	forward implications	forward $\wedge$-implications
0 (0)	4 (+2)	0 (0)	2 (+1)
0 (+3)	3 (+6)	0 (0)	1 (0)
0 (0)	3 (+4)	0 (0)	1 (0)
0 (+2)	4 (+2)	0 (0)	2 (+1)

(c) implications derived using simulation and the proposed data structure for each k-nary clause

Fig. 6. Representation of complex primitive.

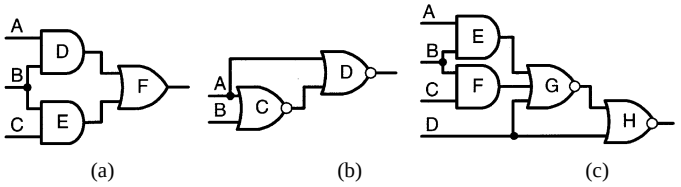


Fig. 7. Network examples 1–3.

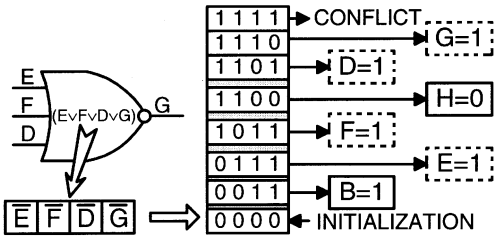


Fig. 8. Representation of indirect $\wedge$-implications.

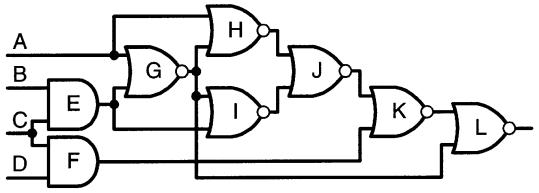


Fig. 9. Network example 4 [30].

value assignment sets $k-1$ variables in a k-nary clause and the clause is still unsatisfied. Then a direct $\wedge$-implication is performed and the last variable of the clause is set to a value so that the clause is satisfied. For example, value assignment $B=0$ for the circuit in Fig. 7(a) sets variables D and E to 0 and clause $(D \vee E \vee \overline{F})$ is still unsatisfied. Next, the binding procedure performs forward $\wedge$-implication and sets the last variable of this clause F to 0. Thus, backward indirect implication $(F=1 \rightarrow B=1)$ is found by rule 1 (contradiction). Also, value assignment $D=1$ for the circuit in Fig. 7(b) sets variables A and C to 0, and clause $(A \vee B \vee C)$ is still unsatisfied. Next, the binding procedure performs a backward $\wedge$-implication and sets the last variable B to 1. Thus, forward indirect implication $(B=0 \rightarrow D=0)$ is found by rule 1.

D. Deriving Indirect $\wedge$-Implications (Learning Rule 2+)

In [30], some indirect implications are derived as an intersection of the implications for satisfying an unjustified line, called *learning rule 2* here. Let us consider how the static learning procedure based on rule 2 finds an indirect implication $(H=1 \rightarrow B=1)$ for the circuit shown in Fig. 7(c). First, the iterative learning procedure calculates a transitive set for each value assignment consisting of all direct and indirect implications derived by the transitive and contrapositive laws. For example, since value assignment $H=1$ sets variables D and G to 0, then to satisfy k-nary clause $(D \vee E \vee F \vee G)$ either variable E or F must be set to 1. However, each one of these value assignments implies that variable B must be set to 1. Since $B=1$ is an intersection of the transitive sets of value assignments $E=1$ and $F=1$, $B=1$ is a necessary assignment for satisfying k-nary clause $(D \vee E \vee F \vee G)$. Thus, an indirect implication $(H=1 \rightarrow B=1)$ is found. Clearly, this indirect implication cannot be found by learning rule 1.

Example 1: Let us consider how an indirect implication $(H=1 \rightarrow B=1)$ for the circuit in Fig. 7(c) can be found by deriving indirect $\wedge$-implications during static learning, called *learning rule 2+*. First, assignment $B=0$ sets variables E and F to 0 and clause $(E \vee F \vee D \vee G)$ corresponding to gate G is still unsatisfied. To take into account this new relation, the learning procedure based on rule 2+ adds an indirect $\wedge$-implication $B=1$ to $\wedge$-node $G\langle 0011 \rangle$; see Fig. 8. Next, assignment $H=1$ sets variables D and G to 0 and clause $(E \vee F \vee D \vee G)$ corresponding to gate G is still unsatisfied. Since the k-bit key of gate G is equal to $\langle 0011 \rangle$, then all implications of $\wedge$-node $G\langle 0011 \rangle$ are valid, i.e., $B=1$ is a necessary assignment.

Clearly, learning rules 2 and 2+ are equivalent with respect to of the number of indirect implications derived during static learning, while learning rule 2+ has lower complexity than rule 2 because there is no need to calculate transitive sets and an intersection of these sets. In addition, the indirect $\wedge$-implications derived during static learning can be used to find some dynamic implications during branch and bound search and dynamic learning [25].

E. Recursive Learning (Learning Rule 3.N)

The first complete learning algorithm, *recursive learning* called *learning rule 3.N* here, is introduced in [16]. If the level of recursion N is not restricted, then all implications can be found. For example, indirect implication $(C=0 \rightarrow L=0)$ for the circuit shown in Fig. 9 cannot be found by rules 1 and 2, while this is possible by rule 3, first level recursive learning. During iterative value assignments through the variables,

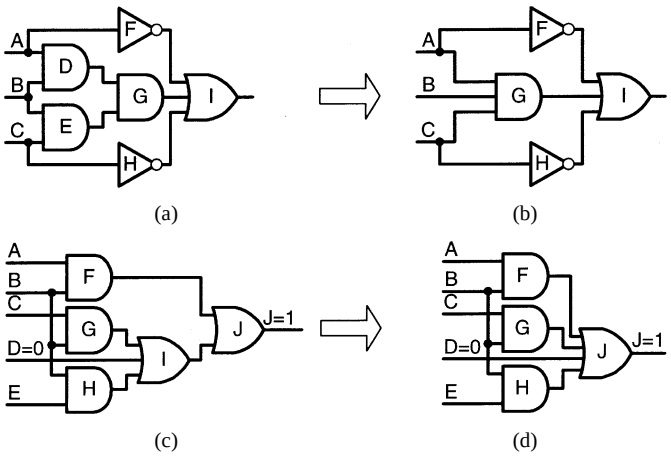


Fig. 10. Network examples 5 and 6.

the static learning procedure first finds indirect implication ($E = 0 \rightarrow H = 0$) by rule 1, since value assignment $H = 1$ sets variable E to 1. Next, value assignment $C = 0$ sets variables E, F, and H to 0, and clause $(A \vee G \vee H)$ is still unsatisfied. To satisfy this clause, either variable A or G must be set to 1, but both these assignments set variable L to 0. Therefore, $L = 0$ is a necessary assignment and indirect implications ($C = 0 \rightarrow L = 0$) is found. These indirect implications cannot be found by rule 2 because neither value assignment $A = 1$ nor $G = 1$ *a priori* implies $L = 0$.

