Use of Fault Dropping for Multiple Fault Analysis

Younès KARKOURI, El Mostapha ABOULHAMID, Eduard CERNY and Alain VERREAULT

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

ABSTRACT

A new approach to fault analysis is presented. We consider multiple stuck-at-0/1 faults at the gate level. First, a fault collapsing phase is applied to the network, so that equivalent faults are eliminated. During the analysis we consider frontier faults where there is at least a normal path from each faulty line to a primary output. It is shown that the set of frontier faults is equivalent to the set of multiple faults. Given an input vector, we evaluate the fault free circuit and then propagate fault effects. Assuming that fault free response is observed, a fault dropping procedure is then applied to eliminate faulty conditions on lines, that are either absent or may be hidden by other faulty conditions. This method is applied to some benchmark circuits and achieves high degree of efficiency.

Keywords: logic circuits, stuck-at faults, fault collapsing, fault dropping, multiple fault analysis.

I. INTRODUCTION

This paper presents a new approach to multiple fault analysis. Given a set of input vectors, our objective is to determine the set of multiple stuck-at-0/1 (s-a-0/1) faults that are not present in the circuit under test (CUT). Multiple faults have been considered as very difficult since a m-line circuit may have 3^m - 1 faulty situations compared to $2m$ faulty situations under the single fault model. Usually, research has been directed to fault simulation and ATPG for single stuck-at faults, assuming that a circuit is frequently tested so that it does not contain more than one fault at a time. This frequent test strategy was shown to be inefficient in the presence of redundant faults [8], when a defect appears as a multiple fault or when test sets are incomplete [9]. Other works showed that test sets generated for single faults detect a high number of multiple faults, but this is valid only for some classes of circuits [2] (depending on their structures), multiple faults of small multiplicities [9] and the adopted measures to evaluate the fault coverage [7].

Even if it seems very hard to tackle the problem of multiple faults, this model is very important in the design of logic circuits [1, 3, 5, 6]. Several analysis methods have been proposed [1, 5, 6, 11] for finding multiple faults not detected by a given set of input vectors, and dropping faults that cannot be present in the circuit. However, these methods have a limited possibility of fault dropping: No fault is dropped unless a normal path (i.e., faultless path) has been deduced from its site to a primary output, and its effect is observable on a primary output along this path.

In this paper, we propose a multiple fault analysis method using a new fault dropping technique. The method can be summarized as follows: First, a fault collapsing phase is applied to the network, so that equivalent faults on lines are eliminated. After this phase, only combinations of remaining faulty lines are considered. Furthermore, these combinations are restricted to frontier faults, each one of them has at least one normal path from each faulty line to a primary output. We show that the set of frontier faults is equivalent to the set of multiple faults. Given an input vector, we evaluate the fault free circuit and propagate fault effects. Assuming that fault free response is observed, a fault dropping procedure is then applied to eliminate faulty conditions on specific lines that are either absent or hidden by other faulty conditions. This premature fault dropping on possibly hidden lines is based on the frontier fault model and significantly improves the performance of the method. Frontier faults are enumerated implicitly, that is each time we drop a fault on a line, a whole class of frontier faults that involve this line is in fact dropped because no fault masking can occur. The method manipulates single vectors to model the circuit behavior under multiple faults. This makes our approach much simpler than methods requiring manipulation of masking expressions between faults [5], backtracking [1] or pairs of vectors [6, 11]. Experiments were performed on the ISCAS'85 benchmark circuits and high fault coverage is achieved at reasonable cost.

The rest of the paper is organized as follows: Section II introduces fault collapsing and the fault model. Section III presents the fault analysis method. Section IV reports experimental results, and finally we conclude in Section V.

II. FAULT MODEL

Our method is based on early fault dropping of multiple faults. This is done in two manners: Fault collapsing which is based on the circuit topology only, and fault dropping which is performed during the analysis and based on the frontier fault model.

A. Fault Collapsing

We consider faults on circuit lines only; the gates are assumed to perform fault free functions. In order to reduce the number of faults to deal with, multiple fault collapsing has been proposed in [3, 5]. It is shown that any multiple fault is equivalent to a combination of faults on checkpoints [3]. Primary inputs that do not fan out and fanout branches of the circuit are checkpoints. Cha [5] showed that if the number of checkpoints is c, then the number of faulty conditions is 3^c - 1 which may be large even for small values of c. He then introduced the notion of prime faults for circuits consisting of AND, NAND, OR, and NOR gates only. Under this fault model, there is at most one fault on each line of the circuit, and the number of faulty conditions is 2^p - 1, where p is the number of prime faults.

