On the Generation of Test Patterns for Multiple Faults

Younès KARKOURI, El Mostapha ABOULHAMID and Eduard CERNY

Dép. d'informatique et de recherche opérationnelle
Université de Montréal, C.P. 6128, Succ. "A"
Montréal, (Québec), H3C-3J7, Canada.

ABSTRACT

This paper presents a new method to generate test patterns for multiple stuck-at faults in combinational circuits. We assume the presence of all multiple faults of all multiplicities and we do not resort to their explicit enumeration: the target fault is a single component of possibly several multiple faults. New line and gate models are introduced to handle multiple fault effect propagation through the circuits. The method tries to generate test conditions that propagate the effect of the target fault to primary outputs. When these conditions are fulfilled, the input vector is a test for the target fault and it is guaranteed that all multiple faults of all multiplicities containing the target fault as component are also detected. The method uses similar techniques to those in the FAN and SOCRATES algorithms to guide the search part of the algorithm and includes several new heuristics to enhance the performance and fault detection capability. Experiments performed on the ISCAS'85 benchmark circuits show that test sets for multiple faults can be generated with high fault coverage and a reasonable increase in cost over test generation for single stuck-at faults.

Keywords: combinational circuits, stuck-at faults, multiple faults, fault analysis, test pattern generation.

1. Introduction

The increased density of logic in digital circuits has a great impact on the complexity of testing. It is well known that the fault detection problem is NP-Complete [13, 21, 23] even if the set of faults is restricted to the traditional single stuck-at fault model (SSF). This model is commonly used because it represents a large class of potential physical failures. It assumes that only one line in the circuit under test is faulty, i.e., having a constant logic value of 0 or 1 independently of the circuit inputs [6, 37] . Therefore, a circuit of n lines can have up to $2n$ different SSFs. Due to the high density of modern circuits, this fault model may no longer be adequate. A manufacturing defect may result in a fault involving more than one line, and the ability of the SSF model to represent this physical defect decreases considerably. Consequently, fault detection methods may have to deal with the multiple stuck-at fault (MSF) model. Under this model, a circuit of n lines can have up to 3^n-1 different MSFs which makes test generation for all of them difficult in most practical cases. Techniques such as fault equivalence and collapsing [29] reduce the number of MSFs to deal with, but this is still too high for explicit fault enumeration.

Numerous fault simulation and test pattern generation methods have been developed for SSFs [2] . Experience from actual circuit testing justifies the adoption of the SSF model: First, circuits are assumed to be frequently tested so that at most one defect occurs at a time. Second, when evaluating the detection capability of MSFs by a test set developed for SSFs, a high coverage of MSFs is achieved for some classes of circuits (depending on their structure and the chosen measure for fault coverage). For a fanout-free circuit, Hayes [17] showed that there exists a complete test set for SSFs that detects all MSFs. Schertz [34] showed that for internal fanout-free circuits a complete test set for SSFs detects all MSFs. Also, Kohavi [27] determined that any complete test set for SSFs in an irredundant two-level circuit detects all MSFs. Unfortunately, these categories of circuits appear rarely in actual designs which may be redundant and may contain a large number of internal reconvergent fanouts (the cause of the NP-Completeness of the fault detection problem). These two characteristics create a phenomenon of masking between faults [11] which limits a test set derived for SSFs from detecting MSFs [38] . A fault f_1 is said to be masked by a fault f_2, if the test vector generated to detect f_1 does not detect the simultaneous occurrence of f_1 and f_2. A cycle of such masking phenomena will result in a situation where all the fault components of a MSF are detectable with a given test set, but the MSF itself is not.

Considering such masking relations, Agarwal and Fung [4] derived a prediction algorithm for the lower bound on MSF coverage using SSF test sets. They showed that the presence of reconvergent fanouts can drastically reduce the coverage of multiple faults on the gates involved in the fanout paths. Hughes and McCluskey [20] reported a simulation study on the coverage of MSFs by different SSF test sets for the 74LS181 4-bit ALU. They found that any complete test set for SSFs detects more

that 99.96 percent of double faults, and many of the test sets detect most of the simulated triple and quadruple faults. These results are analytically explained by Jacob and Biswas [22] . They showed that at least 99.67 percent of all multiple faults in any circuit are detected by a test set for SSFs if the number of observable outputs is greater or equal to three. Kubiak and Fuchs [28] recently reported a multiple-fault simulation method that determines the fault coverage of MSFs with test sets for SSFs. They confirm the results in [20] , i.e., high fault coverage is achieved for multiple faults of small multiplicities in some well-known benchmark circuits.

We believe that the results reported by the above methods [20, 22, 28] are too high because the measure of fault coverage is not adequate. Misleading conclusions can be drawn, and as it will be shown later, the definition of the fault coverage as given in [22] can be independent of the circuit size. Also, a complete test set for SSFs is often incomplete for MSFs [19] due to fault masking [12] , and the analysis of MSFs for test generation is of great interest for achieving high reliability of VLSI circuits. This has contributed to the development of methods to analyze MSF detection in combinational circuits either explicitly under certain assumptions [5, 8, 18] , or implicitly [1, 10, 39] .

Bossen and Hong [5] reduced the list of potential faults by considering stuck-at faults on checkpoints, i.e., fanout branches and primary inputs that do not fan out. They showed that the set of all combinations of these faults is equivalent to the set of all possible MSFs. The approach to test generation establishes global equations for the circuit output functions, while considering all possibilities of faulty conditions. The manipulation of such equations is impractical in the presence of a large number of checkpoint faults or in the presence of redundancy, because masking between faults can occur before reaching primary outputs. This is also true for the method presented in [18] which uses the "G-F" formulas to derive test sets for MSFs. Cha [8] considered prime faults, an equivalent class to the MSFs. He established masking relations between undetected potential faults, relative to a given test set. These relations are used to break masking cycles and add new test vectors to the initial set. This approach is not obvious in the presence of a large number of faults and reconvergent fanouts. Abramovici and Breuer [1] developed a method to identify potential faults not detected by a given test set. The method uses concepts similar to the *D*-algorithm [33] , but analyzes a set of vectors rather than one vector at a time. However, it does not generate additional tests for undetected MSFs and its performance on benchmark circuits was not reported.

Other analysis methods deal with multiple faults implicitly [10, 39] and achieve a high degree of efficiency. An unavoidable pessimism is inherent in these methods because they use a conservative evaluation of the behavior of circuits under the presence of all possible MSFs (even if fault collapsing is performed [39]). Given an initial test set, they determine the set of lines in the circuit that cannot have a particular fault (fault dropping) under the MSF model when the correct output is observed. The results obtained are not invalidated by the presence of undetected or undetectable faults: A fault is dropped only if its effect is not masked by any other fault or combination of faults. The resulting fault

coverage is a lower bound on the MSF coverage that can be achieved in reality: Even though some MSFs involving a particular line are detected, the fault on that line is not dropped unless all MSFs involving this line are detected.

Based on the conservative approach to MSF analysis [39] , we present in this paper a fault-oriented test pattern generation method (TPG) for MSFs in combinational networks. The method is similar to SSF TPG except that error propagation, line justification and implication consider signal ambiguity introduced by the possible presence of all faults in the circuit. The search vehicle uses techniques similar to the FAN [14] and SOCRATES [36] algorithms. Various heuristics as well as a specific fault collapsing procedure are used to reduce signal ambiguity and to find test conditions that detect a target fault. The signal ambiguity is also reduced by fault dropping: when a fault is declared as detected it is dropped from the circuit and its effect is not considered any more.

Our approach to test pattern generation does not perform explicit enumeration of multiple faults; the targeted fault is a single fault component of possibly several MSFs. For each target fault, the algorithm tries to identify test conditions, from a combination of values on primary inputs, which propagate the effect of the target fault to primary outputs regardless the effects of other faults which might be present in the circuit. The generated test vector thus guarantees the detection of the target fault, unless its effect is hidden by a MSF which is really present in the circuit under test (the effect of the target fault would never be observable on primary outputs under any other input vector). Consequently, all MSFs containing this targeted fault component are implicitly detected by the generated vector because no fault masking can occur. Furthermore, since the propagation of fault effects is based on necessary conditions, the approach is conservative. It is thus possible that a test for a fault is not found although it exists, but the algorithm will never claim a fault as detected while in reality it is not. The algorithm also incorporates the efficient fault analysis method proposed in [39] . The TPG method was applied to the ISCAS'85 benchmark circuits [7] , and satisfactory fault coverage was obtained. The experimental results also provided an indication of the cost of deriving good test sets for multiple faults, which is about 10 times the cost of deriving tests for SSFs.

The rest of the paper is organized as follows: Section 2 presents the fault, line and gate models under the MSF model. Section 3 contains an overview of the test pattern generator for MSFs. Section 4 presents the MSF test pattern generator and the different tools and heuristics included to guide the search process. Section 5 reports the experimental results. Finally, Section 6 concludes with notes on possible directions for multiple fault test pattern generation.

2. The Models

Dealing with multiple faults requires a particular care when representing and propagating faulty values in the circuit. The computation of the values is conservative in order not to create optimistic conditions for fault detection that may invalidate the obtained results.

2.1. The Fault Model

We assume that all faults occur on circuit lines only; we refer to a single fault component of a multiple fault as a *fault*. The gates are fault free and may have up to m inputs ($m \geq 1$) for AND, NAND, OR and NOR gates. XOR and XNOR gates have two inputs. A path from line i to a primary output is said *normal* if all lines along this path are normal, i.e., are faultless, but the other inputs to gates along this path may be faulty.

Fault collapsing is first performed to retain only one fault per equivalence class, and it is based on the intuitive fact that a fault effect is easier to observe if it is closer to the primary outputs. Thus, the representative fault is the closest one to the primary outputs. For example, a stuck-at-0 (s-a-0) on an input of an AND gate is removed and placed on the output. We do not consider that all inputs of an AND gate are simultaneously stuck-at-1 (s-a-1), because we assume an equivalent s-a-1 fault at the gate output.

The following faults are retained after collapsing:

- s-a-1 (s-a-0) faults on all inputs of AND/NAND (OR/NOR) gates. The multiple fault consisting of s-a-1 (s-a-0) on all inputs is not considered.
- No fault on the input of an inverter and a buffer gate, and on a fanout stem.
- Both s-a-0 and s-a-1 on primary outputs.
- Both s-a-0 and s-a-1 on each input of XOR and XNOR gates. The multiple fault consisting of s-a-1 or s-a-0 on both inputs is not considered.

Furthermore, there is at least one normal path from each faulty site to a primary output.

In a circuit consisting of n lines, a multiple fault is represented by a tuple with at most n faults and denoted by $f = (s_i^\alpha, s_j^\beta, \dots)$, $i \neq j$, where s_i^α represents the *status of line i:* $\alpha = 0$ for s-a-0, $\alpha = 1$ for s-a-1. A line k not present in the tuple is not faulty in that multiple fault. According to this definition, a multiple fault f in the circuit under test partitions the lines into three disjoint categories, *Hidden Lines*, *Faulty Lines* and *Normal Lines*:

- Line i is *faulty* if $s_i^\alpha \in f$.

- Line j is *normal* if $s_j^\alpha \notin f$ and there is a normal path from j to a primary output.

- Line k is *hidden* if there is no normal path from k to a primary output.

During TPG, we assume that the circuit under test contains one multiple fault consisting of a combination faults that has not yet been dropped (detected). The effective values (real values) on hidden lines are unknown, because they are unobservable due to the multiple fault, and there is no algorithmic way to determine these values (Normal Path Theorem in [1]). We assume that *these lines carry fault free values*. As will be seen in Section 4.3, this assumption does not invalidate the generated tests.

2.2. The Line Model

Various algebras have been used to describe the behavior of the fault free and the faulty circuit using *D* symbols [33] . We represent the fault free value of a line *i* by a *normal value* n_i = 0, 1 or X (unspecified). Each line may carry a value that is different from the normal one due to propagated fault effects. A fault effect is potential (may or may not be present), since all the faults can be potentially present in the circuit.

