# Incremental Diagnosis of Multiple Open-Interconnects

J. Brandon Liu, Andreas Veneris
University of Toronto, Department of ECE
Toronto, ON M5S 3G4, Canada
{liuji, veneris}@eecg.utoronto.ca

Hiroshi Takahashi
Ehime University, Department of CS
Matsuyama, Ehime 790-8577, JAPAN
takahashi@cs.ehime-u.ac.jp

## Abstract

*With increasing chip interconnect distances, open-interconnect is becoming an important defect. The main challenge with open-interconnects stems from its non-deterministic real-life behavior. In this work, we present an efficient diagnostic technique for multiple open-interconnects. The algorithm proceeds in two phases. During the first phase, potential solution sets are identified following a model-free incremental diagnosis methodology. Heuristics are devised to speed up this step and screen the solution space efficiently. In the second phase, a generalized fault simulation scheme enumerates all possible faulty behaviors for each solution from the first phase. We conduct experiments on combinational and full-scan sequential circuits with one, two and three open faults. The results are very encouraging.*

## 1. Introduction

Open-interconnect is becoming an important fault in today's advanced manufacturing process [17]. Current CMOS ICs may have six or more metal interconnect layers along with numerous vias and contacts. These are susceptible to open faults due to material defects or processing anomalies, electro-migration or thermal stress [5, 8, 17]. Furthermore, according to the SIA road map, the total length of interconnect wires grows exponentially, as shown in Figure 1. This also increases the likelihood of occurrence of this type of defects in a chip.

Modeling an open defect at logic level presents a challenge. In the case of finite resistance, an open can be modeled by a combination of transition faults. However, if the defect has an infinite resistance, it may cause itself and its fanout branches to float [4, 12]. As shown in Figure 2, the logic value on a stem with an open can be resolved differently at its fanout branches. This is due to the different threshold values of the gates at the fan-outs, which depend on various layout and physical parameters [5, 8], which are commonly not available to logic level diagnostic tools. Opens can also cause sequential behaviors by creating capacitive feedback paths [9]. We only investigate opens with infinite resistance in this work, as in [12, 17].

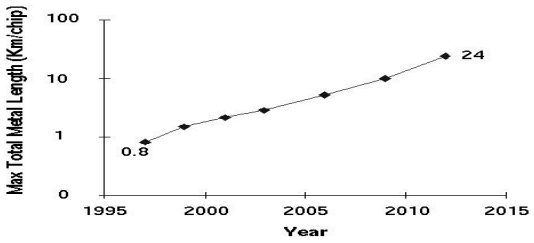


**Figure 1. SIA's National Technology road map on interconnect length (property of SIA).**

Consequently a cause-and-effect (e.g. building fault dictionary) approach is encumbered by the lack of a simple deterministic fault model. A mixed mode simulation scheme can circumvent the problem but it may result in higher costs [5, 8]. Recently, Venkataraman and Drummonds proposed a diagnostic algorithm for open-interconnect faults on logic level [17]. In that work, the authors introduce the notion of a net diagnostic model, which is a superset of all observable faults for a stem and its fanout. Their method then proceeds to build a composite output pattern for each open fault, which is used in matching the output response of the faulty chip.

In this work, we present a simulation-based model-free diagnosis algorithm for multiple open-interconnect faults. The algorithm proceeds in two phases. The **first phase** identifies tuples (pairs, triples, etc) of suspect faulty lines following an incremental diagnosis methodology. During *incremental diagnosis* [16], the algorithm iteratively identifies suspect faulty locations one at a time. For each location, it selects a suitable fault/error model, which brings the logic behavior of the circuit and its specification closer, and proceeds to the next iteration. Experiments show that, in practice, incremental diagnosis achieves nearly linear performance for multiple faults and errors.

In this paper, the logic unknown value [1] is introduced instead of a fault model. Logic unknown has been used in region-based diagnosis as an alternative to fault models [2, 14]. Unlike [2, 14], this work tailors the notion of logic unknown to an incremental framework in which faults in random error locations, not in regions, are identified. The **second phase** reduces the number of candidate fault tuples through a generalized fault-simulation scheme that enumerates the effects of an arbitrarily complex fault.