Our approach to fault collapsing is based on the intuitive fact that a fault effect is easier to observe if it is closer to primary outputs. Therefore we delay the effect of a fault as far as possible. For example, if there is a s-a-0 on an input of an AND gate, we remove it and place it on the output. We do not have to consider that all inputs of an AND gate are simultaneously s-a–1, because we assume a s-a-1 fault on its output. The resulting procedure from our collapsing is the following [12]:

for all gates in the circuit put:

- s-a-1 (s-a-0) faults on all inputs of any AND/NAND (OR/NOR) gate (not present simultaneously on all inputs).
- No fault on the input of an inverter or on a fanout stem.
- s-a-0 and s-a-1 on inputs of XOR and XNOR gates (not present simultaneously on all inputs).
- Both s-a-0 and s-a-1 on primary outputs.

Experimentations show that using our collapsing procedure we consider at most 4% more faults than [5]. This percentage is roughly the percentage of combinations of faults that exist explicitly in the circuit (for example, simultaneous faults on all inputs of a gate are represented explicitly by a fault on the output). Note, however, that on the ISCAS'85 benchmark circuits the number of remaining faults after our collapsing is 30% less than the number of faults on checkpoints (each checkpoint has two faults).

Fig. 1 illustrates the remaining faults when the collapsing procedure is applied to the example circuit: lines 3, 4, 5 and 6 can be s-a-1 or normal; lines 7 and 8 can be s-a-0 or normal; line 9 can be s-a-0, s-a-1 or normal. Lines which are inputs of the same gate, i.e., (4, 5), (3, 6) and (7, 8), cannot be faulty at the same time. There are 80 multiple faults to be considered after fault collapsing. However, our analysis method will not enumerate them.

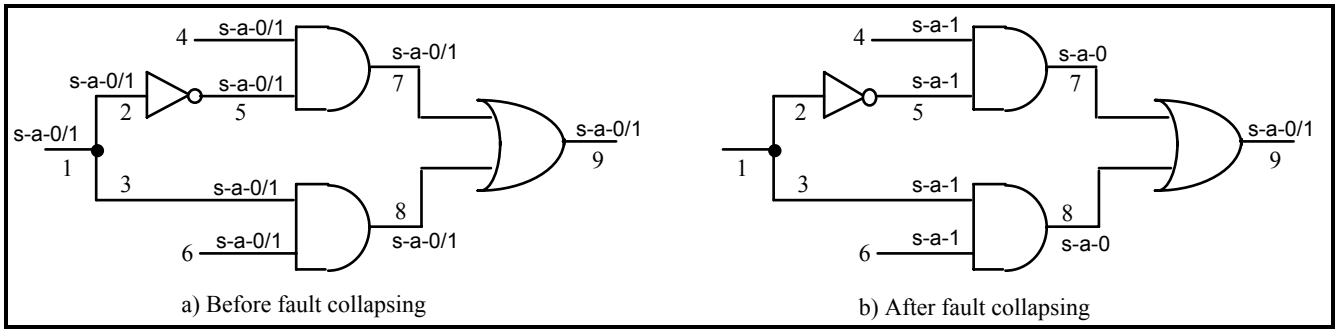

Fig. 1. Fault collapsing example.

B. Fault Model

We consider multiple faults which are combination of faults remaining after collapsing. We then characterize frontier faults which are equivalent to all multiple faults. This characterization allows us to drop a fault from a line even if its effect may be hidden.

In a circuit consisting of m lines, a multiple fault is represented by a tuple with at most m components and denoted by $f = (f_i^{\alpha}, f_j^{\beta}, \ldots)$, $i \neq j$, where f_i^{α} represents the status of line i : $\alpha = 0$ for s-a-0, $\alpha = 1$ for s-a-1. A line k missing in the tuple is not faulty in that specific multiple fault.

Definition: A path from line i to a primary output is said *normal* if all lines along this path are normal, but the other inputs to gates along this path may be faulty.

Definition: Let $f = (f_i^{\alpha}, f_j^{\beta}, \ldots)$ be a multiple fault. Line i is a *faulty line* iff f_i^{α} belongs to f and there exists at least one normal path from line i to a primary output.

Definition: Let $f = (f_i^{\alpha}, f_j^{\beta}, \ldots)$ be a multiple fault; then f is a *frontier fault* iff for each fault component f_i^{α}, line i is a faulty line.

The following lemma shows that each multiple fault is equivalent to a frontier fault, then as a consequence, if a test set covers all frontier faults then it will cover all multiple faults.

Lemma 1: Every multiple fault $f = (f_i^{\alpha}, f_j^{\beta}, \ldots)$ is equivalent to a frontier fault.