We represent the fault effect on a line *i* using a *propagation bit* p_i = 0, 1, or X. p_i = 1 if line *i* propagates a potential fault effect different from its normal value. p_i = 0 if no fault effect can propagate to line *i* : Line *i* is carrying its normal value only, or is hidden or is faulty. Initially, all lines have unspecified normal values (n_i = X) and p_i = X if under the current value assignment, the lines driving *i* still have unspecified normal values. In the rest of the paper, the notation n_i / p_i will be used to represent the value assignment to line *i*. For example, a line carrying the value 1/1 has a normal value of 1 and may be propagating a fault effect (which can change its value to 0). Table I summarizes the possible values of a line *i*.

TABLE I
LINE VALUE INTERPRETATION

n_i / p_i	Fault free	Faulty
0 / 0	0	0
0 / 1	0	0 or 1
0 / X	0	X
1 / 0	1	1
1 / 1	1	1 or 0
1 / X	1	X
X / 0	X	X
X / 1	X	X
X / X	X	X

The faulty value of a line is assumed unspecified when its normal value or propagation is equal to X. Primary inputs are directly controlled from the circuit environment, hence no fault effects can propagate to them.

Each line *i* is associated with its status s_i^α which can be viewed as an atomic proposition:

- s_i^α = 0 if the fault s-a-α is not on line *i* (e.g., has been dropped),
- s_i^α = 1 if the fault s-a-α is potentially present on line i.

2.3. The Gate Model

The gate model computes the value on the output of a gate given the status and the values of its inputs. It thus determines the normal value and the propagation of fault effects arising from a combination of fault effects and potential stuck-at faults on its inputs. Since fault effects are assumed independent, the evaluation is conservative and includes the real behavior of the circuit.

Example 1: Fig. 1 shows the computation of the value and the propagation bit for an AND gate for different input situations. In Fig. 1a, given the input values i = 0/0 and j = 1/0, the normal value on the gate output line g is n_g = 0. If line g is normal then it may propagate a fault effect issued from the combination of the fault s_i^1 and the normal value n_j = 1. Hence, g carries the value 0/1.

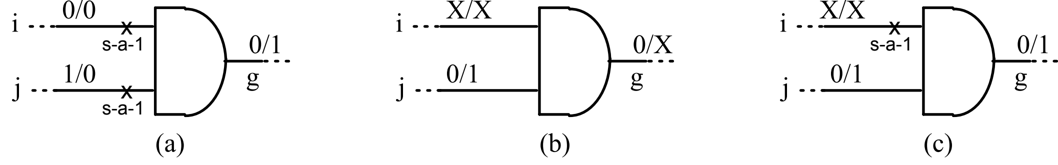


Fig. 1. Fault effect propagation.

In Fig. 1b, we assume that both lines i and j are normal and line j may carry a fault effect (p_j = 1). The propagation bit at the output g remains unspecified because it depends on n_i and p_i. This not the case in Fig. 1c where the fault effect on j may be combined with the fault s_i^1 to propagate a fault effect on g independently of n_i and p_i. Notice that a definite value is assigned to the output only if it is guaranteed regardless the interpretation of X on the inputs.

This conservative computation of propagated values is performed using equations similar to those reported for 2-valued logic in fault analysis [39] . For TPG, these equations are extended to handle 3-valued logic for both the normal values and the propagation bits. Let a = 0,X designate that the variable a can take the value 0 or X (a □ {0, X}), then the equation to determine the propagation bit p_{out} on the output out of a m-input AND gate is the following† (P and Q are predicates; P is true if p_{out} can be equal to 0; Q is true if p_{out} can be equal to 1):

- $P = [\exists i, (n_i{=}0,X \wedge p_i{=}0,X \wedge s_i^1{=}0)] \vee [\forall i, (n_i{=}0,X \wedge p_i{=}0,X)] \vee [n_{out}{=}1 \wedge (\forall i, p_i{=}0,X)]$
- $Q = [(\exists i,\ n_i{=}1,X \vee p_i{=}1,X) \wedge (\forall \mathrm{i},\ n_i{=}0,X \Rightarrow (p_i{=}1,X \vee s_i^1{=}1))] \vee [n_{out}{=}1 \wedge (\exists i, p_i{=}1,X)]$

The value of p_{out} is then assigned as follows:

$$p_{out} = 0 \Leftrightarrow (P \wedge \overline{Q});\ \ p_{out} = 1 \Leftrightarrow (Q \wedge \overline{P});\ \ p_{out} = X \Leftrightarrow (P \wedge Q)$$

For example, no fault effect propagates to the output of the AND gate if there exists a normal input having a normal value of 0 with no possible fault effect; this input disables the propagation of any fault effect to the output. Also, the gate will not propagate a fault effect if all its inputs have a normal value of 0 and none of them is carrying a fault effect. The fault effect on the output of the gate is considered unspecified if it can be equal to 0 or 1 (i.e., P = Q = true). The equations for NAND, OR, NOR, XOR and XNOR gates are in Appendix Ia. For a fanout stem, the propagation bit is broadcast as is to all its branches. For inverters and buffers, the propagation bit is also transmitted as is to the output, because there are no faults on their inputs. As discussed in Section 4.1, the necessary conditions for sensitizing a gate to the effect of a target fault rely on assigning values (0 or 1

† "∀ i" in all equations represents "for each input i of the gate, i = 1, ..., m".

depending on the gate type) to unspecified normal values and/or propagation bits of its inputs that are not reachable from the target fault site.

3. System Overview

We consider that the circuit under test may contain a multiple fault consisting of a combination of faults remaining after collapsing. This multiple fault partitions the lines into hidden, faulty and normal lines (recall that hidden lines are assumed fault free). The goal of TPG is to generate a test set that determines as many lines as possible that cannot be faulty when the fault free response is observed. To do so, the algorithm identifies for each target fault the test conditions which allow to declare the respective line necessarily not carrying that fault. The line may in reality be either hidden or normal.

A flowchart of the TPG system is in Fig. 2. First, a preprocessing phase is performed to analyze the circuit structure, required for incorporating heuristics that guide the search part of the algorithm, including the values of controllability/observability measures [3, 16] . As commonly used in TPG systems, a random test pattern generation (RTPG) may be performed first. A non negligible number of faults are usually dropped during this phase, which accelerates the subsequent deterministic TPG. RTPG is stopped when either a maximum number of vectors has been reached or the last *n* consecutive vectors do not detect any additional fault. This phase is performed at reasonable cost since the multiple fault analyzer can manipulate 32 vectors simultaneously [24] .

A target fault is arbitrarily selected from the list of remaining faults. The TPG gradually determines a set of objectives that are necessary for activating the target fault and for propagating its effect to a primary output, while taking into account the potential effects of the other faults still in the fault list. To meet these objectives in terms of a combination of values on the primary inputs, the TPG uses similar procedure as in FAN [14] and SOCRATES [36] , but using our models (Section 2):

- An *implication procedure* determines as many line values as possible that are uniquely implied. Both local [14] and global implications [36] (learned during the preprocessing phase) are performed.
- A *multiple backtrace procedure* concurrently traces more than one path to satisfy an objective.
- A *unique sensitization procedure* is applied whenever the *S*-frontier (which is similar to the *D*-frontier in SSF TPG) consists of a single gate.
- The backtrace is stopped at *head lines*, since their justification can be done without conflicts even under the MSF model, and is then completed in the final stage of the TPG process.

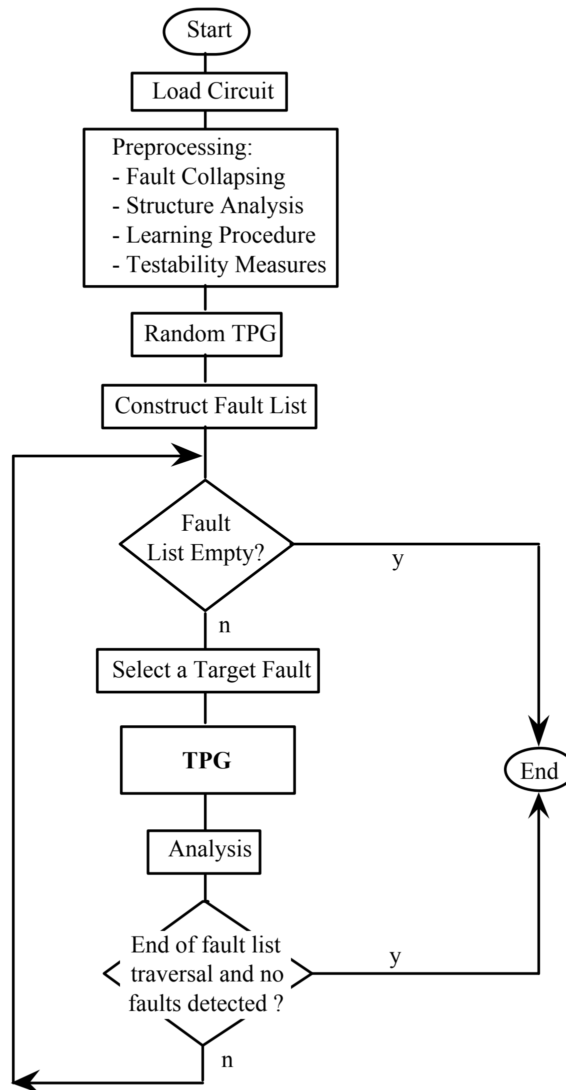


Fig. 2. Flowchart of the test pattern generator.

The test conditions which activate the target fault and propagate its effect to a primary output, are characterized as follows (the justification of these conditions is presented in Section 4.3):

*C*1: The effect of the target fault cannot be masked by any other fault(s), **and**

*C*2: The effect of the target fault was propagated through a normal path, **or**

*C*2': The effect of the target fault was propagated through all paths to primary outputs if no normal path yet exists to a primary output.

Condition *C*1 is maintained during the propagation of the target fault effect due to the conservative evaluation of line values (Section 2). Conditions *C*2 and *C*2' are verified during a backward deduction phase which is engaged each time the target fault effect reaches a primary output. If fulfilled, the target fault is declared as detected and is dropped from the fault list. An alternative equivalent interpretation of the detection conditions is as follows: The effect of the target fault is observable on a primary output through a fault free path (condition *C*2), or is not observable under any input vector

because its site is hidden by another MSF (condition $C2'$). Because of $C1$, the test vector detects all multiple faults containing the target fault since the conservative evaluation guarantees that no fault masking occurs.

Multiple fault analysis is then applied, using the generated test vector with random values assigned to unspecified primary inputs. This step is similar to performing fault simulation after generating a test for a fault under the SSF model, and it allows to drop additional faults. The analysis determines the lines that must be faultless in order to observe fault free response on the output of the circuit. Given an input vector, the fault free circuit is evaluated and the effects of all remaining faults are propagated using the conservative evaluation of line values. Assuming now that the fault free response is observed, a backward deduction procedure identifies and drops faulty conditions (i.e., fault effects and faults) on lines that are not masked by other faults or could be hidden by yet undetected faults. Similarly as in [10] , to reduce the pessimistic behavior, the deduction procedure also performs event analysis between two consecutive vectors to retrace paths that propagated, from primary inputs to the outputs, an event (a 0 to 1 or 1 to 0 transition). If found, such paths must be normal and all faults along them are thus dropped. All equations to propagate fault effects and to deduce the values and the status on a gate inputs are Boolean, and are implemented using bit strings allowing to analyze up to 32 test vectors simultaneously (typically the length of a machine word) [24] . This efficient implementation is about ten times faster than the original single pattern analysis [39] .

Fault dropping permits gradual identification of normal paths (using the backward deduction phase) and reduces the ambiguity caused by faults when the next target is selected for TPG. Fault masking relations are not retained during TPG; therefore, each time the traversal of the target fault list is completed with some of the faults detected, it is reprocessed with the aim to detect faults that were possibly masked when targeted in the previous list traversal.

Since our TPG uses conservative value propagation and bases fault detection on necessary conditions only, the method is not a complete algorithm in that it may not always find a test for a fault at a given step in the TPG, although it may find it later when other faults will have been detected and dropped from the circuit. Given enough time to generate test patterns, the procedure will not find a test only when circular masking between faults exists. But then such a test may not exist at all if all the faults of the masking relation are present. Moreover, the method is also not complete for generating tests for all detectable multiple faults: There are situations where a multiple fault consisting of undetected faults is detectable, but the algorithm will not discover it because it is not explicitly enumerated and the individual faults are not detectable in the presence of the other faults.