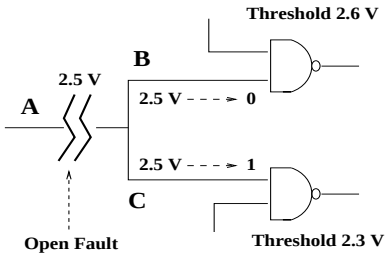


**Figure 2. Open fault on a stem.**

The main contributions of this paper are 1) devise a worst-case fault model for open-interconnect fault, 2) propose a model-independent algorithm for incremental diagnosis of multiple open faults, 3) develop theory and heuristics to capture open faults efficiently 4) evaluate the efficiency of the proposed method by experiments on ISCAS'85 and full-scan versions of the ISCAS'89 benchmark circuits.

The paper is organized as follows. We present the open fault model and the motivation for this work in Section 2. An overview of the algorithm is presented and illustrated with a simple example (Section 3). The subsequent two sections (4 and 5) describe the method and Section 6 contains results for ISCAS'85 and full-scan ISCAS'89 sequential circuits. Section 7 concludes this work.

## 2. Fault Model and Model-Free Diagnosis

In this work, the behavior of a chip corrupted with multiple open-interconnect faults is not explicitly emulated due to the lack of a suitable (deterministic) fault model. Instead, a worst case scenario is used: *make the logic value of the wire with an open fault behave completely at random*. The purpose of this fault model is not to emulate realistic behavior of open-fault but to generate a worst-case scenario for evaluating the diagnostic algorithm.

In this model, all fanout branches of a stem behave at random independent of the value on the stem or each other. As illustrated in Figure 2, if a stem has an open fault, its value can be interpreted differently by its branches even for the same vector [4, 5, 8]. This is equivalent to having all the branches floating. In implementation, we allow the branches take an arbitrary logic 1 or 0 for each test vector.

When an input test vector has at least one erroneous primary output (EPO), we say that a *symptom* is generated. It is seen, that this worst-case approximation covers all possible open symptoms. For simplicity, we refer to this approximation of an open-interconnect fault as *open fault* in the rest of our discussion.

In most logic-level diagnostic tools [7, 10, 11, 16, 18, 19, 20], explicit knowledge of fault models is necessary. In these instances, diagnosis is simplified in the sense that a fault model acts as an assumption to help narrow the possible candidates. For example, if a net is injected with a stuck-at fault, the cardinality of the fault model is 3 (i.e. SA-0, SA-1, and fault-free). The accuracy of such a diagnostic algorithm depends on the validity of the fault model. In the case of simple fault models, this approach is efficient and accurate. In the case of complex phenomena, such as bridging faults and design errors, it may even fail to capture some faulty effects [10, 16].

These observations motivate to find a *model-free* diagnosis algorithm. In this work, we demonstrate such an algorithm that identifies candidate locations *incrementally* [16]. It reasons with ternary logic values without any assumption about how a fault should behave or where it is located. We also evaluate its performance in presence of multiple open faults. In the future, we intend to investigate cases of multiple (stuck-at, open, bridge, etc) faults co-existent in a circuit.

## 3. Method Overview

We refer to the lines under consideration by diagnosis as a *suspect line*. Any set of lines returned by diagnosis is called a *candidate tuple*. Using this terminology, our diagnostic problem is formulated as follows. Given a logic netlist and a faulty chip, we seek a set of candidate tuples, any one of which may potentially account for the logic behavior of the corrupted implementation. Observe that a fault tuple can be classified as either *actual* or *equivalent* in the diagnosis context [15]. Obviously no diagnostic method can distinguish between them because both explain the fault effects. A test engineer needs to inspect the physical chip to find the actual cause of the fault(s).

The algorithm accepts a specification and a faulty chip, and produces a list of probing sites for test engineers. Figure 3 outlines its overall flow. The diagnosis proceeds in two independent phases which are described in Section 4 and 5. The **first** phase performs incremental diagnosis by simulating *logic unknown* values on single lines to capture all possible fault effects [2]. This phase also compacts the candidate tuples in classes to return a somewhat pessimistic list of candidate tuples that may explain the erroneous circuit behavior. The **second** phase takes this list and conducts a simulation of all combinations of logic values on the tuples. Their ability to change primary output values is used as a