Proof: Construct a multiple fault f_r from f as the following: For all components f_i^{α}, if there is no normal path from line i to a primary output then remove f_i^{α} from f because line i is hidden. The resulting multiple fault f_r will contain only components f_i^{α} of faulty lines, hence f_r is a frontier fault. QED

Example: For the circuit in Fig. 1, the number of frontier faults is only 16 compared to 80 multiple faults after collapsing. The set of frontier faults is: $\{(f_3^1), (f_4^1), (f_5^1), (f_6^1), (f_7^0), (f_8^0), (f_9^0), (f_9^1), (f_3^1, f_4^1), (f_3^1, f_5^1), (f_3^1, f_7^0), (f_4^1, f_6^1), (f_4^1, f_8^0), (f_5^1, f_6^1), (f_5^1, f_8^0), (f_6^1, f_7^0)\}$.

According to the previous definition, a frontier fault f in the circuit under test (CUT) partitions the lines into three categories, *Hidden Lines*, *Faulty Lines* and *Normal Lines*, defined as follows:

- Line i is *faulty* if $f_i^{\alpha} \square f$.

- Line j is *normal* if $f_j^{\alpha} \square f$ and there is a normal path from line j to a primary output.
- Line k is *hidden* if there is no normal path from k to a primary output.

During fault analysis, we assume that the circuit under test contains one frontier fault consisting of faults that has not been yet dropped. The effective values (real values) on hidden lines are unknown since they are unobservable due to the frontier fault, and there is no algorithmic way to determine these values (Normal Path Theorem in [1]). In our case, we assume that *these lines carry fault free values*. As it will be seen in Section III.C, this assumption does not invalidate the deductions made during analysis.

Example: For the circuit in Fig. 2, the presence of the frontier fault (f_a^1, f_j^0) partitions its lines as follows: Hidden lines = {d, g, h}; Faulty lines = {a, j}; Normal lines = {b, c, e, i, k, m}.

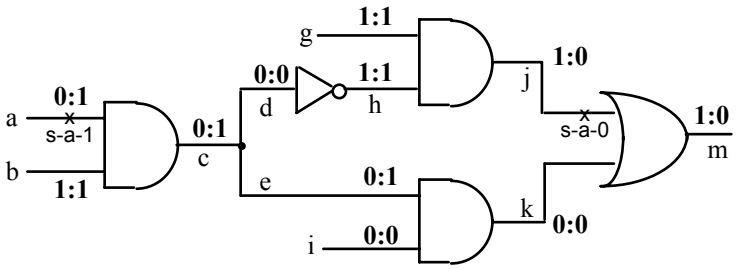

Fig. 2. Lines behavior in the presence of a frontier fault.

Each line carries a couple of values x:y, where x is the fault free value and y is the value in the CUT. Lines *d, g* and *h* are hidden, so their values are equal to the fault free ones. The faulty value on line *a* propagates to lines *c* and *e* but not to *d* and *h*. During analysis, we consider simultaneous presence of all possible faults -since their presence is potential- and their effects are then propagated through the circuit whenever they are activated by the current input vector.

III. FAULT ANALYSIS

Fault analysis assumes that the set of multiple faults to be covered is the set of all frontier faults and it is performed into two main phases: *propagation phase* and *backward phase.* Forward phase determines all possible values on the circuit lines. It takes into account any possibility of propagating faulty values due to any frontier fault not yet dropped. Backward phase starts from primary outputs assuming that the fault free response is observed and progresses toward primary inputs. For a given gate, it deduces values that are actually present on its inputs in order to observe fault free response only. From these deduced values, faults and fault effects may be dropped on lines and the backward phase continues on the driving network. Once a fault is dropped, it means that it cannot be component of any other frontier fault, and cannot reappear because we assume permanent faults. Therefore, a whole class of frontier faults is dropped at once because no fault masking can occur. (We say that a fault f_i^{α} is masked by a fault f_j^{β}, if the test vector that detects f_i^{α} does not detect the simultaneous occurence of f_i^{α} and f_j^{β}).Thus, the results obtained are not invalidated in the presence of redundant

and undetected faults: a fault is dropped when it is not masked by another fault or combination of faults and it is either observable or its site may be hidden in the CUT.

A. Line Model

Given an input vector, a line i propagates a fault effect if its value in the CUT is different from its fault free one due to some fault(s) in the driving network. To model the behavior of a line i in the CUT, we associate the following bit variables: n_i / p_i*, s_i^1, s_i^0:

• n_i designates the fault free value on line i.

• $s_i^1 = 1$ ($s_i^0 = 1$) if f_i^1 (f_i^0) is a component of a frontier fault still to be considered.

• p_i is called the propagation bit. It is equal to 1 if line i may be normal and may propagate a fault effect. It is equal to 0 if line i carries fault free value only, or it is hidden or faulty.