F. Super Gate Extraction (Auxiliary Learning Rule A1)

Next we show how the super gate extraction, called *auxiliary learning rule A1* here, and rule 2+ improve the implication process. A super gate of gate X can be found by performing all direct backward implications of value assignment $X = A$ where A is a noncontrolling value of the gate output X [21].

Example 2: Let us show how super gate extraction improves static learning. For the circuit shown in Fig. 10(a), indirect implication ($I = 0 \rightarrow B = 0$) can be found by rule 3.1 (first level recursive learning). In the transformed circuit shown in Fig. 10(b), gates D, E, and G are replaced by their super gate (three-input AND gate). In this case, ($I = 0 \rightarrow B = 0$) is a direct implication.

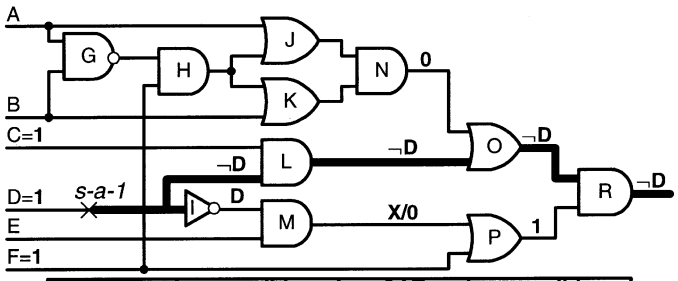
Example 3: Let us show how super gate extraction and deriving $\wedge$-implications improve branch and bound search and dynamic learning. After static learning based on rule 2+, two indirect $\wedge$-implications for gate I are found for the circuit shown in Fig. 10(c). After value assignment $D = 0$, these indirect $\wedge$-implications validate dynamic implications ($I = 1 \rightarrow B = 1$) and ($B = 0 \rightarrow I = 0$). Using implication ($I = 1 \rightarrow B = 1$), dynamic implication ($J = 1 \rightarrow B = 1$) can be found by dynamic learning based on rule 3.1 (first level recursive learning); otherwise, dynamic learning based on rule 3.2 (second level recursive learning) is necessary. If super gate extraction is applied before static learning, then dynamic implication ($J = 1 \rightarrow B = 1$) can be found without dynamic learning. More precisely, in the transformed circuit shown in Fig. 10(d), gates I and J form super gate J and the indirect $\wedge$-implications of this super gate derived during static learning validate dynamic implication ($J = 1 \rightarrow B = 1$) after assignment $D = 0$ without dynamic learning.

G. Deriving k-nary Clauses (Learning Rule 4)

This learning technique is introduced in [26] and has become a basic attribute of the most successful SAT-solvers [31]. More formally, this technique utilizes an analysis to identify a minimum set of value assignments involved in each conflict. If the number of value assignments involved in the conflict is less than a certain number, then a new clause is added into the formula. Clearly, indirect implications and $\wedge$-implications for the existing k-nary clauses (gates) cannot represent all relations between the signals in the circuit. For example, if clause $(B \vee D \vee \overline{J})$ is derived and included into the formula during static learning for the circuit shown in Fig. 10(c), then after assignment $D = 0$ implication ($J = 1 \rightarrow B = 1$) can be found without dynamic learning during branch and bound search. This result cannot be achieved by static learning based on rules 1, 2+, 3.N without transformation of the circuit by auxiliary learning rule 1A. In fact, the complexity of static learning based on rule 4 does not seem to be so high, especially when static learning is restricted to a set of variables, for example, the fanout stems having many fanout branches. In this case, the proposed data structure for the complete implication graph may facilitate such approach. A disadvantage of this learning technique is that it is difficult to identify the type of indirect $\wedge$-implications, forward or backward, derived as a result of adding new clauses. As we discuss later in Section V-B, it is important to distinguish the type of implications in order to improve the efficiency of branch and bound search during justification as well as decrease the number of specified inputs.

IV. Propagation Process

Propagation is a specific ATPG process that is not valid for other SAT-based applications. In general, the propagation is based on the assumption that for each detectable fault at least one path exists that propagates the fault effect from the fault location to a primary output of the circuit. If such path does not exist, then the fault under consideration is undetectable or redundant. Although there are different approaches for propagation, we address here only the single path-oriented propagation (SPOP) [8], [12]. Starting from the fault location forward to the primary output for the current test session, SPOP sensitizes path-segment by path-segment where a *path-segment* is defined as a subpath starting at the fault location or a fanout branch and ending at a fanout stem or primary output. For the SPOP approach, the fanout stems are decision points and next path-segment is selected by initial sorting of the fanout branches using propagation coefficients (see Fig. 2). In this way, the SPOP approach is an alternative to the D-frontier for the structural TPG algorithms and extracting structural constraints for the SAT-based TPG algorithms. The main advantage of the SPOP approach is that the easiest path for fault effect propagation according to the selected criteria is sensitized first. In contrast, the alternative approaches can block the easiest propagation path by justification of the current objectives (path-segment). The disadvantage of the SPOP approach is that during propagation the number of the unjustified lines may increase considerably and it is possible that a sensitized propagation path cannon be justified. Such a path is called *unjustifiable*. To reduce the number



SAT variables	possible assignments	SAT variables	possible assignments
A	{0,1}	I	{D}
B	{0,1}	J	{0,1}
C	{0,1}	K	{0,1}
D	{1}	L	{0,¬D}
D'	{¬D}	M	{0,D}
E	{0,1}	N	{0,1}
F	{0,1}	O	{0,1,¬D}
G	{0,1}	P	{0,1,D}
H	{0,1}	R	{D,¬D}

Fig. 11. Network example 7.

of the sensitized unjustifiable propagation paths we use static learning [6], static 16-V search space reduction, and dynamic learning by rule 3.1 (first level recursive learning) on the unjustified lines. On the other hand, the propagation process can be sped up by avoiding nonsolution areas and deriving as many implications as possible at each level of the decision tree. To do so, we apply the X-path check [3], unique sensitization [4] based on structural dominators [5], [6] and dynamic learning by rule 3.1 [7].