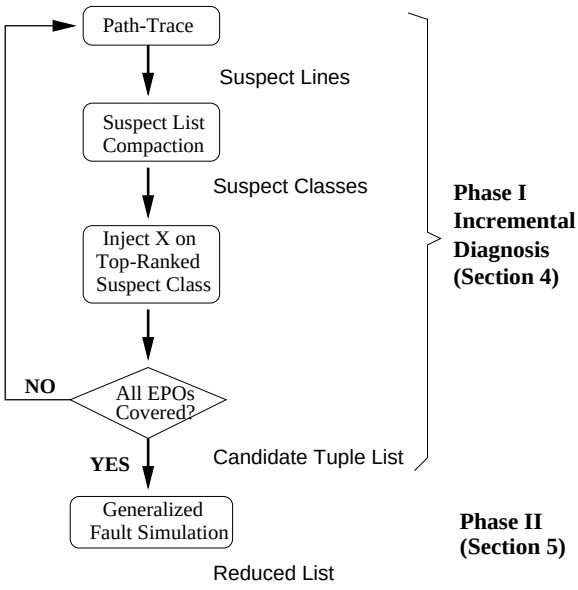


**Figure 3. Algorithm flow.**

criterion to reduce the length of the list. In the remainder of the section, we use an example to illustrate the conceptual approach. Throughout this example, a line is identified by the name of the gate that drives it. A branch is named by concatenating the name of its stem followed by the name of the gate it fan-ins.

*Example 1:* Figure 4 (a) contains a circuit with two open faults located on lines $G_8$ and $G_9$. In that figure, each line also carries its simulated fault-free/faulty value [1] for a single vector. In this example, it is seen that both primary outputs are erroneous. Recall that in fault diagnosis, faulty values are the "desired behavior" we try to capture by injecting faults at appropriate locations in our simulatable netlist. Apparently, the internal values of the faulty implementation are not visible to the algorithm.

The first phase of the diagnosis algorithm tries to identify the fault locations one at a time. Assume that this phase returns with location $G_9$. Next, the algorithm places a logic unknown on $G_9$ and simulates at its fanout cone, as shown in Figure 4(b). It can be seen that the unknown propagates to $G_{17}$ which is originally an EPO. At the second iteration of phase I (Figure 4 (c)), the algorithm ignores all lines with unknown value and identifies $G_8$ as the second potential fault location. Again, it places an unknown value $X$ on $G_8$ and simulates its fanout cone. Since every EPO has an unknown value on it, this concludes the first phase and $\{G_8, G_9\}$ is returned as a candidate tuple.

During the second phase, for each tuple returned by the first phase, the algorithm enumerates all the possible combination of binary logic values (0 and 1) for each vector simulated on all its member lines. In the case of a stem, its branches are considered as separate lines so that we cover all possible error effects. Intuitively, this exhaustive simulation captures all possible faulty effects that may arise from the underlying locations for each vector. In this example, the tuple $\{G_8, G_9\}$ is expanded to $\{G_8, G_{9\to 12}, G_{9\to 15}\}$. There are $2^3$ possible combination of logic values for this triple, only two of which are shown in Figure 4(d). The values in the upper square boxes are those produced by assigning the ordered tuple logic values $\{1, 1, 1\}$ and the lower ones are produced by logic values $\{1, 0, 0\}$. We observe that both combinations produce the faulty value on $O_1$ but only the lower one does on $O_2$. As long as a *single* combination produces the observed EPO behavior, phase II qualifies the tuple as a valid solution. Therefore, $\{G_8, G_9\}$ is the output of the algorithm.

## 4. Phase I: Incremental Diagnosis

The first phase of diagnosis identifies a set of candidate tuples so that if the logic unknown value is placed on every member line of the same tuple, all the EPOs will have the unknown value. In this phase, the algorithm finds the faults in each tuple iteratively. As illustrated in Figure 3, each pass consists of three steps. First, a *3-valued path-trace routine* marks suspect lines. Then a novel *suspect compaction scheme* reduces the suspect list by grouping lines with similar unknown simulation behavior into a single *class*. It also ranks these classes by using a matching formula at the primary outputs. Finally, the top ranking class is chosen and the unknown value $X$ is placed on one representative member of the class. Multiple passes are conducted until all the EPOs are covered by the unknown value. All suspect classes picked along one path are grouped together to form one candidate class tuple. Each of these tuples is a potential solution to the diagnosis problem.