As a consequence of the definition of the propagation bit, fault effects are potential and may propagate on normal lines only. That is on hidden and faulty lines the propagation bit is always equal to 0 (hidden lines carry fault free values only). These properties are reflected through the propagation bit in our analysis algorithm.

B. Forward Phase

We assume that primary inputs are directly controlled from the circuit environment and that no fault effects propagate to them. Starting from primary inputs and proceeding in topological order toward primary outputs, we compute fault free values and propagate fault effects assuming that the output of the gates are normal, otherwise they will not propagate any fault effect by definition. Fault effects are assumed independent (no correlation between fault effects issued from the same fault or fanout stem), hence, the evaluation of line values is conservative and includes the behavior of the CUT. Thus, in computing the output value, we take into account the gate functionality and any possibility of faulty values on its inputs.

The propagation bit on the output of an AND/NAND gate is set to 1 (i.e., the gate propagates a fault effect) in the following cases:

i) All fault free values of its inputs are equal to 1 and at least one input i carries a fault effect, or

ii) For each input i with $n_i = 0$, either p_i or s_i^1 are equal to 1. Furthermore, if for each input i, $n_i = 0$, then there must be at least one input carrying a fault effect.

In case i) all inputs have fault free value of 1, and if there is a fault effect on one of them, say input i, it is assumed normal and the output is normal so the propagation bit is set to 1 on the output. In case ii) the fault free value on the output is 0, but if for each input i with $n_i = 0$ there is a possibility of a s-

* n_i / p_i has a different meaning from the D notation. $D = 1/0$ means that line i is equal to 1 in the fault free circuit and 0 in the CUT. n_i / $p_i = 1/1$ means that line i is equal to 1 in the fault free circuit and may be 0 or 1 in the CUT, and this effect may be observed on some primary outputs.

a-1 or a fault effect then the gate may propagate a faulty value on the output. If all inputs have a fault free value 0, we do not consider them simultaneously s-a-1. This can be summarized in the following formula:

$$p_{out} = 1 \Leftrightarrow [(\forall i, n_i = 1) \Leftrightarrow (\exists i, p_i = 1)] \Delta$$
$$[(\exists i, n_i = 0) \Leftrightarrow (\forall i, n_i = 0 \Leftrightarrow (p_i \Delta s_i^1 = 1)) \Leftrightarrow (\exists i, n_i \Delta p_i = 1)]$$

Fig. 3 illustrates two situations of the propagation bit computation for an AND gate. For each line i we represent its fault free value and propagation bit by n_i / p_i. Values in bold characters constitute one of the possible combinations that may propagate a fault effect on the gate output.

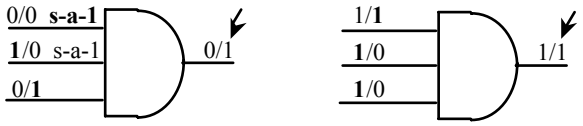

Fig. 3. Propagation bit computation for an AND gate.

Similarly, the propagation bit on the output of an OR/NOR gate can be computed as follows:

$$p_{out} = 1 \Leftrightarrow [(\forall i, n_i = 0) \Leftrightarrow (\exists i, p_i = 1)] \Delta$$
$$[(\exists i, n_i = 1) \Leftrightarrow (\forall i, n_i = 1 \Leftrightarrow (p_i \Delta s_i^0 = 1)) \Leftrightarrow (\exists i, \overline{n}_i \Delta p_i = 1)]$$

For a XOR or XNOR gate, a fault effect is propagated on the output on the output if there is at least a fault effect on one of its inputs; this will change the input value and then the output value too. Also, there is a fault effect on the output if the fault free value of one input is v with a possible stuck-at-$\overline{v}$ fault. For example, if the fault free value of an input is 1 and with a possible s-a-0 fault, combined with the fault free value of the other input will produce a different value from the fault free one at the output (i.e., propagates a fault effect at the output). This is computed as follow:

$$p_{out} = 1 \Leftrightarrow (\exists i, p_i = 1) \Delta (\exists i, (n_i = v \Leftrightarrow s_{i\overline{v}} = 1))$$

The propagation bit is transmitted as is from the input of an inverter to its output sine no faults are considered on its input. For a fanout stem, it is broadcast to all its branches. Fig. 4 illustrates an example of the computation of the propagation bit on each line given the input vector $\boldsymbol{t} = 000$. The s-a-1 fault on line 4 combined with the fault free value of line 5 propagates a fault effect on line 7 ($p_7 = 1$). There is no fault effect on line 8 ($p_8 = 0$) because we do not consider simultaneous s-a-1 (or s-a-0) on the inputs of any gate. The fault effect on line 7 combined with the s-a-0 or the fault free value on line 8 propagates a fault effect on line 9.