A. Static 16-V Search Space Reduction (SSR)

SSR is a preprocessing step for each test session. SSR starts from the fault location forward to the primary output and calculates the possible value assignments in the set of {0,1, D, $\overline{D}$} for each variable in the formula. SSR also takes into account the number of the possible propagation paths and reduces values {0,1} when one propagation path is available. Using directly observable points, some propagation paths can be reduced and even the fault under consideration can be proved as undetectable with respect to the current primary output during SSR. For example, if a propagation path includes a line that is directly observable for another primary output, then the values {D, $\overline{D}$} for the corresponding variable and, respectively, all propagation paths that include this line can be reduced. If the set of the possible value assignments of a variable is empty, then the fault under consideration is proved as undetectable with respect to the current primary output without search. In this way, SSR sets more precise constraints for the search process and also validates some indirect implications for the faulty circuit. The next example illustrates this.

Example 4: Let us consider the stuck-at-1 fault on line D in Fig. 11(a). Fig. 11(b) shows the possible value assignments for each variable after SSR. Let *D'-L-O-R* be the first sensitized propagation path. However, this propagation path is unjustifiable because line N cannot be justified. If R is not a primary output, this may produce a sensitization of many propagation paths until the alternative path *D'-I-M-P-R* is sensitized. In this particular case, backtracking during propagation can be avoided by static learning. For example, if static leaning is performed during preprocessing, then four indirect implications will be found: ($F = 1 \rightarrow J = 1$), ($F = 1 \rightarrow K = 1$), ($N = 0 \rightarrow H = 0$), and ($R = 0 \rightarrow H = 0$). Using these indirect implications, the inconsistency is found by value assignment $O = \overline{D}$ during propagation. After static learning without SSR, variables P and F are set to value $X/1$ and the inconsistency will be found during justification because the indirect implications derived by static learning are valid only for the fault-free circuit.

B. Augmented Unique Sensitization

According to unique sensitization [4], if a gate belongs to all potential propagation paths starting from the fault location, then this gate is called *unique* and must be sensitized. Unique sensitization is based on a representation of the circuit as a directed acyclic graph $G = (V, E)$ where the set of nodes V contains all primary inputs and gates having more than one input. If a direct path exists from A to B, then line B is said to be *reachable* from line A. If no such path exists, then line B is said to be *unreachable* from line A. Also, if a unique gate exists, then all inputs of this gate that are unreachable from the fault location should be set to a noncontrolling value, i.e., the value that propagates the fault effect via the unique gate. In [5]–[7], the unique sensitization technique was improved using structural dominators. For example, if all paths starting from node A include node B, then line B is called a *structural dominator* of A. Let us now consider that a propagation path to line A is sensitized, then gate B becomes a *dynamic unique gate*. Therefore, all inputs of gate B that are unreachable from the fault location must be set to a noncontrolling value. In this way, the fault effect will be propagated via this gate. Clearly, unique sensitization is more efficient for the single-cone processing approach than for the whole circuit processing approach while calculating the structural dominators for each cone instead of the whole circuit makes this technique more costly. Also, we found that many potential conflicts during propagation can be avoided if dynamic learning is restricted to an area near to the primary output. The propagation procedure based on this assumption has two extra steps.

Step 1) This step is performed after the fault under consideration and all unique gates are sensitized. First, the propagation procedure identifies all alternative decisions for the end of the propagation path. Next, some propagation paths near to the primary output are invalidated by assigning value D and $\overline{D}$ to certain lines. Finally, a convergent gate for all valid propagation paths is determined. Let us assume that $A1, \ldots, An$ are the alternative decisions for the end of the propagation path, then some implications can be found by rule 3.1 (first level recursive learning), i.e., as an intersection of the implications produced by the alternative decisions for the end of the propagation path.

Step 2) This step is based on restricted dominators, i.e., the structural dominators in the last path segment starting from a fanout branch and ending at the

primary output. If the next path segment has a restricted dominator, then the propagation procedure finds some implications using dynamic learning by rule 3.1 (first level recursive learning) on the alternative decisions for sensitization of the last path segment starting from the restricted dominator and ending at the primary output. More formally, the propagation procedure sensitizes this path segment by propagating both value D and $\overline{D}$ to the primary output. Dynamic implications are derived as an intersection of the implications produced by both alternative decisions propagates value D or $\overline{D}$ to the primary output.

The implemented augmented unique sensitization procedure also includes the X-path check for earlier identification of nonsolution area during propagation. More formally, X-path checks, whether at least one propagation path to the primary output exists such that the current value for all signals on this path, are consistent with the target value for its sensitization.

V. Justification Process

Justification is performed after a propagation path to the primary output is sensitized and has to justify all unjustified lines.

A. Backward Justification

Unjustified lines are the decision points for justification and the implemented justification procedure uses justification coefficients calculated during the circuit preprocessing (see Fig. 2) to guide the search process. The justification coefficients take into account the structure of the circuit and measure the relative difficulty for justification of each line. By sorting the inputs of each logic gate using the justification coefficients, we suppose that each unjustified line can be easily justified by assigning a controlling value to the first unspecified input of the corresponding gate. During propagation all unjustified lines are included in a list called *J-frontier*. During justification the J-frontier is dynamically updated by adding new unjustified lines. These are some useful heuristics for selection of next unjustified line for justification from the J-frontier.

H1: *Depth-first search* strategy gives a higher priority to the unjustified lines more recently included in the J-frontier.

H2: *First-difficult/First-easy* are two orthogonal heuristics giving a higher priority to the unjustified lines more difficult/easier for justification according to the justification coefficients.