To improve the run-time efficiency, the results of parallel vector simulation for every line in the circuit are stored in indexed arrays as in [15]. The $i$-th entry of this logic array for line $l$ contains the well-defined (0 or 1) logic value of $l$ when the $i$-th input vector is simulated or a logic unknown $X$ due to Phase I of diagnosis. This array is properly updated to reflect the effect of injecting and simulating unknown values. The following definition aids the remaining discussion.

**Definition** A *diagnostic configuration* is a partially diagnosed circuit during incremental diagnosis and consists of the circuit structure and the indexed array of logic values on each line. A diagnostic configuration that has at least one EPO remaining is called *active*. Otherwise, it is *inactive*.

### 4.1 3-valued Path-Trace

The 3-valued path-trace is an extension of the Critical Path Tracing [1] and Path-Trace [19] procedures. It starts

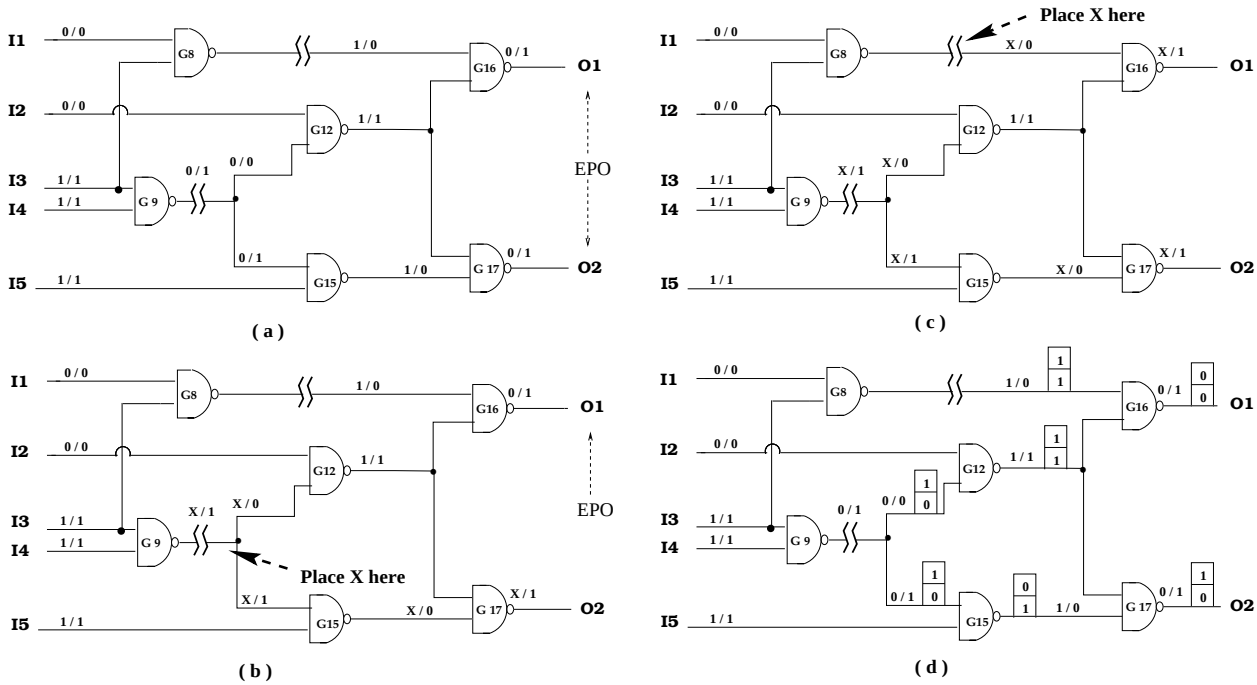


**Figure 4. Diagnosis of two open-interconnects.**

from an EPO and pessimistically marks lines that may belong to a sensitized path (i.e. a path of lines with different logic values under the influence of some fault(s)). If the output of a gate has been marked and the gate has one or more fan-in(s) with controlling values, then the procedure randomly marks one controlling fan-in; if the gate has all fan-ins with non-controlling inputs, then all fan–ins are marked; if a branch is marked, then it marks the stem of the branch. In the context of this work, this procedure is modified by adding the following rule to accomodate for logic unknowns:

**Rule:** Never mark a line with logic unknown value on it.

Path-trace may deduce information by following a trail of (possibly) erroneous values in the circuit. Due to the conservativeness of the unknown value [1, 3], lines with unknown values can *never* increases the number of EPOs. Therefore, a line with an $X$ can provide no information to path-trace.