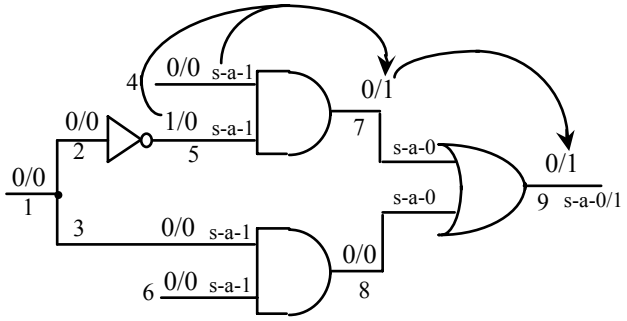

Fig. 4. Evaluation and fault effect propagation.

C. Backward Phase

The purpose of this phase is to eliminate faulty conditions that contradict the observed fault free response on primary outputs, or that may be hidden by other faulty conditions. The backward phase starts from primary outputs and progresses toward primary inputs. It assumes that fault free response is observed and drops fault effects on these outputs by resetting to 0 their propagation bits. Then, for each gate feeding the outputs, it deduces possible values that are actually carried by their inputs: The deductions may result in dropping fault effects and faults that are not masked and cannot be present in order to observe fault free response only.

A fault effect on a line is dropped by resetting to 0 its propagation bit, i.e., the line carries fault free value only. Each time the propagation bit on a line is reset to 0, the backward phase continues further in the driving network. This phase is not performed for any gate whose propagation bit on its output is not reset to 0. Depending on types of the encountered gates, propagation bits and status of its inputs are deduced according to its functionality and the following deduction lemma which determines, if possible, the unique input responsible for a fault effect on the gate output, and which is not be masked by any other fault effects. For an AND/NAND the deduction lemma can be stated as:

Lemma 2: 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 on each input i of the gate the fault effect and the fault:

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

Proof: Since p_g was reset to 0, then g could be in one of the following states:

- g is a faulty or a hidden line: All the inputs of the gate are hidden; therefore they carry fault free value only, i.e., $\forall i,\ p_i = 0$ and $s_i^1 = 0$.
- g is normal:
  - (i) $\forall i,\ n_i = 1$: A fault effect on any input i will propagate to the output g. Since $p_g = 0$ then $\forall i,\ p_i = 0$.
  - (ii) There is an input i such that $n_i = 0$ and $(\forall j \neq i,\ (n_j = 1$ and $p_j = 0))$: A fault effect on i will be the only cause of the fault effect on g. Since $p_g = 0$ then $p_i = 0$ and $s_i^1 = 0$.

QED

The corresponding lemmas for others types of gates are stated as follows (their proofs are similar to the previous one according to the gate functionalities):.

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 on each input i of the gate the fault effect and the fault:

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

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 on each input i of the gate the fault effect and the fault:

- Drop the fault effect: $p_i := 0 \Leftrightarrow (n_j = v) \wedge (p_j = 0) \wedge (s_j^{\bar{v}} = 0), j \neq i$
- Drop the fault: $s_i^v := 0 \Leftrightarrow (n_i = \bar{v}) \wedge (p_j = 0), j \neq i$

For inverters and buffers, if the propagation bit on the output is reset to zero, it is also reset to zero on the input. For a fanout stem, the conditions are stated in the following lemma:

Lemma 5: Let s be a fanout stem. Sufficient conditions for resetting the propagation bit p_s to 0 are:

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

Proof: We can have one of the following situations in the circuit under test:

- Let b be a branch of s, b is normal and $p_b = 0$ then obviously $p_s = 0$ otherwise p_b would necessarily have to be 1.
- For every branch b of s, $p_b = 0$:

  (i) All branches are hidden or faulty lines. The stem s is hidden and carries fault free value, therefore $p_s = 0$.

  (ii) At least one branch b is normal, then $p_s = 0$ otherwise p_b would necessarily have to be 1.

QED

Example: Fig. 5 shows an example of deductions made during the backward phase. If the observed fault free value on line 9 is 0, then the frontier fault (f_4^1) is detected, or line 4 is hidden by the undetected fault (f_7^0) or (f_9^0). The fault f_4^1 is thus dropped according to the deduction lemmas and implicitly all frontier faults containing f_4^1 as component are dropped too, i.e., the set $\{(f_4^1), (f_3^1, f_4^1), (f_4^1, f_6^1), (f_4^1, f_8^0)\}$.

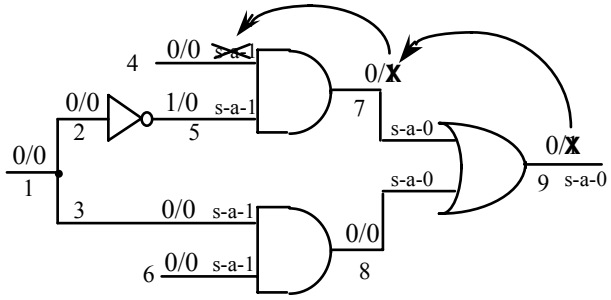

Fig. 5. Fault dropping example.