Heuristic H1 is basic for the backward justification procedure. It is implemented by adding the newest unjustified line at the end of the J-frontier when unjustified lines are processed starting from the end to the beginning of the J-frontier. Since more that one unjustified line can be included into J-frontier as a result of one decision, then the implication process indirectly impacts the selection of the next unjustified line for justification. To be more precise here, we give some details for the implemented implication procedure. It is based on the depth-first search strategy and applies implications for each node in the following order: first forward implications, backward implications,

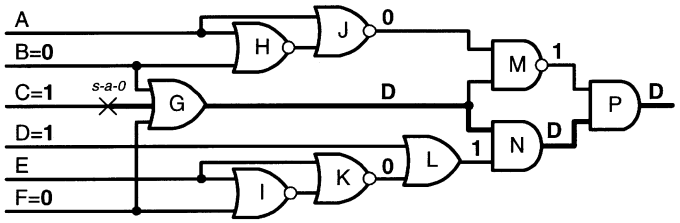


Fig. 12. Network example 8.

and indirect implications, next, all $\wedge$-implications for the current node and its structural successors, i.e., all fanout branches corresponding to this node, and finally, if the current node and/or its successors are unjustified lines, then they are included in to the J-frontier. Actually, this is a dynamic reordering of the variables based on a depth-first search strategy. We found that this strategy is much more efficient than the static order typical for the premier SAT-based TPG algorithms [1], [10], [11]. Heuristic H2 is used for an initial ordering of the J-frontier just before justification. For example, first, the justification is applied using first-difficult strategy. Next, if the specified backtrack limit is exceeded, then the justification is performed with the reverse ordered J-frontier. If both these orthogonal strategies cannot justify or prove that the sensitized propagation path is unjustifiable, then backward justification is performed using dynamic learning using rule 3.1. (first level recursive learning) on the unjustified lines in the J-frontier.

B. Duality of Learning

The decision tree reduction is important to improve the efficiency of search process. For example, the unjustified lines in the conflict-free area, called headlines [4], and inactive clauses are not useful to resolve conflicts and they increase the size of the decision tree. If the headlines are justified at the end of search process and the inactive clauses are deleted, then the size of the decision tree can be reduced. Also, some unjustified lines are justified by implication if other unjustified lines are justified when the reverse case is not valid. Clearly, if these relations between the unjustified lines are identified, then the size of the decision tree can be further reduced. The next example shows that finding good heuristics for justification becomes even more difficult after static learning when some indirect implications are also included into the implication graph.

Example 5: Let us consider three cases for justification of the stuck-at-0 fault on line C in Fig. 12.

(Case A): If static learning is applied during preprocessing, then four indirect implications are found: forward implications ($B = 0 \rightarrow J = 0$) and ($F = 0 \rightarrow K = 0$) and backward implications ($M = 1 \rightarrow J = 0$) and ($N = 0 \rightarrow K = 0$). After value assignment $C' = D$, the fault effect is propagated to the primary output and the value of signals is shown in Fig. 12. As a result, four primary inputs are specified and two unjustified lines, J and K, are included in the J-frontier. During justification, variables A and E are set to 1 in order to justify lines J and K, i.e., in this case all primary inputs are specified.

(Case B): Let us consider the same example without static learning. In this case, variable K will be unspecified after prop-

agation since in case A variable K was set to 0 by forward indirect implication ($F = 0 \rightarrow K = 0$). As result, the J-frontier includes lines J and L. To justify these lines, it is enough for variables A and D to be set to 1. Therefore, without static learning, primary input E is unspecified.

(Case C): This case is equivalent to case A during propagation but the primary inputs A and E are left unspecified because both lines J and K are already justified.

In summary, three solutions for justification are possible: (case A) all variable are specified, (case B) variable E is unspecified, and (case C) variables A and E are unspecified. If the circuit shown in Fig. 12 is a part of large circuit, each solution will produce a different number of unjustified lines that have to be justified. This phenomenon is called *duality of learning* here. Obviously, static learning is important to decrease the number of backtracks. However, static learning may produce some spare unjustified lines that have to be justified. On the other hand, static learning can also be used for identification of the relations between the unjustified lines, for example, in case C.

Example 6: Let us consider the circuit shown in Fig. 7(b). Without static learning after assignment $B = 0$, the ternary clause ($A \vee B \vee C$) is still unsatisfied. To satisfy this clause either variable A or variable C should be set to 1 but both these value assignments set variable D to 0, therefore $D = 0$ is a necessary assignment. If this new dynamic forward implication derived by rule 3.1 is performed, then a new spare unjustified line D will be included in the J-frontier. In fact, line D is justified if line B is justified.

Taking into account these examples, to avoid including spare unjustified lines in the J-frontier *(overspecification)* all forward static indirect implications and $\wedge$-implications as well as all forward dynamic implications should be ignored. In our justification procedure, we avoid overspecification by removing all forward indirect implications and $\wedge$-implications after static learning and restricting dynamic learning to the unjustified lines in the J-frontier. The efficiency of this heuristic was experimentally checked in [32].