4. Multiple Fault TPG

The TPG assumes that the circuit may contain a multiple fault, hence it takes into account all possible values of the status of the lines when generating a test for a given target fault. Consequently, the concepts of SSF TPG must be generalized to accommodate this factor.

4.1. Multiple Fault TPG Concepts

When the effect of the target fault is propagated to a primary output under the SSF model, the circuit lines are divided into two classes: those carrying the fault effect (*D*-drive), and those set to 0 or 1 to propagate the fault effect. With our MSF model, no distinction is made between these two classes, since fault effects may be issued from either the target and/or other faults. We thus introduce the concept of *sensibility* for implementing an equivalent notion to the *D*-drive and for distinguishing the effect of the target fault from all the other fault effects.

Definition 1: A line j is *sensible* to a target fault s_i^{α} if under the current value assignments to lines, line j propagates the effect of s_i^{α} (i.e., $p_j = 1$). The sensibility of line j to s_i^{α} is denoted by the attribute $w_j = 1$. $w_j = 0$ indicates that any fault effect on line j is not due to the target fault.

According to the above definition, $s_i^{\alpha} = 1$ □ $p_j = 1$, but the fault effect on line j could also be due to other faults in addition to s_i^{α}. Sensibility is propagated through the circuit similarly as D propagation for SSF TPG. A line i is sensible to its own fault s_i^{α} whenever $n_i = \overline{\alpha}$. The propagation of sensibility through gates depends on the gate types, the polarity of the input line(s) sensible to the target fault, and the normal values and the propagation bits of the inputs not reachable from the fault site (called *non-sensible* inputs in the rest of the paper). Fig. 3 illustrates three examples of propagation of sensibility through an AND gate, assuming that input i is sensible to the target fault (i.e., $w_i = 1$).

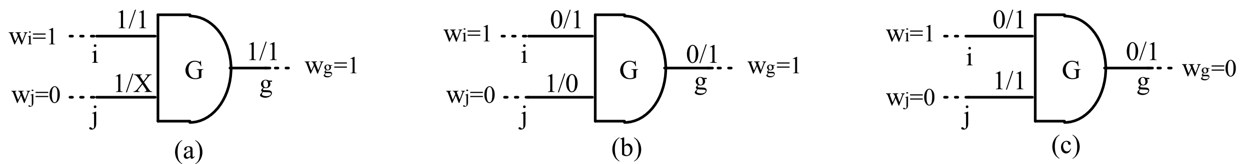


Fig. 3. Sensibility propagation through an AND gate.

In Fig. 3a, the output g is sensible to the target fault ($w_g = 1$) because either line j propagates no fault effect ($p_j = 0$) and the fault effect on g is due to $p_i = 1$ (since $n_j = 1$), or both i and j propagate a fault effect ($p_j = 1$) in which case $p_g = 1$ indicates that the effect of the target fault cannot be distinguished from another effect, but it is certainly present. If the sensible input has a normal value 0, then the gate output is sensible if every non-sensible input j has $n_j = 1$ and $p_j = 0$. In Fig. 3b, $p_g = 1$ is due to the effect of the target fault only, hence $w_g = 1$. On the other hand, if input j may carry a fault

effect ($p_j = 1$) (Fig. 3c), it may mask the effect of the target fault and even if $p_g = 1$ (because of the conservative propagation of fault effects), the gate output is not declared sensible ($w_g = 0$).

Sensibility to the target fault is propagated through the circuit using specific equations for each gate type (Appendix Ib). For example, for an AND gate with output g, the sensibility propagation is:

$$w_g = 1 \Leftrightarrow (\exists i,\ w_i{=}1) \wedge [(\forall i,\ n_i{=}1) \vee (\forall i, (n_i{=}0 \wedge w_i{=}1) \vee (w_i{=}0 \wedge n_i{=}1 \wedge p_i{=}0))]$$

During TPG for SSFs, the propagation of the effect of the target fault consists of selecting one gate from the D-frontier and assigning values to its unspecified non-sensible inputs so that the gate output propagates a D or $\overline{D}$ value. Similarly, in the case of MSFs the *S-frontier* is a set of all gates having one or more inputs sensible to the target fault, and the non-sensible ones having an unspecified normal value and/or propagation bit such that the target fault effect is not masked. For example, the inclusion of an AND gate G in the S-frontier is determined using the following expression:

$$G \,\square\, S\text{-frontier} \,\square$$
$$(\exists i,\ w_i{=}1) \wedge [[(\forall i,\ n_i{=}1,X) \wedge (\exists i,\ n_i{=}X)] \vee [(\forall i, ((n_i{=}0 \wedge w_i{=}1) \vee (w_i{=}0 \wedge n_i{=}1,X \wedge p_i{=}0,X))) \wedge (\exists i,\ n_i{=}X \vee p_i{=}X)]].$$

To illustrate the preceding concepts, consider the circuit in Fig. 1. Assuming that line j in Figures 1b and 1c is sensible to the target fault, the AND gate is in the S-frontier since its output can become sensible if 1/0 is assigned to line i. The required normal value on each non-sensible input j of the gate is always 1 (non-controlling value), and the propagation bit p_j on this input must be 0 whenever the sensible input has the normal value 0, while p_j is X if the sensible input has a normal value of 1. In the latter case, the presence or absence of fault effects on the non-sensible inputs does not affect the sensibility of the gate output to the target fault. Note that a gate is not included in the S-frontier if two of its sensible inputs have different normal value polarities. This is captured in the above inclusion expression of a gate in the S-frontier (see also Appendix Ic).

We summarize in Table II the required values on the non-sensible inputs in order to propagate the target fault effect through different types of gates. For a XOR or a XNOR gate, if the non-sensible input j has both potential s_j^0 and s_j^1, the gate is not included in the S-frontier. The assignment to a non-sensible input j of a XOR or a XNOR gate in Table II assumes that either s_j^0 or s_j^1 was dropped in an earlier iteration of TPG, hence the assigned normal value n_j depends on the remaining fault. If both were dropped, then n_j is chosen depending on the value of the controllability measure of line j (Section 4.4).

TABLE II
REQUIREMENTS FOR THE PROPAGATION THROUGH A GATE IN THE *S*-FRONTIER

Gate types	Sensible input value(s)	Requirements on non-sensible input(s)	Output Assignment
AND (NAND)	1 / 1	1 / X	1 / 1 (0 / 1)

	0 / 1	1 / 0	0 / 1 (1 / 1)
OR (NOR)	0 / 1	0 / X	0 / 1 (1 / 1)
	1 / 1	0 / 0	1 / 1 (0 / 1)
XOR (XNOR)	0 / 1 or 1 / 1	1 / 0 if $s_j^0 = 0$ 0 / 0 if $s_j^1 = 0$	1 / 1 or 0 / 1 0 / 1 or 1 / 1

The backward justification of values assigned to the non-sensible inputs is performed by backward implication procedure in the same manner as in TPG algorithms for SSFs, except that it again takes into account the propagation bits when required to be 0.

4.2. Implication and Unique Sensitization

The assignments which propagate the effect of the target fault through the circuit are justified by a search over possible line values (n_i / p_i) that have a good likelihood in satisfying them. This search explores the space of possible solutions using a branch-and-bound technique [15] . The convergence to a solution is improved using heuristics and techniques, described as implication and unique sensitization, that are similar to those in FAN [14] and SOCRATES [36] .

• *Implication*

The role of the implication procedure is to identify as many line values that are uniquely determined as possible, to keep track of lines sensible to the target fault, to update the *S*-frontier, and to check for the consistency of value assignments. Line values are determined by forward and backward implications in the circuit. Forward implication consists of a simple computation of line values using the gate model defined in Section 2.3. Backward implication determines, when possible, the values on gate inputs that uniquely justify the value on the output of the gate. Both normal value and propagation bit values are determined based on the function and the propagation bit equation of the gate (Appendix Ia). The input values are left unspecified when there is no unique value assignments. Such inputs are later processed in the multiple backtrace procedure (Section 4.4). Fig. 4 shows unique normal value assignments when the propagation bit on the output of an AND gate is not specified.

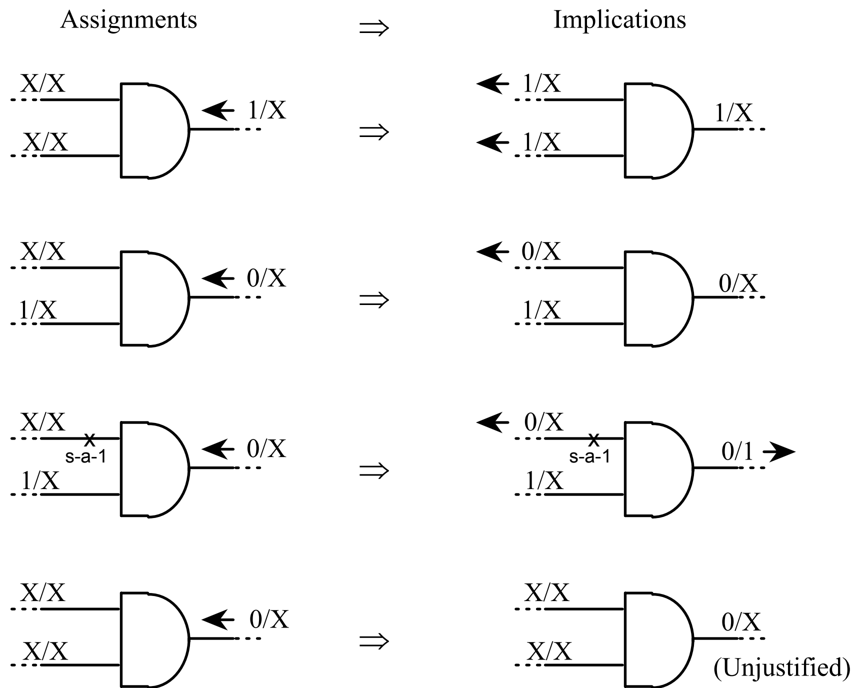


Fig. 4. Uniquely determined input normal values of an AND gate.

When the propagation bit on the output of an AND gate is specified to be 0, the only two situations which are uniquely determined are:

(1) If s-a-1 faults on all its input are still possible, then all the inputs are assigned 0/0. (Recall that these s-a-1 faults cannot occur simultaneously).

(2) If there exists exactly one input i such that $s_i^1 = 0$, then this is the unique input that, when set to 0/0, disables the propagation of any fault effect to the gate output.

Fig. 5 shows unique input assignments when the propagation bit on the output of an AND gate is 0. For other gate types, the unique assignments are easily derived from their propagation bit equations (Appendix Ia).