Each time a fault is dropped on a line, the backward phase determines in the same time the possible presence of a normal path from that line to a primary output in order to declare this line as normal. At the end of fault analysis, the used fault dropping technique will have determined four classes of lines which are summarized in Table I. Deduction 1 and Deduction 2 in the table designate the deductions made on a line, if any, in order to belong to the corresponding class.

TABLE I
FAULT ANALYSIS RESULTS

Class	Deduction 1	Deduction 2
Normal	Y	Y
Normal or Hidden	Y	N
Faulty or Normal	N	Y
Faulty, Normal or Hidden	N	N

Deduction 1: Fault dropped on the line.
Deduction 2: Normal path(s) from the line to a primary output.

D. Analysis Example

The circuit example presented in Fig. 6 makes the emphasis on the contribution of our analysis method compared to the existing ones in the literature. In Fig. 6a, the forward phase is performed for the input vector *adg* = 010. Faults (f_b^1), (f_h^1) and (f_i^1) may propagate fault effects on lines *k*, *l* and *m* respectively (i.e., $p_k = p_l = p_m = 1$). In Fig. 6b, the backward phase is applied to the circuit assuming that the fault free value 0 is observed on line *out*. Fault effects on *k*, *l* and *m* are reset to 0 by Lemma 3. According to our model, faults (f_b^1), (f_h^1) and (f_i^1) are dropped when the backward phase continues for the gates feeding lines *k*, *l* and *m*, respectively.

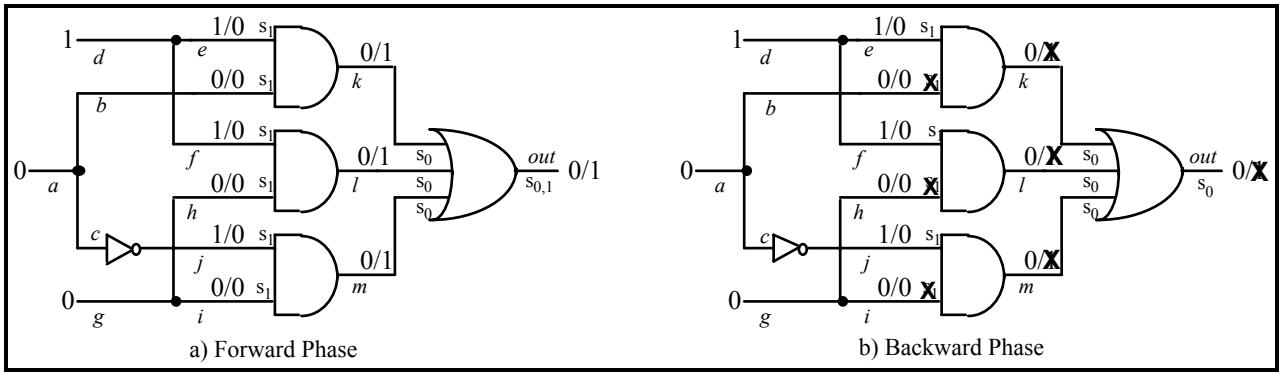

Fig. 6. Fault analysis example.

In this example circuit, the single fault f_l^0 is redundant and thus, no method can claim the presence or absence of (f_l^0) and would not drop the faults (f_f^1) and (f_h^1) since no normal path could be deduced from lines *f* and *h* to the primary output *out*. Using our analysis method, a complete test set will drop all faults in the circuit except (f_l^0). The faults of interest f_f^1 and f_h^1 are also dropped because either lines *f* and *h* are hidden by (f_l^0) if present in the CUT, or in the absence of this latter fault, their effects will be observable on the output.

E. Event Analysis

When testing our algorithm, we noted that it includes an inherent pessimism due to the use of single vectors in the presence of multiple faults and the conservative evaluation of line values, and that event analysis improves the performance of the method. The second step of our algorithm analyzes events which represent signal changes (0 to 1 or 1 to 0) between two consecutive vectors [1]. Lines that propagate such changes can be asserted as normal. Compared to the method presented in [6], our approach to event analysis does not perform explicit propagation and deduction of pairs of vectors through the network. After the analysis of each single vector, event analysis consists in retracing signal changes that have been propagated and observed on a primary output, by simply comparing the deduced values between the current and precedent input vectors.