The following two theorems prove that 3-valued path-trace is pessimistic enough to guarantee inclusion of all equivalent tuples.

**Theorem 1** *If the logic unknown value is simulated simultaneously on all lines with opens then all EPOs obtain the unknown value as well.*

**Proof.** Placing the unknown values on all fault-injected lines is equivalent to just placing the unknown on all these lines in a fault-free circuit. As such, the unknown value propagates identically in both circuits due to the absence of any fault effects. A fault free circuit has no EPO. Therefore, all EPOs are eliminated (by logic unknown) from the circuit. ◇

**Theorem 2** *If all the nodes of the search tree are visited, all equivalent fault tuples are included in the candidate list.*

**Proof.** It has been shown in [15] that the original path-trace algorithm marks at least one member of each equivalent solution tuple. The added rule does not affect its proof. Because placing the unknown value on a line annuls its fault propagation, any lines belonging to the fanout portion of its sensitized path cannot be marked in the subsequent path traces. Therefore, if the search tree is allowed to be fully explored, each level-by-level traversal monotonically reduces the number of faults in each equivalent tuple by one. Eventually, all the solution tuples are selected. By Theorem 1, they all qualify as candidate tuples. ◇

This path-trace algorithm is implemented to trace simultaneously from all EPOs generated by hundreds of vectors. For each line, we count the number of times it has been traced. Only the top ranked lines, specified by a use-defined parameter, qualify for subsequent procedures.

## 4.2 Suspect Compaction

Although fast and useful, the 3-valued path-trace has low resolution due to its conservative nature. To prune the

search space (tree) quickly without losing fault coverage, we devise a *suspect class compaction* method. Experiments show that this helps achieve dramatic reduction in search space and run-time when compared to brute-force. Suspect class compaction performs the following operations:

- collapse the suspect list (set of candidate circuit lines) of each node into fewer classes, which are used instead of individual lines in subsequent diagnosis.
- compute a score for each compacted class, which is a weighted sum of how many EPOs it can correct, how many previously correct primary output now contains logic unknown, and how frequently it is marked by the 3-valued Path-Trace.
- rank the classes by their score for selection during tree traversal.

Suspect compaction reduces the number of children for each node in the tree and the overall level of the tree. As seen in the experimental section, very few rounds are needed to capture all the injected faults. The suspect compaction does so by partitioning the suspect list into classes, each of which can be represented by one representative member line. The conceptual example that follows illustrates the motivation behind this approach.

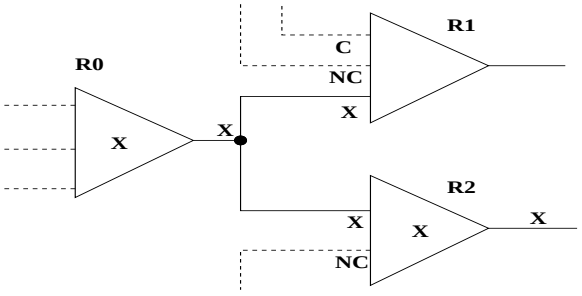


**Figure 5. Class compaction.**

*Example 3:* Each triangle in Figure 5 represents some fanout-free circuitry. The solid lines are those in R0's fanout cone; the dotted lines are inputs from other parts of the circuit. A logic unknown is placed on one of the lines inside R0 and propagated to its headline (i.e. the first stem that dominates R0 [1]). In R2, the logic unknown at its input propagates to its output because all the other fan-ins of R2 have non-controlling (NC) values. However, this is not the situation for R1 because one of its fan-ins has a controlling value (C). In this example, a logic unknown at R0 or a logic unknown at R2 or at both have the same effect at the primary outputs. We also observe that simulating a logic unknown at R0 introduces a superset of unknown values in the circuit. Therefore, simulating the propagating effect of each one separately seem wasteful. Rather, we can *compact* R0's and R2's fault effects together by placing a logic unknown in R0 and simulating it at its fan-out cone. This approach gives an upper bound estimation of the EPOs a line from R0 or R2 can rectify (i.e. change to logic unknown value).