C. Avoiding Critical Area

This heuristic was inspired by the work in [29], [33], and [34]. Clearly, the justification is the critical phase of test generation. For example, in [29], the proof of NP-completeness of the TPG problem is based on a polynomial transformation of the justification into the 3CNF-SAT problem. Hereafter, we suppose a similar transformation and represent the justification by a dynamically updated formula called *reduced formula* where inactive clauses in the formula are deleted. In this case, the reduced formula includes a part of the CNF formula corresponding to a circuit involved in the justification process because, as we discussed earlier, justifying and satisfying the CNF formula are not equivalent. In [33], Larrabee showed that for each clause-to-variable ratio less than 4.2, the percentage of satisfiable randomly generated 3CNF formulas increases, and for each clause-to-variable ratio greater than 4.2, the percentage of satisfiable formulas decreases as a function of number of variables. Also, around this ratio the 3CNF formulas need much more CPU time to be satisfied or to be proved as unsatisfiable. We call this area *critical*. In [34], a relation between the cut-width property of a circuit and the worst case complexity for test generation is discussed. Let us associate the unjustified lines and the primary inputs involved in the justification process with the clauses and the variables of the reduced formula, respectively. Clearly, each value assignment during search process changes the clause-to-variable ratio and the cut-width properties of the reduced formula. Until this point, we used these properties by choosing the SPOP approach for propagation. Let us compare the two alternative approaches for justification: 1) justification of the whole propagation path, i.e., the SPOP approach and 2) propagation and justification for each path segment with respect to the critical instances, hard to prove redundant faults. We may consider the SPOP approach is more effective because: 1) the application of the orthogonal strategies of heuristic H2 is more restricted for the second approach and 2) the reduced formula can fall into a critical area somewhere between the fault location and the primary output. In this case, the second approach for justification needs more time to prove the formulas as unjustifiable within the critical area. In [32], we experimentally showed that many redundant faults can be identified during propagation, i.e., without justification. However, SPOP may sensitize one or more propagation paths for redundant faults that are difficult to be proved as unjustifiable because the formula falls into a critical area when the primary output is reached. To avoid the critical area in these cases, the justification procedure selects up to six variables and proves that the sensitized propagation path is unjustifiable for all possible value assignments of the selected variables. To improve the cut-width properties, the justification procedure selects variables corresponding to the fanout stems having a maximum number of fanout branches. As a result, many or all possible value assignments for the selected variables are inconsistent. The remaining cases need fewer backtracks because of the improved cut-width properties of the formula.

VI. Experimental Results

The proposed techniques and heuristics were implemented in SPIRIT [32], [35] and we ran experiments on a 1-GHz Pentium-III PC.

First, we assessed the efficiency of the proposed data structure for the complete implication graph by performance of static learning as well as the number of indirect implications and $\wedge$-implications derived during static learning. The experimental results for the ISCAS'85 [36] and a full scan version of the ISCAS'89 [37] and ITC'99 [38] benchmark circuits are given in Table I. Column 2 gives the number of variables (#primary inputs + #gates having two or more inputs). Columns 3-8 give the number of constant-value assignments, direct and indirect implications and $\wedge$-implications before and after static learning by rules 1, 2 and 2+. The contribution of rule 1 and 2+ was 62 378 401 indirect implications as well as 255 458 indirect implications and 16 070 145 $\wedge$-implications respectively. Columns 12 and 13 present the CPU time in seconds for static learning by rule 1 and 2+, respectively. To estimate the impact of static learning on the propagation process, we calculated the

TABLE I
STATIC LEARNING RESULTS

Circuit	Var	Const. Ass.		Implications			∧ impl.	Sensitized unjustifiable paths for redundant faults			CPU time, sec.	
		w/o	rule1	w/o	rule1	rule2	rule2+	w/o	rule1	rule2+	rule1	rule2+
1	2	3	4	5	6	7	8	9	10	11	12	13
C432	156	0	0	1226	+162	+4	126	3	2	0	0.01	0.01
C499	203	0	0	3246	+168	+0	8	0	0	0	0.01	0.01
C880	354	0	0	3696	+220	+0	371	-	-	-	0.01	0.01
C1355	515	0	0	19582	+2392	+0	24	0	0	0	0.01	0.01
C1908	474	0	0	14153	+4173	+0	862	2	0	0	0.01	0.01
C2670	909	3	8	17213	+3531	+0	1745	47	10	10	0.01	0.02
C3540	1006	1	1	78041	+14101	+0	13293	42	0	0	0.06	0.09
C5315	1591	1	1	34413	+13783	+550	8355	0	0	0	0.03	0.05
C6288	2416	17	17	20193	+8309	+0	0	0	0	0	0.02	0.02
C7552	2309	2	4	81590	+59228	+32	7709	28	0	0	0.10	0.12
S5378	1218	7	7	43566	+14510	+0	4860	0	0	0	0.05	0.05
S9234	2274	2	2	122970	+57982	+32	8437	233	0	0	0.19	0.20
S13207	3273	8	8	336780	+147340	+18	23024	16	16	16	0.84	0.90
S15850	4059	6	6	188613	+99363	+304	16914	0	0	0	0.42	0.45
S35932	13967	0	0	2669601	+267303	+0	0	0	0	0	10.59	10.77
S38417	10373	6	6	265599	+52189	+0	7679	8	0	0	0.51	0.53
S38584	12912	143	151	4233098	+49798	+576	75586	128	0	0	14.28	14.59
B14s	4401	2	2	684954	+496038	+776	222446	1055	80	80	1.19	1.65
B15s	8329	27	27	4067520	+2089280	+28368	1391199	2594	3	3	13.02	17.45
B17s	23008	84	84	12092795	+8306969	+37446	3885330	9414	1458	1458	45.81	58.98
B18s	64993	334	334	35275090	+47126536	+167690	8656215	39162	7004	3186	141.89	174.95
B20s	8711	8	8	1591188	+889700	+5630	506254	12836	112	85	2.41	3.50
B21s	9066	6	6	1434569	+1013097	+1678	436844	14884	239	239	2.96	4.18
B22s	13929	9	9	2373873	+1662229	+12354	802864	6028	1057	1027	4.14	5.85
Total:	190446	666	681	65653569	+62378401	+255458	16070145	86480	9981	6104	238.57	294.4

number of the sensitized unjustifiable propagation paths for redundant faults. Clearly, for redundant faults, each sensitized propagation path to the primary output cannot be justified, i.e., it is unjustifiable. The reason that such a path may be sensitized is the lack of implications, i.e., if all implications at each level of propagation are found, then propagation path to the primary output cannot be sensitized. Columns 9–11 present the number of the sensitized unjustifiable propagation paths for redundant faults. This experiment showed that many faults could be proved as redundant during propagation, i.e., without justification when static learning is performed during preprocessing. The number of sensitized unjustifiable propagation paths can be further reduced using dynamic learning by rule 3.1 during propagation. However, this approach is more costly because we should apply dynamic learning during propagation for each fault and test session. In contrast, dynamic learning during justification is applied only when the current sensitized propagation path cannot be justified or proven as unjustifiable without dynamic learning.