We define an event on a line as a change that must have occurred on that line if the circuit were fault free. We also define a potential event as the possibility of having a change on a line due to remaining potential faults, i.e., when a change may propagate in the CUT but not in the fault free one. In order to simplify explanations, we consider 2-input gates; we designate them as the upper and the lower inputs. A generalization to *n*-input gates ($n > 2$) is immediate.

Fig. 7 illustrates different combinations of events and potential events. Notation *x-y* represents a change from value *x* to value *y* in the fault free circuit, and *x-* represents no change. In Fig. 7a, there are events with different polarities on both inputs, then there is no change in the fault free circuit but due to a s-a-1 or a fault effect on the lower input, there may be a change on the gate output in the CUT (potential event). Fig. 7b represents the propagation of a potential event. Fig. 7c represents the case where an event occurs regardless the state of the lower input. Fig. 7d represents the absorption

of an event due to the dominant value on the lower input (we assume that the s-a-1 fault has been dropped and that there is no fault effect on this input).

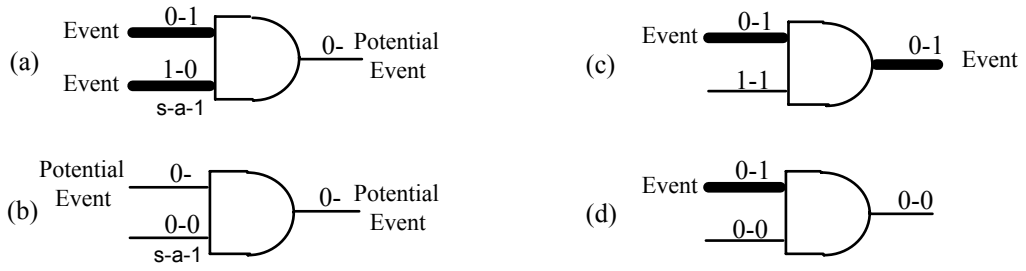

Fig. 7. Propagation of events and potential events.

The objective of the event analysis is to trace backward single paths along which events have occurred and declare all lines belonging to such paths as normal. Fig. 8a shows such cases: if we deduce an event on the output of an AND gate and it is only due to an event on the upper input, then this input must be normal, i.e., the presence of a fault would have been detected.

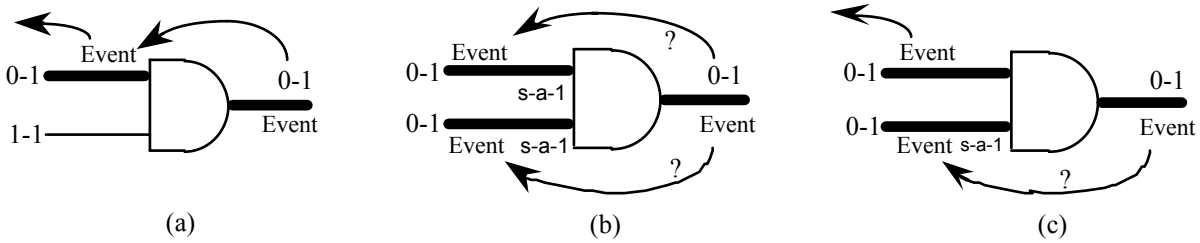

Fig. 8. Possible deductions in the presence of events.

The event on one input combined with an appropriate event or a stuck-at fault on the other input may produce a change on the output, as illustrated in Fig. 8b where two events are combined at the output of the gate; we are not sure if both events occurred or only one of them. In this case the event analysis is not resumed to pass through this gate. In Fig. 8c, since we have observed an event at the output of the gate, we can deduce the event on the upper input when no fault effect is possible during the first vector. For the lower input, two situations are possible: An event or a s-a-1 fault. These situations are not distinguishable because both of them, if present, when combined with the event on the upper input, can produce the event at the output of the gate.

When no deductions can be made on the inputs of a gate which propagate an event, we try to identify, if possible, the unique stem that feeds this gate and propagates an event too. If such stem exists, then we are sure that it is the unique responsible of the event observed on the gate output. Therefore, the fault effect that disables the event on this fanout stem is dropped and the analysis is resumed starting from this stem toward primary inputs. This is similar to the stem regions analysis [6, 10] which identifies reconvergent fanout stems and then correlates values emanating from them.

IV. EXPERIMENTAL RESULTS

The analysis program consists of approximately 2000 lines of code. The test vectors were randomly generated, except for the ALU 74181 where it is well known that there are 16 basic vectors that detect all multiple faults. For other circuits, we analyze random vectors, and a number of them were retained to constitute the test sets presented in Table II.