Pseudocode for the compaction algorithm is found in Figure 6. Lines 10 through 12 of the algorithm give the compaction criterion: line $M$ is in the same class with $L$ only if logic unknown can always propagate from $L$ to $M$ for all test vectors. This condition enforces the requirement that simulating the fanout cone of $L$ can produce a upper bound of the number of EPOs turning into Xs for *any* line inside the class. Clearly a compaction class has to be associated with a particular diagnostic configuration.

When applied to Example 3, it groups R0 and R2 into one compacted class and recognizes some line from R0 as the class representative. The representative of the class has the useful property that simulating the unknown value on it makes the most EPOs turning to unknown that any other member in its class. Experiments show that diagnostic time is reduced at the cost of having a somewhat lower diagnostic resolution. However, as explained in Section 5, the resolution is regained in the second phase of the algorithm.

### 4.3 Search Tree

Because numerous suspect locations exist for each active diagnostic configuration, a *search tree* is built to facilitate data management similar to [16]. Each node (represented by an ellipse) of the tree represents a diagnostic configuration. For each active diagnostic configuration, a ranked list of candidate classes are compiled, as explained in Section 4.2. In the example of Figure 7, candidate classes are named by a single letter and placed inside the squares. The root node of the tree, $N0$, has 3 candidate classes: $A$, $B$ and $C$. Node $N4$ is inactive so it has no candidate classes. To go from a node to one of its branches, a candidate class is chosen (represented by a labeled arrow). A logic unknown value placed on its representative member and its fanout cone simulated to produce the new diagnostic configuration. Whenever an inactive configuration is reached, we call it a *successful leaf* (e.g. $N4$). An *unsuccessful leaf* occurs when the depth bound (user-defined at the start of the algorithm) is violated by a node with an active configuration. Therefore, the candidate classes identified along the path between the root and a successful leaf form a solution set. In the figure, $\{ A, E \}$ forms such a candidate tuple.

To guarantee we capture all faults in reasonable time, the search tree is traversed in a breadth/depth first manner simultaneously and it proceeds in *rounds* [16]. During each round, all nodes with active configurations are extended by one branch. For example, the root node ($N0$) is first produced in the first round in Figure 7. In the subsequent two rounds, $\{$N1$\}$ and $\{$N2, N3$\}$ are identified respectively. Node $N4$ is identified in the fourth round along with the

```
( 1) Compact(candidate list)
( 2) START
( 3) Sort the candidate list ascendingly by the level of each line.
( 4) Start from the beginning of the list, repeat for each line L of the list
( 5)   if L does not belong to any compacted class
( 6)      place X on L
( 7)      for all the vectors, simulate the fanout cone of L
( 8)      create a new compacted class C
( 9)      insert L into C
(10)      examine each line M in the candidate list after L
(11)         if M has X value for all test vectors
(12)             place M into C
(13)      set L to be the representative member of C
(14)      compile a output matching statistics and attach to C
(15)      restore the logic values in the circuit
(16) sort all the classes according to their matching score
(17) END
```

**Figure 6. Class compaction algorithm**

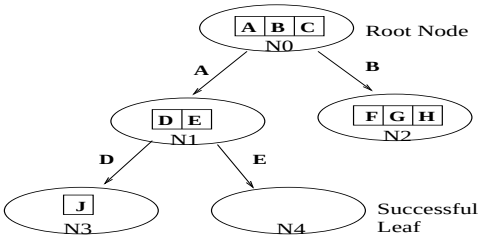


**Figure 7. Search tree.**

children for {N3} and {N2} (not shown in figure). At the end of the Phase I, all the successful leaves are examined to extract tuples of candidate classes. The Cartesian product of all lines in these classes is computed and this is the input to Generalized Fault Simulation in Phase II, described next.

## 5. Phase II: Generalized Fault Simulation

Phase I has the advantage of pruning the diagnostic space quickly. However, the list of candidate tuples it returns can be large due to the pessimistic nature of the underlying algorithms and the logic unknown. In Phase II, each of the the candidate tuples undergoes a full or partial enumeration of its values in a process called Generalized Fault Simulation (GFS). GFS is a generalization of the error simulation process developed in [15].

GFS emulates the effect of an arbitrarily complex fault on a line tuple. It accepts as inputs the circuit specification, a symptom and a list of candidate tuples, returned by Phase I in this case. As explained in Section 3, stems are replaced with their branches to comply to the worst-case fault model of Section 2. Next, GFS simulates all possible logic values on lines of the candidate tuples and for all test vectors, as explained below.