Next, we ran two experiments for test generation. For the first experiment, we used the ISCAS'85 benchmark circuits and a full scan version of the ISCAS'89 benchmark circuits as well as a single-phase version of SPIRIT based on the following three approaches: the single-cone processing, single path-oriented propagation, and backward justification. The maximum number of the sensitized unjustifiable propagation paths during propagation and backtracks during justification was set at 15. Likewise, the maximum number of value assignments per test session was set at 1000. In this case, static learning based on learning rule 1 (only for the ISCAS'85 benchmark circuits) and the 16-V search space reduction were enough to achieve complete fault coverage without fault simulation for these benchmark circuits. This experiment showed that the benchmark sets that inspired the research on test generation for the last 15 years are very easy when proper approaches and techniques are applied. A comparison with the published TPG algorithms able to achieve complete fault coverage without fault simulation for these benchmarks is given in Table II. The columns 2–5 present the basic characteristics of each circuit, the number of detectable and redundant faults as well as the average and maximum cone size estimated by the number of variables (#primary inputs + #gates). Columns 6-9 present the CPU time in seconds for test generation of TEGUS [11] (IBM RS/6000 320), TIP [13] (Digital Alpha 4100 5/533), ATOM [9] (200-MHz Pentuim-Pro PC), and SPIRIT. These results made SPIRIT competitive to the best-published combinational TPG algorithms. The last two columns present the average and maximum numbers of the specified inputs of the test sets generated by SPIRIT. These results showed that the proposed TPG algorithm also gives an efficient solution for overspecification. Without special effort, SPIRIT was able to achieve competitive results for the maximum number of the specified primary inputs as the problem-oriented TPG algorithm reported in [39].

For the second experiment, we used a full scan version of the ITC'99 benchmark circuits.Table III provides the basic characteristics of these benchmark circuits. Columns 2–8 give the number of gates, primary inputs and outputs, detectable and redundant faults, as well as the average and maximum cone size

TABLE II
EXPERIMENTAL RESULTS FOR TPG WITHOUT FAULT SIMULATION FOR THE ISCAS'85 AND '89 BENCHMARKS

Circuit	#Faults		Cone size		CPU time. s				#specified inputs	
	Detected	Redundant	Ave	Max	TEGUS[11]	TIP[13]	ATOM[9]	SPIRIT	Ave	Max
1	2	3	4	5	6	7	8	9	10	11
C432	520	4	125	204	3	0.07	0.3	0.05	12.1	26
C499	750	8	151	190	11	0.17	1.1	0.07	33.8	41
C880	942	0	102	187	3	0.07	0.3	0.06	9.7	23
C1355	1566	8	382	462	19	0.82	12.7	0.36	35.9	41
C1908	1870	9	533	784	14	1.67	2.8	0.39	20.1	31
C2670	2630	117	307	1077	22	1.57	3.9	0.33	15.4	57
C3540	3291	137	582	1832	68	6.55	9.9	0.99	13.4	30
C5315	5291	59	272	1106	42	5.32	6.7	0.72	13.3	47
C6288	7710	34	1327	3526	360	39.44	86.3	13.78	24.7	32
C7552	7419	131	547	1317	86	13.94	23.6	2.20	31.2	100
S5378	4563	40	135	571	9	2.80	2.6	0.38	7.9	36
S9234	6475	452	355	1238	48	19.21	13.8	1.26	11.7	43
S13207	9664	151	189	2004	56	33.31	20.8	2.47	7.7	92
S15850	11336	389	625	1750	125	28.33	23.9	3.65	8.2	40
S35932	35110	3984	78	134	61	238.55	74.5	14.61	4.8	9
S38417	31015	165	197	641	193	175.10	46.6	10.51	13.1	85
S38584	34797	1506	93	1081	98	341.08	40.9	10.10	6.1	54
Total:	164949	7194	285	3526	1218	908	370.7	61.93	10.7	100

TABLE III
EXPERIMENTAL RESULTS OF SPIRIT FOR THE ITC'99 BENCHMARKS

Circuit	#Gates	#Inputs	#Outputs	#Faults		Cone size		With fault sim.		Without fault sim.	
				Detect	Red	Ave	Max	Aborted	Time. s	Aborted	Time. s
1	2	3	4	5	6	7	8	9	10	11	12
B14s	4124	277	299	12537	274	486	2519	0	14	0	24
B15s	7844	485	519	23020	508	1166	4046	0	157	0	336
B17s	21556	1452	1512	64108	1356	1207	4022	0	525	0	987
B18s	61636	3356	3343	184751	3787	1483	8120	0	2779	0	13838
B20s	8189	522	512	24852	486	688	3071	0	35	0	50
B21s	8544	522	512	26084	496	753	3070	0	38	0	53
B22s	13162	767	757	39488	776	739	3083	0	91	0	111
Total:	125055	7381	7454	374840	7683	1185	8120	0	3639	0	15399

estimated by the number of variables (#primary inputs + #gates). In this experiment, we used a two-phase version of SPIRIT where the implemented techniques were carefully selected in order to minimize the impact of the added techniques on the performance of the first phase. The selected techniques for the first phase were the augmented unique sensitization, X-path check, and dynamic learning by rule 3.1 on the J-frontier during justification. For the first phase, the maximum number of sensitized unjustifiable paths during propagation and backtracks during justification was set at 3. Likewise, the maximum number of value assignments per test session was set at 10 000. For the second phase, these limits were set 100 times higher and the heuristics for avoiding the critical area and dynamic learning by rule 3.1 on the J-frontier during propagation were applied to increase the robustness of SPIRIT. The experimental results of SPIRIT with and without fault simulation are provided in columns 9-12.

In this work, we examined many other TPG techniques, but they were ineffective for these benchmark circuits. Some of these techniques were dynamic headlines [4] and dependency-directed backtracking [26], [40]. In many cases, SPIRIT was able to achieve the best performance without static learning because a high dependency between signals increases the CPU time for static learning and degrades the performance of the implication process. The experimental results showed that, in general, static learning increases the robustness of the TPG algorithms especially with respect to the redundant faults but also may decrease the average-case performance for test generation. Table IV presents an impact of the presented techniques on the efficiency of SPIRIT. Also, to assess the negative impact of the techniques added in the first phase of test generation, we repeated the first experiment for the ISCAS'85 and ISCAS'89 benchmark circuits with the new version of SPIRIT. The total CPU time was 69.39 s, i.e., the performance of SPIRIT was degraded by 12%.