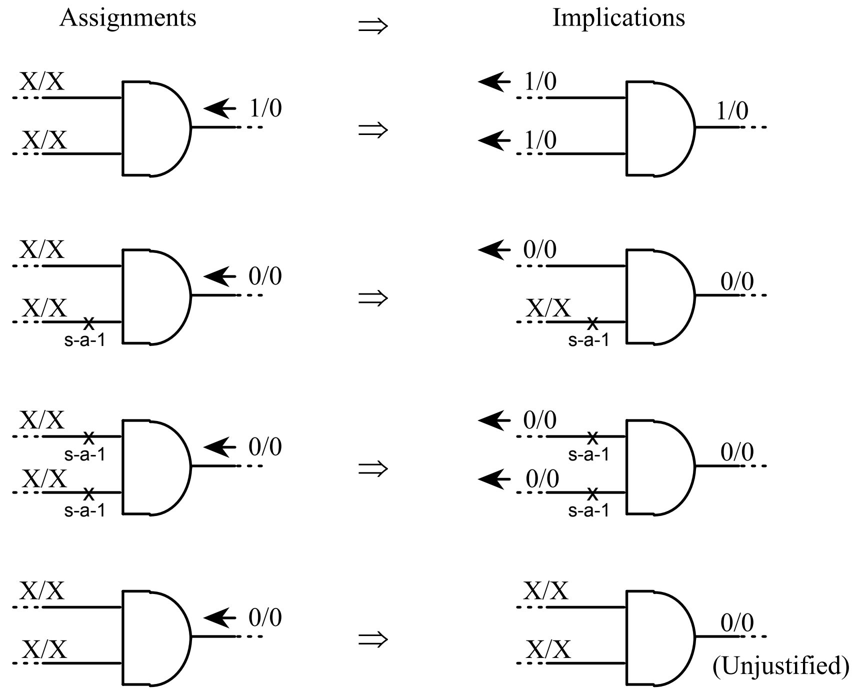
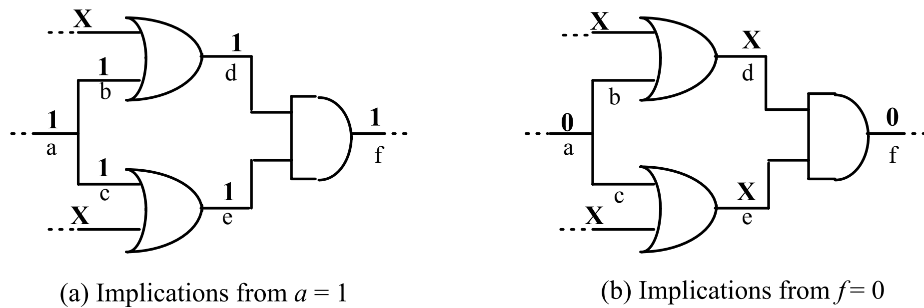


Fig. 5. Uniquely determined input values of an AND gate.

In FAN, the backward implication procedure makes unique assignments locally from the output of each gate. In addition, global implications learned during the preprocessing phase were introduced in SOCRATES [36] The global implications help to reduce the number of backtracks and permit early recognition of conflicts and redundancies. We illustrate this learning procedure using an example. The details can be found in [36] . Fig. 6 shows a small part of a circuit, in which the value 0 has been assigned to *f*. The gate feeding *f* becomes unjustified since either *d* or *e* set to 0 would satisfy *f* = 0. The preprocessing phase uses a learning procedure to determine that (a = 1) □ (f = 1) (Fig. 6a) which also means (f = 0) □ (a = 0) (Fig. 6b).



(a) Implications from a = 1 (b) Implications from f = 0

Fig. 6. Learning global implications [36] .

In our case, only normal values are determined, and thus the learning procedure is the same as in [36] . Propagation bits are not considered since invalid implications may result in the presence of a multiple fault. In the circuit of Fig. 6, to justify the value 0/0 on *f* during the backward implication,

only n_a = 0 is assigned. p_a will be specified by local backward implications depending on the remaining faults on the gate inputs.

Before assigning a value to a line, the implication procedure checks for the consistency of the assignment. According to our line model, a conflict occurs when either a value α is to be assigned to a line i having $n_i = \overline{\alpha}$, or when 0 is to be assigned to p_i which is already equal to 1. In such cases, the implication procedure signals inconsistency and is aborted.

• *Unique Sensitization*

In FAN [14] , when the *D*-frontier consists of a single gate, a unique sensitization procedure is used to find immediately as many unique assignments of line values as possible. These assignments decrease the number of choices and thus the number of backtracks. Fig. 7 shows an example of the application of the unique sensitization procedure using our models, when the *S*-frontier consists of a single gate.

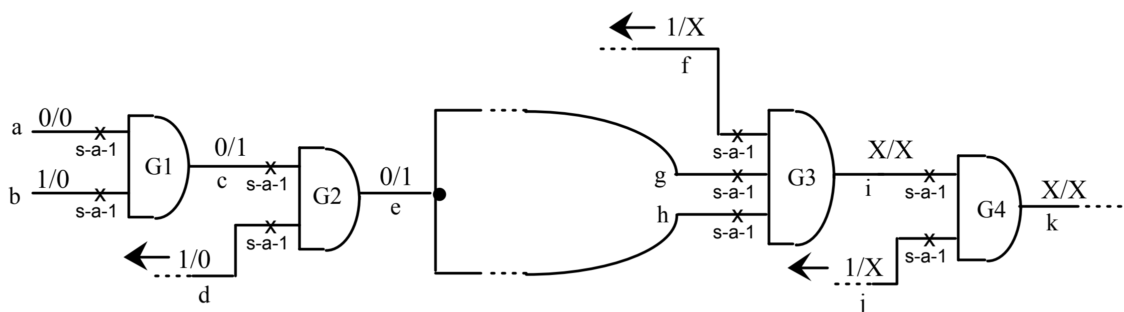


Fig. 7. Application of the unique sensitization.

The target fault is s_a^1 and the input a is assigned the value 0/0. The only gate in the *S*-frontier is $G1$, and every path from $G1$ to primary outputs passes through c-e and i-k. In order to propagate the effect of s_a^1, we have to sensitize paths c-e and i-k. To do so, non-sensible inputs of gates $G1$, $G2$, $G3$ and $G4$ have to be assigned the non-controlling normal value 1 (from Table II). Justifying a propagation bit set to 0 on a line is more difficult due to remaining faults, potentially requiring many more assignments (as illustrated in Fig. 5). Propagation bits on these inputs are specified only if the polarity of all the paths from the fault site to the corresponding gates is the same (i.e., even or odd but not both), in order to propagate the target fault effect (to avoid its masking). Hence, we assign 1/0 to lines b and d since the polarity of the path from line a to $G1$ and $G2$ is even. For gate $G3$, assuming that the polarity of all the paths from a to g and to h can be either even or odd, that is, g and h may be assigned 1/1 or 0/1. Therefore, 1/X is assigned to f and has to be justified backward. The propagation bit on f will be set to 0 only if a subsequent forward implication assigns 0/1 to g and h in order to sensitize $G3$ to the target fault. $p_f = 0$ has to be subsequently justified by backward implication. But, if the forward implication assigns 1/1 to g and to h, there is no need for specifying p_f, because the value 1/X on line f is sufficient to sensitize $G3$ to the target fault. The case for line j is similar. To conclude the example,

the unique sensitization procedure assigns 1/0 to *b* and *d*, and 1/X to *f* and *j* and then would justify them, if possible, by backward implication.

The unique sensitization procedure requires that line dominance be determined during the preprocessing phase [26, 36] . Line *y* dominates line *x* if all directed paths from *x* to the primary outputs pass through *y*. The polarity of these paths is also computed. For each line *x* we retain all gates that dominate *x* and all their inputs which are not reachable from *x*. In Fig. 7, the outputs of the gates *G*1, *G*2, *G*3 and *G*4 dominate line *a*, and the list of retained inputs is ($b_{[E]}\ d_{[E]}\ f_{[-]}\ j_{[-]}$), where the polarity of the paths from *a* to these gates is indicated between brackets ([E]: Even, [O]: Odd, and [-]: Even and odd).

4.3. Fault Detection

The lines of the circuit are hidden, faulty or normal. The goal of our TPG is to generate a test set that determines the lines that cannot be faulty (hidden or not) when the fault free response is observed. The algorithm uses the sensibility concept to propagate the effect of the target fault through the circuit, but cannot declare this fault as detected when a primary output is reached and is sensible (as done under the SSF model when a primary output is assigned a D or $\overline{D}$ and all justifications of line values succeed). The generated test may be invalid in the presence of another fault that masks the effect of the target fault on the sensitized path only. Thus, we must establish additional test conditions under which the target fault can be dropped (i.e., detected) assuming that the fault free response is observed. These conditions are:

*C*1: The target fault is sensitized to a primary output through a normal path and thus the fault cannot be present if the fault free response is observed on this output.

*C*2: No normal path has been identified from the target fault site, but all the paths to primary outputs are sensitized to the target fault. In this case, if the fault free response is observed on these primary outputs, the target fault site is either normal or hidden. Thus in both states, the target fault can be dropped.

To verify these conditions during the propagation of a target fault effect through the circuit, the algorithm must maintain a list of all the paths along which the target fault can propagate (normal paths) or must be propagated (not normal paths). This may result in an inefficient implementation since a list traversal has to be performed at each step. To remedy this inefficiency, the algorithm does not maintain such a list, but instead performs a *backward deduction phase* each time the target fault effect is propagated to a primary output and all justifications are successful. This phase verifies the presence of the test conditions that allow to drop the target fault, and at the same time it drops other faults along sensitized paths for which the test conditions *C*1 and *C*2 are fulfilled. In TPG for SSFs, this corresponds to dropping all activated faults along sensitized paths. Note that this phase is similar to the backward deduction used in fault analysis [39] . It is a linear time algorithm that performs

backward sweeps from primary outputs having specified value toward primary inputs. It assumes that fault free response is observed on the primary outputs and drops fault effects on them by resetting to 0 their propagation bits. Then, for each gate feeding these outputs, it deduces the possible values that are actually carried by their inputs: The deductions may result in dropping fault effects and faults that are not masked and thus cannot be present or their sites are hidden.

A fault effect on a line is dropped by resetting to 0 its propagation bit, i.e., the line carries fault free value only. When the propagation bit on a line is reset to 0, backward deduction continues further into the driving network. Backward deduction is not performed through any gate having propagation bit on its output different from 0. Depending on the type of the gates, the propagation bits and the status of the inputs are deduced using the following deduction lemmas. For an AND/NAND the deduction lemma can be stated as follows:

Lemma 1: If the propagation bit p_g on the output g of an AND/NAND gate is reset to 0, then the following are sufficient conditions for dropping the fault effect and the fault on each input i of the gate:

- Drop the fault effect: $p_i = 0 \square\ p_i = 1 \square [(n_i = 0 \square (\forall j \neq i, n_j = 1 \square p_j = 0)) \Delta (\forall j, n_j = 1)]$
- Drop the fault: $s_i^1 = 0 \square\ n_i = 0 \square (\forall j \neq i, n_j = 1 \square p_j = 0)$

For a fanout stem, the conditions are stated in the following lemma:

Lemma 2: Let s be a fanout stem. Sufficient conditions for resetting the propagation bit p_s to 0 are:

- $\forall$ fanout branches b of s, $p_b = 0$, <u>or</u>
- $\exists$ fanout branch b of s such that $p_b = 0$ and b is normal.

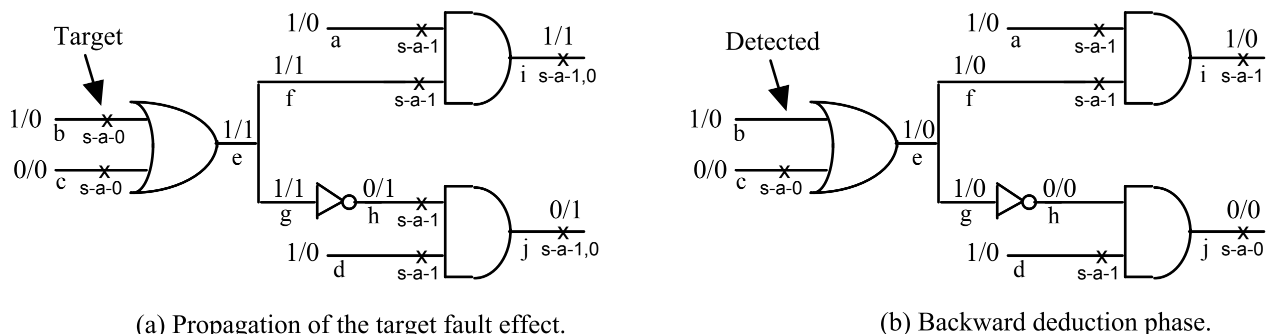
The lemmas for all types of gates are presented in Appendix II and their proofs are similar to those given for the analysis method [25, 39] . For inverters and buffers, if the propagation bit on the output is reset to zero, it is also reset to zero on the input.

A line i is declared normal if $s_i^1 = 0$ and $s_i^0 = 0$ (i.e., both faults were dropped) and there exists a normal path from i to a primary output (i.e., all faults along this path have also been dropped). The backward deduction phase determines such normal paths while backtracing in the circuit (Appendix III).

The following example illustrates some of the possible deductions during the backward phase.