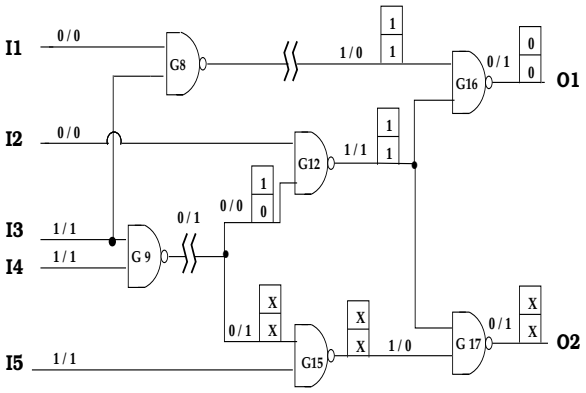


**Figure 8. Generalized fault simulation.**

Assume tuple with two lines $A$ and $B$. In the following discussion, letters are used to indicate both the line and its logic value for a test vector. GFS will simulate a total of four logic value combinations for each vector: $AB$, $\bar{A}B$, $A\bar{B}$, and $\bar{A}\bar{B}$ ($AB$ is included because neither errors are activated for some vectors). These combinations enumerate *all* the possible logic values any opens on $A$ and $B$ can produce. It can be shown [15], that for $n$ candidate lines, GFS requires $2^n$ simulations at the fan-out cones of the respective lines. A suspect tuple qualifies as a worst-case open fault candidate if it reproduces the observed behavior with at least one logic value combination. Parallel GFS is implemented so that many tuples are screened in each simulation pass. We omit implementation details which can be found in [15]

When the number of branches is large, it may be computationally expensive to simulate all applicable combinations of logic values. Instead, we place a logic unknown on a subset of them and perform the complete GFS on the remain-

| ckt name | wire count | # nodes visited | Phase I | | | Phase II | | | hit ratio |
|---|---|---|---|---|---|---|---|---|---|
| | | | # tuples | # probe sites | time(s) | # tuples | # probe sites | time(s) | |
| C432 | 412 | 9 | 36.5 | **6.3** | 0.94 | 16 | **6.3** | 0.078 | 100 |
| C499 | 1249 | 8 | 83.4 | **12.8** | 2.37 | 24.6 | **6** | 0.39 | 100 |
| C880 | 915 | 9 | 32 | **7.8** | 0.45 | 11.6 | **6.2** | 0.23 | 100 |
| C1355 | 1238 | 6 | 37 | **10.5** | 1.99 | 14.8 | **5.7** | 0.34 | 100 |
| C1908 | 859 | 8 | 77.6 | **13** | 1.5 | 10.5 | **6** | 0.21 | 100 |
| C2670 | 1377 | 9 | 41 | **8** | 3.95 | 15.3 | **8** | 0.40 | 100 |
| C3540 | 2282 | 11 | 47.9 | **5.5** | 17 | 22 | **5.5** | 0.81 | 100 |
| C5315 | 3697 | 10 | 37 | **8** | 5.1 | 13.0 | **3.5** | 1.47 | 100 |
| C6288 | 6319 | 13.2 | 67.0 | **16.5** | 25.2 | 6.5 | **3** | 2.24 | 70 |
| C7552 | 5262 | 6 | 65.3 | **12.7** | 7.54 | 22.8 | **9.4** | 2.22 | 100 |
| S838 | 404 | 9.7 | 30 | **4.6** | 0.13 | 21.5 | **4.6** | 0.16 | 100 |
| S1196 | 593 | 9 | 107.2 | **11.8** | 0.54 | 52.6 | **9.3** | 0.42 | 100 |
| S1494 | 625 | 11 | 42 | **7.1** | 1.1 | 16 | **7.1** | 0.44 | 100 |
| S9234 | 2339 | 9.65 | 26.8 | **5** | 3.11 | 16 | **5** | 1.89 | 100 |
| S38417 | 25585 | 10 | 21.8 | **12.4** | 57.1 | 11.4 | **11** | 9.7 | 100 |
| S38584 | 22447 | 10.5 | 14.3 | **6.3** | 36.3 | 5 | **5** | 17.2 | 100 |

**Table 1. Results for one open fault.**

ing branches. For example, in the circuit of Figure 8, an unknown is placed on $G_{9\rightarrow 15}$ and GFS is performed on $G_8$ and $G_{9\rightarrow 12}$. Again, we show the results of simulating only two fault excitation situations in that figure. When qualifying tuple $\{G_8, G_9\}$, we ignore output $O_2$, which has logic unknown on it. Only the other output needs to produce the faulty response.