VII. CONCLUSION

In this work, we proposed an efficient data structure for the complete implication graph, made a classification of the

TABLE IV
IMPACT OF EACH TECHNIQUE ON THE EFFICIENCY OF SPIRIT

	SPIRIT	Pass 1			Pass 2 (total)		
		Detected	Aborted	Time. s	Detected	Aborted	Time. s
1	All techniques	374836	61	3525	374840	0	3639
2	All techniques only detectable faults	374836	4	2593	374840	0	2607
3	All techniques only redundant faults	0	57	1324	0	0	1427
4	Without fault simulation	374294	603	13957	374840	0	15399
5	Without static learning	374838	424	3703	374840	2	3874
6	Without duality of learning	374837	60	3600	374840	0	3738
7	Static learning based on rule 2+	374836	35	3606	374840	0	3716
8	Unique sensitization without steps 1 and 2	374835	98	4716	374840	24	41986
9	Unique sensitization without X -path check	374836	133	3926	374840	24	10238
10	Backward justification without heuristics H1 and H 2	374835	62	3536	374840	2	3673
11	Backward justification without avoiding critical area	374836	61	3525	374840	14	4074

learning techniques, and assessed an impact of low complexity static learning on the robustness of TPG algoritthms. Also, we examined some not so popular approaches such as the single-cone processing, single path-oriented propagation, and backward justification and showed that they are efficient to increase the robustness of TPG algorithms. Next, we proposed efficient techniques and heuristics for these approaches. In fact, some of the proposed techniques elaborated on the well-known techniques, while other techniques gave new ideas for improving the robustness of the TPG algorithms. As a result, SPIRIT was able to generate complete test sets for a large set of benchmark circuits in a reasonable amount of time. In this way, SPIRIT became the first reported TPG algorithm able to achieve 100% fault efficiency for the ITC'99 benchmark circuits both with and without fault simulation.

ACKNOWLEDGMENT

The authors would like to thank J. Waicukauski for his encouragement and support in implementation and validation of some of the presented techniques in the TetraMAX® ATPG system.

REFERENCES

[1] T. Larrabee, "Test pattern generation using Boolean satisfiability," *IEEE Trans. Computer-Aided Design*, vol. 11, pp. 4–15, Jan. 1992.
[2] C. Chen and S. K. Gupta, "Efficient BIST TPG design and test set compaction via input reduction," *IEEE Trans. Computer-Aided Design*, vol. 17, pp. 692–705, Aug. 1998.
[3] P. Goel, "An implicit enumeration algorithm to generate tests for combinational logic circuits," *IEEE Trans. Comput.*, vol. C-30, pp. 215–222, Mar. 1981.
[4] H. Fujiwara and T. Shimono, "On the acceleration of test generation algorithms," *IEEE Trans. Comput.*, vol. C-32, pp. 1137–1144, Dec. 1983.
[5] T. Kirkland and M. Mercer, "A topological search algorithm for ATPG," *Proc. IEEE/ACM Design Automation Conf.*, pp. 502–508, 1987.
[6] M. Schulz, E. Trischler, and T. Sarfert, "SOCRATES: A highly efficient automatic test pattern generation system," *IEEE Trans. Computer-Aided Design*, vol. 7, pp. 126–137, Jan. 1988.
[7] M. Schulz and E. Auth, "Improved deterministic test pattern generation with application to redundancy identification," *IEEE Trans. Computer-Aided Design*, vol. 8, pp. 811–816, July 1989.
[8] J. Waicukauski, P. Shupe, D. Giramma, and A. Martin, "ATPG for ultra-large structural designs," *Proc. IEEE Int. Test Conf.*, pp. 44–51, 1990.
[9] I. Hamzaoglu and J. H. Patel, "New techniques for deterministic test pattern generation," *J. Electron. Testing: Theory Applicat.*, vol. 15, no. 1/2, pp. 63–73, 1999.
[10] S. Chakradhar, V. D. Agarval, and S. Rothweiler, "A transitive closure algorithm for test generation," *IEEE Trans. Computer-Aided Design*, vol. 12, pp. 1015–1028, July 1993.
[11] P. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Combinational test generation using satisfiability," *IEEE Trans. Computer-Aided Design*, vol. 15, pp. 1167–1176, Sept. 1996.
[12] M. Henftling, H. Wittmann, and K. Antreich, "A single-path-oriented fault-effect propagation in digital circuits considering multiple-path sensitization," *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, pp. 304–309, 1995.
[13] P. Tafertshofer, A. Ganz, and K. Antreich, "Igraine – An implication graph-based engine for fast justification and propagation in the implication graph," *IEEE Trans. Computer-Aided Design*, vol. 19, pp. 907–927, Aug. 2000.
[14] P. Muth, "A nine-value logic model for test generation," *IEEE Trans. Comput.*, vol. C-25, no. 6, pp. 630–636, June 1976.
[15] J. Rajski and H. Cox, "A method to calculate necessary assignments in algorithmic test generation," *Proc. IEEE Int. Test Conf.*, pp. 25–34, 1990.
[16] W. Kunz and D. K. Pradhan, "Recursive learning: A new implication technique for efficient solutions to CAD problems – Test, verification, and optimization," *IEEE Trans. Computer-Aided Design*, vol. 13, pp. 1143–1158, Sept. 1994.
[17] U. Mahlstedt, T. Gruning, C. Ozcan, and W. Daehn, "CONTEST: A fast ATPG tool for very large combinational circuits," *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, pp. 222–225, 1990.
[18] S. Kundu, L. M. Huisman, I. Nair, V. Iyengar, and L. Ready, "A small test generation for large designs," *Proc. IEEE Int. Test Conf.*, pp. 30–40, 1992.
[19] M. Teramoto, "A method for reducing the search space in test pattern generation," *Proc. IEEE Int. Test Conf.*, pp. 429–435, 1993.
[20] T. Inoue, H. Maeda, and H. Fujiwara, "A scheduling problem in test generation," *Proc. IEEE VLSI Test Symp.*, pp. 344–349, 1995.
[21] S. Kajihara, I. Pomeranz, K. Kinoshita, and S. M. Reddy, "Cost-effective generation of minimal test sets for stuck-at faults in combinational logic circuits," *IEEE Trans. Computer-Aided Design*, vol. 13, pp. 1496–1504, Dec. 1995.
[22] K. Tsai, J. Rajski, and M. Marek-Sadowska, "Star test: The theory and its application," *IEEE Trans. Computer-Aided Design*, vol. 19, pp. 1052–1063, Sept. 2000.
[23] J. Waicukauski, E. Eichelberger, D. Forlenza, E. Lindbloom, and T. McCarthy, "Fault simulation for structural VLSI," *VLSI Design*, vol. VI, pp. 20–32, 1985.
[24] L. G. Silva, L. M. Silveira, and J. Marques-Silva, "Algorithms for solving Boolean satisfiability in combinational circuits," *Proc. IEEE DATE-Conf.*, pp. 526–530, 1999.
[25] E. Gizdarski and H. Fujiwara, "A framework for low complexity static learning," *Proc. ACM/IEEE Design Automation Conf.*, pp. 546–549, 2001.
[26] J. Marques-Silva and K. A. Sakallah, "GRASP – A search algorithm for satisfiability," *IEEE Trans. Comput.*, vol. C-48, pp. 506–521, May 1999.
[27] M. Konijnenburg, J. T. van der Linden, and A. J. van de Goor, "Test pattern generation with restrictors," *Proc. IEEE Int. Test Conf.*, pp. 598–605, 1993.