Example 2: Consider the circuit in Fig. 8 containing faults s_a^1, s_b^0, s_c^0, s_d^1, s_f^1, s_h^1, s_i^1, s_i^0, s_j^1 and s_j^0. Any combination of these faults constitutes a multiple fault whenever there is a normal path from each of its components to at least one primary output and excluding simultaneous faults on inputs of the same gate. Let the target fault be s_b^0. The fault is activated (Fig. 8a) by assigning normal value 1 to input b, and $p_b = 0$ since it is a primary input. The assignment $c = 0/0$ propagates the target fault effect to line e. Since there is no normal path from the fanout stem e to a primary output, the effect of the

target fault must be propagated through all its branches. By setting a and d to 1/0, the outputs i and j are reached. At this stage, lines e, f, g, h, i and j are all sensible to s_b^0.



(a) Propagation of the target fault effect.

(b) Backward deduction phase.

Fig. 8. Fault detection example.

The backward deduction is now executed (Fig. 8b). First, it assumes that the fault free response is observed on the primary outputs i and j, i.e., $p_i = p_j = 0$. Faults s_i^0 and s_j^1 are detected and dropped. Backtracing on each of the AND gates feeding i and j, the propagation bits on f and h are reset to 0 according to lemma 1, i.e., these effects will be observable on i and j, unless they are hidden by s_i^1 and s_j^0, respectively. The fault s_h^1 is detected based on the same deductions. The fault effect on stem e is also reset to 0 since it was reset to 0 on all its fanout branches (f and g). Based on Lemma 3 of an OR gate (Appendix II), the fault effect on e is due to s_b^0 only, and since $p_e = 0$ then the fault s_b^0 is detected. The vector $t = (abcd) = (1101)$ is thus a test for the faults s_b^0, s_h^1, s_i^0 and s_j^1. In fact, implicitly, t is a test for all multiple faults containing one or more of these faults as a member. For example, the set of multiple faults containing s_b^0 and detected by t is $F=\{(s_b^0), (s_b^0, s_a^1), (s_b^0, s_f^1), (s_b^0, s_i^1), (s_b^0, s_i^0), (s_b^0, s_d^1), (s_b^0, s_h^1), (s_b^0, s_j^0), (s_b^0, s_j^1), (s_b^0, s_a^1, s_h^1), (s_b^0, s_a^1, s_d^1), (s_b^0, s_f^1, s_d^1), (s_b^0, s_i^0, s_d^1), (s_b^0, s_i^1, s_d^1), (s_b^0, s_j^0, s_a^1), (s_b^0, s_j^1, s_a^1)\}$.

4.4. TPG algorithms

The key concepts introduced in the previous sections are the propagation of sensibility (w_i), the propagation of fault effects (p_i), and the test conditions for fault detection. These concepts and the underlying fault, line and gate models can be incorporated in any TPG algorithm for SSFs to adapt it for multiple faults. In our case, we use a branch-and-bound technique to explore the solution space using a binary decision tree [15] with heuristics techniques to improve the performance. Fig. 9 illustrates the overall TPG procedure, based on a recursive version of FAN [2] . The procedure assumes that the target fault is supplied by the caller procedure. The Boolean variable *Test_Found* is true if the target fault is detected; the test vector is returned when the values on the head lines are justified by the procedure *Justify_Head_Lines()*.

```
procedure TPG( Target_Fault ) : return( SUCCESS, FAILURE );
begin
    if Imply_and_Check() = FAILURE then return( FAILURE );            /* 1 */
    if (error at PO and all bound lines are justified) then
        begin
            Justify_Head_Lines();
            Backward_Deduction( Target_Fault );                        /* 2 */
            if Test_Found then begin
                Analysis() ;                                           /* 3 */
                return( SUCCESS );
            end;
        end;
    if (not error at PO and S-frontier = Ø) then return( FAILURE );
    Add every unjustified bound line to Current_Objectives;
    G :=  Select one gate from the S-frontier;                         /* 4 */
    Fix_Inputs( G );
    (line, value, propagation) = Multiple_Backtrace( Current_Objectives );   /* 5 */
    if (line, value, propagation) is empty then return( FAILURE ); /*no selections */
    Assign (i=line, n_i=value, p_i=propagation);
    if TPG( Target_Fault ) = SUCCESS then return( SUCCESS );
    Assign (i=line, n_i=n̄_i , p_i=X);                                  /* 6 */
    if TPG( Target_Fault ) = SUCCESS then return( SUCCESS );
    Assign (i=line, n_i=X, p_i=X);
    return( FAILURE );
end;
```

Fig. 9. Test pattern generation algorithm.

In the following, we explain the numbered lines in Fig. 9 that represent the newly introduced or extended concepts in our TPG method.

1) *Imply_and_Check():* This procedure is as discussed in Section 4.2. It determines as many line values as possible that are uniquely implied and updates the *S*-frontier. It performs unique sensitization whenever the *S*-frontier consists of a single gate. The procedure fails when a conflict occurs during a forward or a backward implication.

2) *Backward_Deduction(Target_Fault):* This procedure performs a backward sweep from primary outputs that have been assigned a value toward primary inputs. It relies on the lemmas in Section 4.3 to drop fault effects and faults on inputs of gates. The target fault is detected during this phase when conditions *C*1 and *C*2 are fulfilled (Section 4.3).

3) *Analysis():* This is the multiple fault analysis method as in [39] and recently extended to handle up to 32 input vectors in parallel [24] . The analysis is performed when the target fault is detected, in order to detect additional faults. The primary inputs left unspecified by the TPG are assigned random values.

4) The *S*-frontier consists of a list of gates in decreasing order of their observability measure values [3, 16] . This helps to select, at each step of the propagation of the target fault effect, the gate which is the closest to a primary output and whose error propagation is the easiest to observe. This ordering is neglected when the target fault effect is to be propagated first through a normal branch of a fanout stem. In this case, the reachable gate in the *S*-frontier from this normal branch is chosen first. The values are assigned to non-sensible inputs according to Table II (procedure *Fix_Inputs(G)* in Fig. 9). These inputs are added to the *Current_Objectives* to be justified by the multiple backtrace procedure.

5) *Multiple_Backtrace()*: This procedure traces backward multiple paths to satisfy the set of *Current_Objectives*. It is similar to the procedure in FAN, except that an objective consists of a line, a normal value and a propagation bit value. The selection of lines is performed as described in Section 4.2 when a gate output is specified but not its inputs (because of the presence of more than one input choice), and it is guided using testability measure values [3, 16] . The procedure returns the line number of a head line or a fanout stem *i* and values for n_i and p_i that have a good likelihood to satisfy the objectives. This procedure was extended to handle XOR and XNOR gates [36] (Appendix IV).

6) Backtracking: In this recursive version of the algorithm, the decision tree is identical to that of PODEM or FAN. When a value assignment on a line is rejected due to a conflict, the alternative is tried with the propagation bit set to X. It becomes 0 only if required for satisfying a subsequent objective(s).

4.5. A Complete TPG Example

Consider the circuit in Fig. 10. After fault collapsing, all the remaining faults (shown in the figure) are assumed to be simultaneously present in the circuit, which is the initial circuit status. Lines *a*, *e* and *d* are head lines, *b* and *c* are free lines and *f*, *g*, *h*, *i* and *j* are bound lines [14] . The list of faults in an arbitrary order is s_a^1 , s_f^1 , s_g^1 , s_d^1 , s_h^0 , s_i^0 , s_j^1 , s_j^0 , s_b^1 , s_c^1. Target faults are selected in the order of appearance.

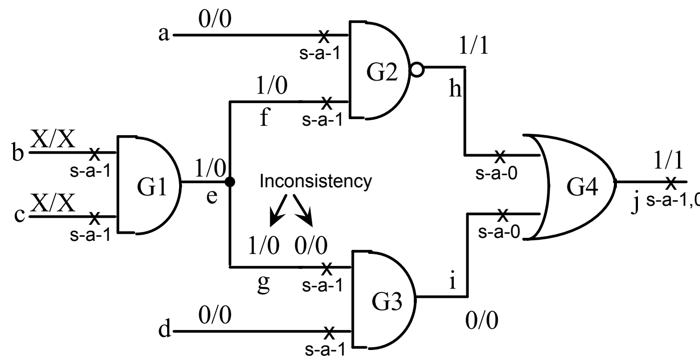
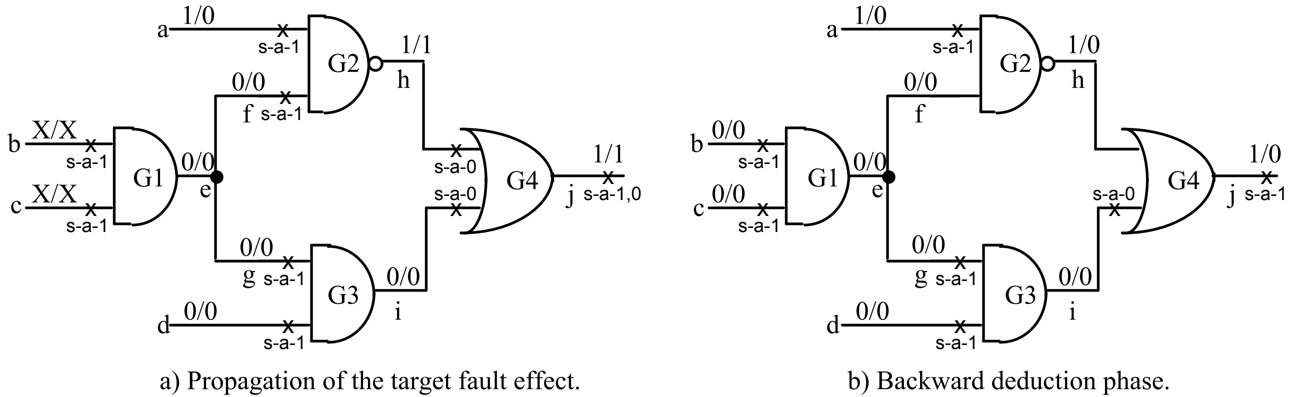


Fig. 10. Aborted Target Fault.

Let s_a^1 be the first target fault. The assignment a = 0/0 activates it and implies w_a = 1, h = 1/X and j = 1 / X. Since lines h and j dominate a, the unique sensitization procedure determines that f = 1/0 and i = 0/0 are necessary to propagate the effect of s_a^1 to the primary output j. f = 1/0 implies e = 1/0, g = 1/0, h = 1/1, w_h = 1, and j = 1/1. i = 1/0 implies d = 0/0 and g = 0/0 (since s_g^1 and s_d^1 are still possible). At this step, the implication procedure stops because an inconsistency occurs on g - it was assigned 1/0 earlier -, as shown in Fig. 10. No backtracking is performed because all assignments were necessary and the target fault is aborted. In fact, s_a^1 would be detected only if s_d^1 is detected because the presence of s_d^1 masks s_a^1 at gate $G4$: if s_d^1 = 0, then d = 0/0 uniquely justifies i = 0/0.

Let s_f^1 be the next target fault. As shown in Fig. 11a, the assignment f = 0/X activates the fault and implies w_f = 1, e = 0/X, g = 0/X, h = 1/X, i = 0/X and j = 1/1. The unique sensitization procedure determines that a = 1/0 and i = 0/0 are necessary to propagate the effect of s_f^1 to primary output j. a = 1/0 implies h = 1/1 and w_h = 1. i = 0/0 implies g = 0/0, d = 0/0, e = 0/0, f = 0/0 and w_j = 1. At this stage, all line justifications succeed and the primary output is sensible. The justification of 0/0 on the head line e results in the unique assignments b = 0/0 and c = 0/0. The backward deduction procedure (Fig. 11b) then drops s_j^0, s_h^0 and s_f^1. The generated vector t = ($abcd$) = (1000) thus detects any MSF containing one of these faults.



a) Propagation of the target fault effect. b) Backward deduction phase.

Fig. 11. TPG example.

Table III summarizes the TPG results for all faults in the circuit of Fig. 10. The columns indicate the selected target fault, the generated test vector, the list of dropped faults, the list of deduced normal lines during backward deduction, and the result of TPG regarding the target fault.