When dealing with multiple faults, different measures have been proposed for the evaluation of the fault coverage [7], and misleading conclusions on this coverage can be drawn. In our case, we adopt the measure presented in [7] as *unconditional fault coverage*, because it is more adequate: In a circuit of m lines, each line can be s-a-0 or s-a-1. These $2m$ faults are components of multiple faults. Therefore, the coverage is defined as the ratio of the number of dropped faults to the total number of faults. Note that this coverage can be used as a lower bound for the actual coverage since it is more pessimistic: A fault on a line is not dropped unless all frontier faults involving this line are dropped. For example, in the circuit in Fig. 6 consisting of 10 lines (excluding fanout stems), the initial number of faults is 20. After the analysis of the first vector (Fig. 6b), we dropped 13 faults out of the 20 possible ones (i.e., 65% fault coverage), and a complete test set will drop all faults except (f_l^0), producing 95% fault coverage.

Table II shows the results obtained from the analysis of the ISCAS'85 benchmark circuits [4]. The table gives the circuit name (C432 indicates that the circuit consists of 432 lines); the number of faults on lines after collapsing (components of frontier faults); the test size; the fault coverage as defined above; and the total CPU time to analyze the complete test set on a SPARC-Station 2. Note that the test size is the length of the input sequences analyzed by the algorithm. Some input vectors may have been repeated.

TABLE II
FAULT COVERAGE FOR ISCAS'85 CIRCUITS

Circuit	Faults	Test Size	Coverage	CPU Time
74181	175	22	100%	0.67 sec
C432	345	146	99.4%	8.2 sec
C499	640	250	99.1%	15.3 sec
C880	692	175	100%	23.6 sec
C1355	1056	601	84.4%	1 mn 50.8 sec
C1908	1109	523	96.6%	2 mn 16.3 sec
C2670	1839	639	88.2%	3 mn 45.5 sec
C3540	2270	1254	97.3%	10 mn 35.4 sec
C5315	3738	865	89.3%	11 mn 31.2 sec
C6288	4832	785	73.3%	226 mn 20.6 sec
C7552	4950	1561	84.8%	27 mn 30.8 sec

Since fault collapsing is first performed, we end up with less faults (second column of Table II) than methods considering faults on checkpoints, as in [6]. Thus, a less number of faults and the use of single vectors make our method efficient enough to achieve high fault coverage at reasonable amount of computing time.

For the ALU 74181, test sets generated for single stuck-at faults are not sufficient to cover all multiple faults [9]. In our case, there are 16 basic input vectors [11] that detect all multiple faults of

all multiplicities. These vectors were applied more than once to obtain a sequence of 78 vectors in [6], while in our case a sequence of 22 vectors suffices and it is analyzed in only 2.7 seconds compared to 14.66 sec (=0.188*78) in [6] (on the same frame, i.e., SUN-3/260).

The C880 circuit is testable for all multiple faults of all multiplicities. Our method were able to achieve a 100% coverage in only 23.6 seconds with a test set consisting of 175 single vectors. However, for the C6288 (a 16 bit multiplier which is time consuming due to its topology), a method of generating very high quality test is more suitable in order to achieve a better fault coverage in less computing time.

V. CONCLUSION

We have presented a new method to analyze multiple faults at the gate level in combinational circuits. The frontier fault model reduces the number of faults to consider during the analysis and improves the deduction power of the method. Faults that are dropped cannot be masked and are either observable on primary outputs or their sites are hidden by yet undetected faults. Early fault dropping avoids the need of backtracking in the analyzed input sequence in order to find (if any) a test vector for a given fault for which a normal path has been deduced during the analysis. When the analysis is completed, the remaining undetected faults in the circuit do not invalidate the obtained results.

The use of single vectors requires simpler algebraic manipulation and is less time consuming than pairs of vectors. Event analysis, which retraces normal paths that propagated events between two consecutive vectors, remedies to the inherent pessimism and loss of value correlations. Clusters of vectors can be useful in this case, but there is a trade off manipulating simple algebraic equations using single vectors against sets of bit sequences using clusters of vectors. The efficiency of clusters over single vectors is not undoubted when considering the combinatorial explosion of values using clusters of vectors. Experimental results show the speedup and efficiency when analyzing multiple stuck-at faults using our method, compared to results published in the literature.

We plan to extend our analysis method to handle diagnosis and we are currently developing an automatic test pattern generator for multiple faults based on the frontier fault model and the concepts of this analysis method.

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] 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.

[3] 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.

[4] 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,

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

[6] 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.

[7] 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.

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

[9] 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.

[10] F. Maamari, J. Rajski, "A Reconvergent Fanout Analysis for Efficient Exact-Fault Simulation of Combinational Circuits", *Proc. of the 18th Fault-Tolerant Computing Symp.*, 1988, pp. 122-127.

[11] J. Rajski, "GEMINI - A Logic System for Fault Diagnosis Based on Set Functions", Technical Report TR-87-5R, McGill University, 1987.

[12] 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.