[28] P. Wohl and J. Waicukauski, "Test generation for ultra-large circuits using ATPG constrains and test-pattern templates," *Proc. IEEE Int. Test Conf.*, pp. 13–20, 1996.
[29] H. Fujiwara and S. Toida, "The complexity of fault detection for combinational logic circuits," *IEEE Trans. Comput.*, vol. C-31, no. 6, pp. 555–560, June 1982.
[30] J. Zhao, E. Rudnick, and J. Patel, "Static logic implication with application to redundancy identification," *Proc. IEEE VLSI Test Symp.*, pp. 288–293, 1997.
[31] M. Velev and R. Bryant, "Effective use of Boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors," *Proc. ACM/IEEE Design Automation Conf.*, pp. 226–231, 2001.
[32] E. Gizdarski and H. Fujiwara, "SPIRIT: Satisfiability problem implementation for redundancy identification and test generation," *Proc. IEEE Asian Test Symp.*, pp. 171–178, 2000.
[33] T. Larrabee and Y. Tsuji, "Evidence for a satisfiability threshold for random 3CNF formulas," in *Proc. AAAI Symp. AI NP-Hard Problems*, 1992.
[34] M. Prasad, P. Chong, and K. Keutzer, "Why is ATPG easy?," *Proc. ACM/IEEE Design Automation Conf.*, pp. 22–28, 1999.
[35] E. Gizdarski and H. Fujiwara, "SPIRIT: A highly robust combinational test generation algorithm," *Proc. IEEE VLSI Test Symp.*, pp. 346–351, 2001.
[36] F. Brglez and H. Fujiwara, "A neutral netlist of 10 combinational benchmark circuits and a target translator in Fortran," *Proc. IEEE ISCAS*, pp. 663–698, 1985.
[37] F. Brglez, D. Bryan, and K. Kozminski, "Combinational profiles of sequential benchmark circuits," *Proc. IEEE ISCAS*, pp. 1929–1934, 1989.
[38] F. Corno, M. S. Reorda, and G. Squillero, "RT-level ITC'99 benchmarks and first ATPG results," *IEEE Design Test*, pp. 44–53, July-Sept. 2000.
[39] S. Hellebrand, B. Reeb, S. Steffen, and H.-J. Wunderlich, "Pattern generation for a deterministic BIST scheme," *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, pp. 88–94, 1995.
[40] J. Marques-Silva and K. A. Sakallah, "Dynamic search pruning techniques in path sensitization," *Proc. ACM/IEEE Design Automation Conf.*, pp. 705–711, 1994.

Emil Gizdarski (M'98) received the B.Sc. and M.Sc. degrees from University of Rousse, Bulgaria, in 1986 and 1987, and the Ph.D. degree from Technical University of Sofia, Bulgaria, in 1994.

In 1989, he joined the Department of Computer Systems, University of Rousse, Bulgaria, as an Assistant Professor. In 1993, he was a Visiting Assistant Professor at Brunel University, U.K. From November 1999 to September 2001, he was a Postdoctoral Fellow of the Japan Society for the Promotion of Science at Nara Institute of Science and Technology, Japan. Currently, he is a Research and Development Engineer at Synopsys, Mountain View, CA. His research interests include automatic test pattern generation and build-in self-test.

Hideo Fujiwara (F'89) received the B.E., M.E., and Ph.D. degrees in electronic engineering from Osaka University, Osaka, Japan, in 1969, 1971, and 1974, respectively.

He was with Osaka University from 1974 to 1985, then Meiji University from 1985 to 1993, and joined Nara Institute of Science and Technology in 1993. In 1981, he was a Visiting Research Assistant Professor at the University of Waterloo, ON, Canada, and in 1984, he was a Visiting Associate Professor at McGill University, Canada. Presently, he is a Professor at the Graduate School of Information Science, Nara Institute of Science and Technology, Nara, Japan. His research interests include logic design, digital systems design and test, VLSI CAD and fault tolerant computing, including high-level/logic synthesis for testability, test synthesis, design for testability, built-in self-test, test pattern generation, parallel processing, and computational complexity. He is the author of *Logic Testing and Design for Testability* (MIT Press, 1985).

Dr. Fujiwara received the IECE Young Engineer Award in 1977, the IEEE Computer Society Certificate of Appreciation Award in 1991, 2000, and 2001, the Okawa Prize for Publication in 1994, an IEEE Computer Society Meritorious Service Award in 1996, and an IEEE Computer Society Outstanding Contribution Award in 2001. He is an advisory member of *IEICE Transactions on Information and Systems* and an Editor of IEEE TRANSACTIONS ON COMPUTERS, *Journal of Electronic Testing*, *Journal of Circuits, Systems and Computers*, *Journal of VLSI Design*, and others. He is a a Golden Core member of the IEEE Computer Society, a fellow of the IEICE (the Institute of Electronics, Information and Communication Engineers of Japan), and a member of the Information Processing Society of Japan.