TABLE III
COMPLETE TPG FOR THE CIRCUIT EXAMPLE

Target Fault	Test Vector	Dropped Faults	Normal Lines	TPG Result
s_a^1	□	□	□	Aborted
s_f^1	1000	s_j^0, s_h^0, s_f^1	□	Detected
s_g^1	□	□	□	Redundant
s_d^1	1110	s_j^1, s_d^1	*j, h, f, e*	Detected
s_i^0	1111	s_i^0	*i*	Detected
s_b^1	1010	s_b^1	*b*	Detected
s_c^1	1100	s_c^1	*c*	Detected
s_a^1	0110	s_a^1	*a*	Detected

The faults s_b^1 and s_c^1 when selected, are sensitized to primary output *j* through the path *e-f-h-j* that was declared normal. In the last row of Table III, the fault s_a^1 is selected for the second time, since it was aborted during the first traversal of the fault list. The final test set T = {1000, 1110, 1111, 1010, 1100, 0110} detects all multiple faults in the example except (s_g^1) because its unique component s_g^1 is a redundant fault.

5. Experimental Results

The TPG for multiple faults was implemented in the MainSail™ programming language on a SUN SPARC-Station 2. Several experiments were conducted on the ISCAS'85 benchmark circuits [7] , but before discussing the results we first present the definition of MSF coverage.

5.1. MSF Coverage

The different definitions of multiple fault coverage found in the literature generally depend on the method for detecting MSFs or for predicting their detection. Hence, these measures have to be distinguished to avoid confusion [9] . For example, the coverage measure given in [22] was often used in the past, but it can be misleading. In a circuit of n lines there are $2n$ faults, producing 3^n - 1 possible MSFs. Assume that there are k lines on which the faults have been dropped because all MSFs involving them are detected. Thus, the total number of remaining MSFs is 3^{n-k} - 1. The lower bound C on the MSF coverage, as defined in [22] , is then

$$C \geq \frac{3^n - 3^{n-k}}{3^n} = 1 - \frac{1}{3^k}$$

This lower bound depends on k but not on n (the circuit size)! For example, if we determine that 4 lines are not faulty, the equivalent lower bound on the MSF coverage is 98.7%, independently of the circuit size. This measure is misleading and we do not adopt it since a near 100% coverage would be always achieved for all circuits. Instead, we use a more meaningful definition of MSF

coverage (an unconditional measure) as in [10] : *The ratio of the number of dropped faults to the total number of faults*.

Note, that this coverage is a lower bound for the measures defined in [4, 20, 22, 28] , since it is more pessimistic: Even though some MSFs involving a particular line in the circuit are detected, the fault on that line is not dropped unless all MSFs involving this line can be detected. For example, in the circuit of Fig. 10, the total number of faults is 18, but only 17 faults are declared as detected, leading to 94.4% fault coverage. If we had used the measure from [22], the coverage would have been ≈ 100%!

Table IV presents some statistical data on the benchmark circuits. The table gives the circuit name; the number of primary inputs and outputs, the number of gates, the number of faults remaining after collapsing, the number of redundant single stuck-at faults [32] , the number of learned implications during preprocessing, and the CPU time of the preprocessing phase. In all tables, the CPU time is given in seconds (on a SPARC-Station 2), and the fault coverage in percent (%) as defined above.

TABLE IV
STATISTICS ON BENCHMARK CIRCUITS

Circuit	PIs	POs	Gates	Faults	Redundant	Implic.	CPU Time
c432	36	7	160	346	4	57	1.2
c499	41	32	202	640	8	40	2.8
c880	60	26	383	692	0	85	2.4
c1355	41	32	546	1056	8	208	20.2
c1908	33	25	880	1109	9	498	12.4
c2670	233	140	1193	1839	117	843	11.1
c3540	50	22	1669	2270	137	3920	65.9
c5315	178	123	2307	3738	59	1371	22.9
c6288	32	32	2406	4832	34	619	120.2
c7552	207	108	3512	4950	131	3356	51.0

5.2. MSF Detection by SSF Test Sets

In the first experiment, we determined a lower bound on MSF detection by test sets developed for SSFs. We analyzed two test sets for each circuit: (1) compressed sets [30] and (2) non-compressed sets [31] . The coverage analysis was performed using the multiple fault analysis method of [24] . Table V summarizes the fault coverage and the CPU times. Columns "(1)" and "(2)" identify the two test sets. Due to fault masking that occurs when assuming the presence of all faults, the analysis of a test set was repeated as long as faults could be dropped (the number of repetitions is given in column "Repeat").

TABLE V
MSF COVERAGE BY TEST SETS FOR SSFS

Circuit	Vectors		Repeat		Coverage (%)		CPU Time (sec.)	
	(1)	(2)	(1)	(2)	(1)	(2)	(1)	(2)
c432	44	520	8	2	81.2	99.4	1.7	4.3
c499	60	750	1	1	47.2	91.7	0.4	3.9
c880	30	942	1	2	69.2	98.2	0.6	16.2
c1355	95	1566	2	2	56.4	70.8	2.1	31.1
c1908	142	1863	2	2	85.5	97.2	15.7	58.2
c2670	67	2621	2	1	78.8	88.7	7.6	61.5
c3540	111	227	2	2	67.2	69.6	15.5	32.5
c5315	34	5261	2	1	81.5	93.1	15.7	259.8
c6288	16	49	2	2	56.3	54.2	20.9	23.3
c7552	87	375	2	4	72.5	75.1	47.4	84.9

The MSF coverage is far from acceptable for the majority of the circuits (e.g., c1355, c2670, c3540, c6288 and c7552), especially in the case of compressed test sets. Therefore, MSF detection using test sets developed for SSFs may not achieve a good MSF coverage. In particular, compressed test sets may activate several faults and a high degree of fault masking can occur. Consequently, this seems to indicate that a TPG for MSFs is necessary.

5.3. Multiple Fault TPG Experiments

We performed two experiments on the benchmarks. The first one applies Random TPG (RTPG) as the first step, while in the second one RTPG was omitted. The results are summarized in Tables VI and VII, respectively. RTPG applies random vectors and is stopped when 64 consecutive vectors do not detect any additional fault, using the fault analysis method of [24] . The backtracking limit in TPG is set to 10. A target fault is aborted when the number of backtracks exceeds the limit without generating a test for the fault. The number of undetected faults after TPG is shown in the second column. It includes redundant faults (if any). The unique sensitization procedure and the global implications learned during preprocessing highly contribute to the identification of 95% of redundant faults without backtracking; they are removed from the fault list. The number of traversals over the fault list is given in the third column. Recall that this list is repeatedly processed as long as additional faults can be dropped.

TABLE VI
TPG RESULTS INCLUDING INITIAL RANDOM TPG

Circuit	Unde-tected	Traver-sals	Back-tracks	Vectors	Fault Coverage	CPU Time (sec.)		
						RTPG	TPG	Total
c432	4	2	45	622	99.4%	4.5	2.0	6.5
c499	8	5	1447	1271	99.1%	2.8	61.8	64.6
c880	0	1	0	701	100%	8.2	3.5	11.7
c1908	20	4	1566	1737	99.4%	10.2	203.2	213.4
c2670	381	2	3312	2477	91.4%	26.6	387.8	414.3
c3540	167	2	627	2300	97.2%	108.3	170.3	278.6
c5315	614	2	5622	2567	93.2%	99.5	1464.5	1564.0
c6288	50	3	560	2039	99.5%	416.7	541.4	958.1
c7552	1231	2	12337	5322	90.2%	302.3	4709.6	5011.9

TABLE VII
TPG RESULTS WITHOUT RTPG

Circuit	Unde-tected	Traver-sals	Back-tracks	Vectors	Fault Coverage	CPU Time (sec.)
c432	4	2	78	532	99.4%	5.4
c499	23	5	1658	1160	97.4%	97.2
c880	0	1	20	977	100%	12.6
c1908	18	4	1752	1960	99.4%	227.7
c2670	392	2	3367	2519	91.1%	409.7
c3540	168	3	1122	3400	97.2%	342.0
c5315	614	2	5600	2231	93.2%	1493.1
c6288	63	3	2142	6056	99.3%	1247.5
c7552	1229	2	12557	6954	90.2%	4936.5

The use of the sophisticated heuristics from FAN and SOCRATES considerably reduce backtracking, however, the conditions for propagating the target fault effect and for detecting the fault are more strict than under the SSF model. For instance, even if the target fault effect on a line for which no normal path has been yet found is propagated to the primary outputs, it is not declared detected unless it can be propagated through all paths. Hence, the high number of backtracking required on some benchmarks is mainly due to the TPG trying other assignments with the objective to propagate the fault effect through all paths. The test sets shown in the fifth column are not compressed.

As expected, TPG for MSFs requires more CPU time than for SSFs, but achieves a fault coverage which is quite high or even complete (e.g., c880). The generated test sets guarantee the detection of at least all the multiple faults containing the dropped faults. Compared to the time required by SOCRATES to derive tests for SSFs, our system is about 8 times slower on the average with a standard deviation of 9. If we consider the complexity of deriving tests for MSFs, this CPU time is quite acceptable, especially if we assume that the SSF model is not adequate.

The c1908 circuit contains 9 redundant SSFs [32, 35] . Under the MSF model, the TPG identifies these redundant faults without backtracking. The undetected faults that are not redundant are masked by the redundant ones. For instance, consider a small part of the c1908 circuit shown in Fig. 13. The faults s^1_{112}, s^1_{420} and s^1_{421} are redundant.

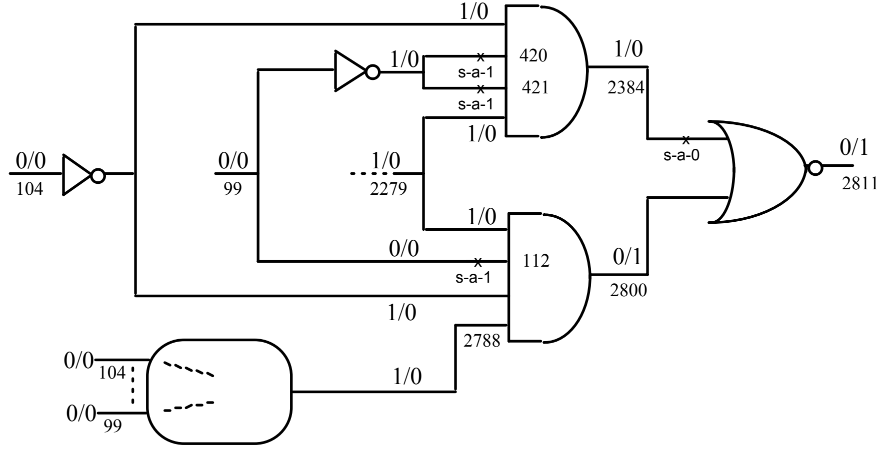


Fig. **13**. A part of the c1908 circuit.

In addition to the redundant faults, s^0_{2384} remains undetected. The TPG attempts to detect this fault by performing the following steps:

(1) To activate the fault s^0_{2384}, line 2279 is assigned 1/0 and the primary inputs 99 and 104 are assigned 0/0.

(2) To propagate the effect of s^0_{2384} to the primary output 2811, line 2800 should be assigned 0/0. Since all inputs of the AND gate feeding line 2800 have been justified, no assignments can be made to satisfy the objective 0/0 on line 2800. The target fault may be masked at the output 2811.

The unique alternative to detect s^0_{2384} is to confirm the absence of s^1_{112}. Since it is redundant, no algorithmic way can confirm the absence or presence of this fault. Therefore, s^0_{2384} remains undetected because it can be masked by s^1_{112} (under the SSF model it is testable, however). The multiple fault (s^1_{112}, s^0_{2384}) is then redundant. This kind of information can be extracted from the list of undetected faults after TPG, and can be helpful in determining the possible fault masking relations.

A final note about the c1355 circuit: This circuit is functionally equivalent to c499 in which the XOR gates are expanded into their NAND gate equivalents. This expansion introduces a high degree of ambiguity into the analysis of MSFs for TPG. Since our method is conservative in nature, the target fault effect is lost (i.e., sensibility) each time it is propagated into a possibly faulty NAND structure of a XOR gate. Fault masking cycles are thus created and the target fault is aborted, which has a great impact on the fault coverage. To break such cycles, one should perhaps explicitly consider the masking relations as presented in [8] .

6. Conclusions

The test pattern generation method presented in this paper is, to our knowledge, the first practical attempt at generating tests for multiple stuck-at faults. The method assumes the presence of all multiple faults of all multiplicities without resorting to their explicit enumeration. New line and gate models were introduced to handle multiple fault propagation. Test conditions were defined so that all multiple faults containing a detected target fault as a component are also detected. The search for test vectors is guided using several techniques from earlier TPG systems for SSFs which were adapted for MSFs. The method thus achieves high multiple fault coverage at reasonable cost. No previous work reported results in MSF TPG on the ISCAS'85 benchmark circuits due to the complexity of the problem.

As seen in the experimental results, MSF detection using test sets developed for SSFs may not achieve a good MSF coverage. In particular, compressed test sets may activate several faults and a high degree of fault masking can occur. The test sets developed using our TPG method guarantee the detection of all multiple faults containing one or more of the detected faults, and are valid even in the presence of redundancy. Due to the conservatism of the method, the fault coverage obtained is a lower bound on the actual coverage that can be really achieved for the circuit under test.

The implementation of the TPG method can be further improved using some recently developed techniques [32] . New heuristics can also be added to improve the performance, namely, the order in which target faults are selected, fault dependencies and masking relations, and a deeper analysis of the circuit structure to reduce backtracking when the target fault site may be hidden. Also, since the method does not explicitly enumerate multiple faults and due to its conservatism, it is not a complete algorithm, i.e., a test may exist for a MSF, but the method will not find it. We could envisage generating tests for each remaining multiple faults (by enumerating them), however, this is feasible if the number of such faults is relatively small.

REFERENCES

[1] M. Abramovici, M.A. Breuer, “Multiple Fault Diagnosis in Combinational Circuits Based on an Effect-Cause Analysis.”, *IEEE Trans. on Computers,* vol. C-29, 1980, pp. 451-460.

[2] M. Abramovici, M.A. Breuer, A.D. Freidman, *Digital Systems Testing and Testable Design*, Computer Science Press, 1990.

[3] M. Abramovici, J.J. Kulikowski, P.R. Menon, D.T. Miller, “SMART and FAST: Test Generation for VLSI Scan-Design Circuits.”, *IEEE Design & Test,* 1986, pp. 43-54.

[4] V.K. Agarwal, A.S.F. Fung, “Multiple Fault Testing of Large Circuits by Single Fault Test Sets.”, *IEEE Trans. on Computers,* vol. C-30, no. 11, 1981, pp. 855-865.

[5] D.C. Bossen, S.J. Hong, “Cause-Effect Analysis for Multiple Fault Detection in Combinational Networks.”, *IEEE Trans. on Computers,* vol. C-20, 1971, pp. 1252-1275.

[6] M.A. Breuer, A.D. Friedman, *Diagnosis & Reliable Design of Digital Systems*, Computer Science Press, 1976.

[7] F. Brglez, H. Fujiwara, “A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortran”, *Proc. of the Intl. Symp. Circuits and Systems*, 1985,

[8] C.W. Cha, “Multiple Fault Diagnosis in Combinational Networks”, *Proc. of the 16th Design Automation Conf.*, 1979, pp. 149-155.

[9] H. Cox, A. Ivanov, V.K. Agarwal, J. Rajski, “On Multiple Fault Coverage and Aliasing Probability Measures”, *Proc. of the Intl. Test Conf.*, 1988, pp. 314-321.

[10] H. Cox, J. Rajski, “A Method of Fault Analysis for Test Generation and Fault Diagnosis.”, *IEEE Trans. on Computer-Aided Design,* vol. 7, no. 7, 1988, pp. 813-833.

[11] F.J.O. Dias, “Fault Masking in Combinatorial Logic Circuits.”, *IEEE Trans. on Computers,* vol. C-24, no. 6, 1975, pp. 476-482.

[12] A.D. Friedman, “Fault Detection in Redundant Circuits.”, *IEEE Trans. Electron. Comput.,* vol. EC-16, 1967, pp. 99-100.

[13] H. Fujiwara, “Computational Complexity of Controllability/Observability Problems for Combinational Circuits”, *Proc. of the 18th Fault-Tolerant Computing Symp.*, 1988, pp. 64-69.

[14] H. Fujiwara, T. Shimono, “On the Acceleration of Test Generation Algorithms”, *Proc. of the 13th Fault-Tolerant Computing Symp.*, 1983, pp. 98-105.

[15] P. Goel, “An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits.”, *IEEE Trans. on Computers,* vol. C-30, no. 3, 1981, pp. 215-222.

[16] L.H. Goldstein, E.L. Thigpen, “SCOAP: Sandia Controllability/Observability Analysis Program”, *Proc. of the 17th Design Automation Conf.*, 1980, pp. 190-196.

[17] J.P. Hayes, “A NAND Model for Fault Diagnosis in Combinational Logic Networks.”, *IEEE Trans. on Computers,* vol. C-20, no. 12, 1971, pp. 1496-1506.

[18] Y. Huan, S.Q. Li, “The "G-F" 2-Valued Formula Generating Complete Set of Tests for Multiple Faults”, *Proc. of the Intl. VLSI Conf.*, 1991, pp. 343-349.

[19] J.L.A. Hughes, “Multiple Fault Detection Using Single Fault Test Sets.”, *IEEE Trans. on Computer-Aided Design,* vol. 7, no. 1, 1988, pp. 100-108.

[20] J.L.A. Hughes, E.J. McCluskey, “Multiple Stuck-at Fault Coverage of Single Stuck-at Fault Test Sets”, *Proc. of the Intl. Test Conf.*, 1986, pp. 368-374.

[21] O.H. Ibarra, S.K. Sahni, “Polynomially Complete Fault Detection Problems.”, *IEEE Trans. on Computers,* vol. C-24, no. 3, 1975, pp. 242-249.

[22] J. Jacob, N.N. Biswas, “GTBD Faults and Lower Bounds on Multiple Fault Coverage of Single Fault Test Sets”, *Proc. of the Intl. Test Conf.*, 1987, pp. 849-855.

[23] Y. Karkouri, E.M. Aboulhamid, “Complexité du test des circuits logiques.”, *Technique et Science Informatiques,* vol. 9, no. 4, 1990, pp. 273-287.

[24] Y. Karkouri, E.M. Aboulhamid, “Multiple Stuck-at Fault Diagnosis in Logic Circuits.”, *Accepted in the Canadian Conf. on VLSI,* 1992.

[25] Y. Karkouri, E.M. Aboulhamid, E. Cerny, A. Verreault, “Use of Fault Dropping for Multiple Fault Analysis.”, *to appear in IEEE Trans. on Computers,* 1992.

[26] T. Kirkland, M.R. Mercer, “A Topological Search Algorithm for ATPG”, *Proc. of the 24th ACM/IEEE Design Automation Conf.*, 1987, pp. 502-508.

[27] I. Kohavi, Z. Kohavi, “Detection of Multiple Faults in Combinational Logic Networks.”, *IEEE Trans. on Computers,* vol. C-21, no. 6, 1972, pp. 556-568.

[28] K. Kubiak, W.K. Fuchs, “Multiple-Fault Simulation and Coverage of Deterministic Single-Fault Test Sets”, *Proc. of the Intl. Test Conf.*, 1991, pp. 956-962.

[29] E.J. McCluskey, F.W. Clegg, “Fault Equivalence in Combinational Logic Networks.”, *IEEE Trans. on Computers,* vol. C-20, no. 11, 1971, pp. 1286-1293.

[30] I. Pomeranz, L.N. Reddy, S.M. Reddy, “COMPACTEST: A Method to Generate Compact Test Sets for Combinational Circuits”, *Proc. of the IEEE Intl. Test Conf.*, 1991, pp. 194-203.

[31] J. Rajski, H. Cox, *Personal Communication*, 1990.

[32] J. Rajski, H. Cox, “A Method to Calculate Necessary Assignments in Algorithmic Test Pattern Generation”, *Proc. of the Intl. Test Conf.*, 1990, pp 25-34.

[33] J.P. Roth, W.G. Bouricius, P.R. Shneider, “Programmed Algorithms to Computer Tests to Detect and Distinguish between Failures in Logic Circuits.”, *IEEE Trans. on Elect. Comput.,* vol. EC-16, no. 5, 1967, pp. 567-580.

[34] D.R. Schertz, G. Metze, “On the Design of Multiple Faults Diagnosable Networks.”, *IEEE Trans. on Computers,* vol. C-20, 1971, pp. 1361-1364.

[35] M.H. Schultz, E. Auth, “Improved Deterministic Test Pattern Generation with Applications to Redundancy Identification.”, *IEEE Trans. on Computer-Aided Design,* vol. 8, no. 7, 1989, pp. 811-816.

[36] M.H. Schultz, E. Trischler, T.M. Sarfert, “SOCRATES: A Highly Efficient Automatic Test Pattern Generation System.”, *IEEE Trans. on Computer-Aided Design,* vol. 7, no. 1, 1988, pp. 126-137.

[37] J.P. Shen, W. Maly, F.J. Ferguson, “Inductive Fault Analysis of MOS Integrated Circuits.”, *IEEE Design & Test,* vol. 2, no. 12, 1985, pp. 13-26.

[38] J.E. Smith, “On Necessary and Sufficient Conditions for Multiple Fault Undetectability.”, *IEEE Trans. on Computers,* vol C-28, no. 10, 1979, pp. 801-802.

[39] A. Verreault, E.M. Aboulhamid, Y. Karkouri, “Multiple Fault Analysis using a Fault Dropping Technique”, *Proc. of the 21th Fault-Tolerant Computing Symp.*, 1991, pp. 162-169.

APPENDIX I
TPG EQUATIONS

In this appendix, we present the equations used in TPG for the propagation of fault effects, the propagation of sensibility and the inclusion of the gate in the *S*-frontier for all gate types. We assume an *m*-input gate *G* with the output *out*, except XOR and XNOR gates which have two inputs only.

A. Fault effect propagation

P and Q are predicates; P is true if p_{out} can be equal to 0; Q is true if p_{out} can be equal to 1.

AND / NAND:

- $P = [(\exists i, (n_i{=}0,X \wedge p_i{=}0,X \wedge s_i^1{=}0)) \vee (\forall i, (n_i{=}0,X \wedge p_i{=}0,X))] \vee [\forall i, (n_i{=}1 \wedge p_i{=}0,X)]$

- $Q = [(\exists i,\ n_i{=}1,X \vee p_i{=}1,X) \wedge (\forall i,\ n_i{=}0,X \Rightarrow (p_i{=}1,X \vee s_i^1{=}1))] \vee [(\forall i,\ n_i{=}1) \wedge (\exists i,\ p_i{=}1,X)]$
- $p_{out} = 0 \Leftrightarrow (P \wedge \overline{Q})$; $p_{out} = 1 \Leftrightarrow (\overline{P} \wedge Q)$; $p_{out} = X \Leftrightarrow (P \wedge Q)$;

OR / NOR:

- $P = [(\exists i,\ (n_i{=}1,X \wedge p_i{=}0,X \wedge s_i^0{=}0)) \vee (\forall i,\ (n_i{=}1,X \wedge p_i{=}0,X))] \vee [\forall i,\ (n_i{=}0 \wedge p_i{=}0,X)]$
- $Q = [(\exists i,\ n_i{=}0,X \vee p_i{=}1,X) \wedge (\forall i,\ n_i{=}1,X \Rightarrow (p_i{=}1,X \vee s_i^0{=}1))] \vee [(\forall i,\ n_i{=}0) \wedge (\exists i,\ p_i{=}1,X)]$
- $p_{out} = 0 \Leftrightarrow (P \wedge \overline{Q})$; $p_{out} = 1 \Leftrightarrow (\overline{P} \wedge Q)$; $p_{out} = X \Leftrightarrow (P \wedge Q)$;

XOR / XNOR:

- $P = [\forall i,\ \ p_i{=}0,X \wedge ((n_i{=}0,X \wedge s_i^1{=}0) \vee (n_i{=}1,X \wedge s_i^0{=}0))]$
- $Q = (\exists i,\ \ p_i{=}1,X) \vee (\exists i,\ (n_i{=}0,X \wedge s_i^1{=}1) \vee (n_i{=}1,X \wedge s_i^0{=}1))$
- $p_{out} = 0 \Leftrightarrow (P \wedge \overline{Q})$; $p_{out} = 1 \Leftrightarrow (\overline{P} \wedge Q)$; $p_{out} = X \Leftrightarrow (P \wedge Q)$;

B. Sensibility propagation

AND / NAND: $w_{out} = 1 \Leftrightarrow (\exists i,\ w_i{=}1) \wedge [(\forall i,\ n_i{=}1) \vee (\forall i,\ (n_i{=}0 \wedge w_i{=}1) \vee (w_i{=}0 \wedge n_i{=}1 \wedge p_i{=}0))]$

OR / NOR: $w_{out} = 1 \Leftrightarrow (\exists i,\ w_i{=}1) \wedge [(\forall i,\ n_i{=}0)] \vee (\forall i,\ (n_i{=}1 \wedge w_i{=}1) \vee (w_i{=}0 \wedge n_i{=}0 \wedge p_i{=}0))]$

XOR / XNOR: $w_{out} = 1 \Leftrightarrow (\exists i,\ w_i{=}1) \wedge [p_j{=}0 \wedge w_j{=}0 \wedge ((n_j{=}1 \wedge s_j^0{=}0) \vee (n_j{=}0 \wedge s_j^1{=}0))], j \neq i$

C. Inclusion in the S-frontier

AND / NAND: G □ S-frontier □ $(\exists i,\ w_i{=}1)$ □
$[(\forall i,\ n_i{=}1,X) \wedge (\exists i,\ n_i{=}X)] \vee [(\forall i,\ (n_i{=}0 \wedge w_i{=}1) \vee (w_i{=}0 \wedge n_i{=}1,X \wedge p_i{=}0,X)) \wedge (\exists i,\ n_i{=}X \vee p_i{=}X)]$

OR / NOR: G □ S-frontier □ $(\exists i,\ w_i{=}1)$ □
$[(\forall i,\ n_i{=}0,X) \wedge (\exists i,\ n_i{=}X)] \vee [(\forall i,\ (n_i{=}1 \wedge w_i{=}1) \vee (w_i{=}0 \wedge n_i{=}0,X \wedge p_i{=}0,X)) \wedge (\exists i,\ n_i{=}X \vee p_i{=}X)]$

XOR / XNOR: G □ S-frontier □ $(\exists i,\ w_i{=}1)$ □
$[p_j{=}0,X \wedge w_j{=}0 \wedge ((n_j{=}1,X \wedge s_j^0{=}0) \vee (n_j{=}0,X \wedge s_j^1{=}0)) \wedge (n_j{=}X \vee p_j{=}X)], j \neq i$

APPENDIX II
DEDUCTION LEMMAS

For each gate type, the backward deduction phase applies the corresponding Lemma to deduce the propagation bit and the status on the inputs of the gate.

Lemma 1: If the propagation bit p_g on the output g of an AND/NAND gate is reset to 0, then the following are sufficient conditions for dropping the fault effect and the fault on each input i:

- $p_i = 0 \square\ p_i=1 \square [(n_i=0 \square (\forall j\neq i, n_j=1 \square p_j=0)) \Delta (\forall j, n_j=1)]$
- $s_i^1 = 0 \square\ n_i=0 \square (\forall j\neq i, n_j=1 \square p_j=0)$

Lemma 2: Let s be a fanout stem. Sufficient conditions for resetting the propagation bit p_s to 0 are:

- $\forall$ fanout branch b of s, p_b=0, <u>or</u>
- $\exists$ fanout branch b of s such that p_b=0 and b is normal.

Lemma 3: If the propagation bit p_g on the output g of an OR/NOR gate is reset to 0, then the following are sufficient conditions for dropping the fault effect and the fault on each input i:

- $p_i = 0 \square\ p_i=1 \square [(n_i=1 \square (\forall j\neq i, n_j=0 \square p_j=0)) \Delta (\forall j, n_j=0)]$
- $s_i^0 = 0 \square\ n_i=1 \square (\forall j\neq i, n_j=0 \square p_j=0)$

Lemma 4: If the propagation bit p_g on the output g of an XOR/XNOR gate is reset to 0, then the following are sufficient conditions for dropping the fault effect and the fault on each input i:

- $p_i = 0 \square\ p_i=1 \square p_j=0 \square [(n_j=0 \square s_j^1=0) \Delta (n_j=1 \square s_j^0=0)], j \neq i$
- $s_i^1 = 0 \square\ n_i=0 \square p_j=0, j \neq i$
- $s_i^0 = 0 \square\ n_i=1 \square p_j=0, j \neq i$

APPENDIX III
BACKWARD DEDUCTION PROCEDURE

The variables associated with each line are the following:
n_i : normal value;
p_i : propagation bit;
$h_i = 1$ if there is no normal path from line i to a primary output;
$h_i = 0$ if there is a normal path from line i to a primay output
$s_i^\alpha = 1$ if the fault s-a-α is still possible on line i.
$s_i^\alpha = 0$ if the fault s-a-α is dropped on line i.

The following procedure performs the backward deduction phase of the TPG:

```
procedure Backward_Deduction();
begin
    for each line g in reverse topological order do
        if (n_g ≠ X) and (p_g ≠ X) then begin
            if g is a PO then begin
                p_g := 0;                                    /* Fault free value is observed on the PO */
                if (s_g^1 = 0) and (s_g^0 = 0) then h_g := 0;   /* PO is normal if all its faults are dropped */
            end
            else if g is a fanout stem then begin
                Apply_Stem_Lemma( g );
                if (∃ branch j of g such that h_j = 0) then h_g := 0;
            end
            else begin
                Apply_Gate_Lemma( Gate feeding g );
                if h_g = 0 then
                    for each input i of the gate feeding g do
                        if (s_i^1 = 0) and (s_i^0 = 0) then h_i := 0
            end;
        end;
end;
```

APPENDIX IV
MULTIPLE BACKTRACE PROCEDURE

In this appendix, we present the multiple backtrace procedure that traces more than one path to satisfy a set of objectives. The procedure uses 3 sets of objectives: *Head_Objectives* for head lines, *Stem_Objectives* for fanout stems, and *Current_Objectives* which contains the unjustified bound lines during the *Imply_and_Check*, the non-sensible inputs (if any) that propagate the target fault effect and the new generated objectives at each iteration of the procedure. An objective is a 3-tuple *(i, val, prop)*, where *i* is the line number, *val* is the chosen value for n_i and *prop* is the requested value for p_i.

As in FAN algorithm, we associate to each line two counters n_0 and n_1 that indicate the number of times the values 0 and 1, respectively, are requested for the normal value of this line. These counters are updated using the procedure *Increment_Request(Line, Value);*

```
procedure Multiple_Backtrace( Current_Objectives ) : return (i, val, prop);
begin

   while Current_Objectives ≠ □ do
   begin
      remove one tuple (i, val, prop) from Current_Objectives;
      if i is a head line then
         add (i, val, prop) to Head_Objectives;
      else if i is a fanout branch then begin
            s := stem( i );
            Increment_Request (s, val);
            add (s, val, prop) to Stem_Objectives;
      end
      else Determine_Gate_Objectives (i, val, prop, Current_Objectives);
   end;

   if Stem_Objectives ≠ □ then begin
      (s, val, prop) := Choose_Highest_Level_Stem( Stem_Objectives );
      val := most requested normal value on s;
      if prop = 0 was requested on s with val then
         prop := 0
      else prop := X;
      if   (s has contradictory requirements) and
         (s is not reachable from the target fault site) then
         return (s, val, prop);
      add (s, val, prop) to Current_Objectives;
      return( Multiple_Backtrace( Current_Objectives ) );
   end;

   remove on tuple (i, val, prop) from Head_Objectives;
   return (i, val, prop);

end;
```

```
procedure Choose_Gates_Objectives (gate, gate_val, gate_prop, Current_Objectives);
begin

    if gate is an inverter or a buffer gate then begin
        i := input of gate;
        val := gate_val ⊕ (inversion of gate);
        add (i, val, gate_prop) to Current_Objectives;
    end

    else if gate is an AND, NAND, OR or NOR gate then begin
            c := controlling normal value of gate;                /* AND/NAND: c = 0;  OR/NOR: c = 1 */
            if gate_val ⊕ (inversion of gate) ≠ c then begin
                for every input i of gate do
                    if (n_i = X) or (p_i = X and gate_prop = 0) then begin
                        Increment_Request (i, c̄);
                        add (i, c̄, gate_prop) to Current_objectives;
                    end
            end
            else if gate_prop ≠ 0 then begin    /* Consider normal values only on the inputs of gate */
                    Select an input i of gate with the lowest cost for c;      /* Selected according to its controllability value */
                    Increment_Request (i, c);
                    add (i, c, X) to Current_objectives;
                end
            else begin          /* The propagation bit on the output of gate is requested to be 0 */
                    Select an input i of gate with the lowest cost for c with s_i^c̄ = 0;
                    if i ≠ 0 then begin        /* This is the input when set to c/0 disables fault effect on the gate output */
                        Increment_Request (i, c);
                        add (i, c, 0) to Current_objectives;
                    end
                    else for every input i of gate do       /* All inputs have s_i^c̄ = 1 □ all of them must be equal to c/0 */
                            if n_i = X or p_i = X then begin
                                Increment_Request (i, c);
                                add (i, c, 0) to Current_Objectives;
                            end;
                end
    end

    else Choose_XOR_Gates (gate, gate_val, gate_prop, Current_Objectives);

end;
```

```
procedure Choose_XOR_Gates (gate, gate_val, gate_prop, Current_Objectives);
begin
    i := First input of gate;  j := Second input of gate;

    C00 := i.C0 + j.C0;   C11 := i.C1 + j.C1;   /* C0 and C1: Values of the controllability to 0 and 1 */
    C01 := i.C0 + j.C1;   C10 := i.C1 + j.C0;

    if gate_prop ≠ 0 then begin
        if gate_val = 0 then begin              /* if gate_val = 1 ... for XNOR gate */
            if n_i = X and n_j = X then
                if C00 < C11 then begin i_val := 0;  j_val := 0 end else begin i_val := 1;  j_val := 1; end
            else if n_i ≠ X and n_j = X then j_val := n_i
            else if n_i = X and n_j ≠ X then i_val := n_j
        end
        else begin
            if n_i = X and n_j = X then
                if C01 < C10 then begin i_val := 0;  j_val := 1 end else begin i_val := 1;  j_val := 0; end
            else if n_i ≠ X and n_j = X then j_val := \overline{n_i}
            else if n_i = X and n_j ≠ X then i_val := \overline{n_j}
        end
    end
    else begin          /* 0 is requested for the propagation bit on the gate output */
        if gate_val = 0 then begin              /* if gate_val = 1 ... for XNOR gate */
            if C00 < C11 then begin  i_val := 0; j_val := 0;
                if n_i = 1 or s_i^1 = 1 or n_j = 1 or s_j^1 = 1 then begin i_val := 1; j_val := 1; end;
            end
            else begin  i_val := 1; j_val := 1;
                if n_i = 0 or s_i^0 = 1 or n_j = 0 or s_j^0 = 1 then begin i_val := 0; j_val := 0; end;
            end;
        end
        else begin
            if C01 < C10 then begin  i_val := 0; j_val := 1;
                if n_i = 1 or s_i^1 = 1 or n_j = 0 or s_j^0 = 1 then begin i_val := 1; j_val := 0; end;
            end
            else begin  i_val := 1; j_val := 0;
                if n_i = 0 or s_i^0 = 1 or n_j = 1 or s_j^1 = 1 then begin i_val := 0; j_val := 1; end;
            end;
        end;
    end;

    if i_val ≠ X then begin
        Increment_Request (i, i_val);
        Add (i, i_val, gate_prop) to Current_Objectives;
    end;

    if j_val ≠ X then begin
        Increment_Request (j, j_val);
        Add (j, j_val, gate_prop) to Current_Objectives;
    end;
```

```
end;
```