## 6. Experimental Results

We run experiments on ISCAS'85 and on full-scan versions of ISCAS'89 circuits. Circuits are first optimized for area (`script.rugged`) using SIS [13]. For every circuit, we perform sixty experiments. In the first twenty a single open fault is randomly injected, in the next twenty two opens are injected and in the last twenty the circuit is corrupted with three opens. Most faults are injected on stems which is a *worst-case location scenario* for an open. In all experiments, diagnosis uses less than 1000 vectors, a combination of stuck-at fault vectors [6] and random vectors. Test vector generation for open faults is not the subject of this work [12]. Experiments are reported on a Sun Ultra 10 machine with 256 MB of memory.

During diagnosis, the search tree is not fully traversed. This is because suspect compaction places the candidates close to the top of the tree. Recall that the algorithm traverses the search tree in rounds. The more rounds the more coverage achieved. When conducting experiments, we observed that the first actual defect is discovered within 5 rounds. So rather arbitrarily, we execute 5, 10 and 15 rounds for circuits corrupted with one, two and three opens respectively.

Results for circuits with one open fault are in Table 1. The first column has the name of the benchmark and the second the number of lines it contains. The next column contains the average number of search tree nodes visited during diagnosis. Next, the average values for the number of candidate tuples, of probe sites and of time spent (in CPU seconds) are shown. The probe sites are the distinct locations a test engineer needs to examine. In most cases, Phase II reduces the number of candidate tuples by a factor of 3 or more, and the probe sites as well.

The last column contains the hit ratio to identify the actual fault. The algorithm identifies both actual and equivalent fault tuples. The injected faults are found with 100% success rate in almost all circuits within short time. The multiplier C6288 contains many fan-outs and reconvergences, which create an unfavorable environment for diagnosis. We find experimentally that it usually takes twice as many rounds to capture the injected faults than in other circuits.

The results for two and three open faults diagnosis are summarized in Table 2. We present the average number of probe sites, total CPU time and hit ratio (h.r.). The algorithm achieves 100% hit ratio in most cases while the number of probe sites remains small. These results demonstrate the flexibility of the proposed approach as it can capture most faults with good resolution (i.e. short probe list). It was also observed that the compaction heuristic reduces the width of the tree (*i.e.*, number of nodes per level) by one order of magnitude or more. Therefore, few rounds are needed to achieve high hit ratio.

| ckt | 2 Faults | | | 3 Faults | | |
|---|---|---|---|---|---|---|
| name | sites | time(s) | h.r. | sites | time(s) | h.r. |
| C432 | **15.6** | 7.9 | 100 | **17.4** | 42.1 | 100 |
| C499 | **30** | 30.8 | 100 | **47.8** | 354 | 100 |
| C880 | **10.4** | 7.9 | 100 | **15** | 191.8 | 97 |
| C1355 | **29.6** | 29.8 | 100 | **35.8** | 522 | 100 |
| C1908 | **19.6** | 18 | 95 | **22.6** | 276.3 | 100 |
| C2670 | **17.4** | 11.8 | 95 | **18.4** | 632 | 100 |
| C3540 | **14.8** | 94.8 | 90 | **21.6** | 490 | 90 |
| C5315 | **20.8** | 139.2 | 100 | **30.2** | 885 | 100 |
| C6288 | **7** | 201.6 | 80 | **20.5** | 1442 | 87 |
| C7552 | **17** | 60.5 | 100 | **22.2** | 898 | 93 |
| S838 | **12** | 2.8 | 95 | **19** | 32.67 | 90 |
| S1196 | **13.4** | 20.2 | 100 | **30** | 220.8 | 97 |
| S1494 | **16.2** | 35.9 | 100 | **30.7** | 346.4 | 100 |
| S9234 | **11.6** | 143.3 | 100 | **21** | 675 | 100 |
| S38417 | **18.3** | 413.8 | 100 | **11** | 1455 | 100 |
| S38584 | **14.3** | 465.7 | 100 | **8.3** | 1133 | 100 |

**Table 2. Results for two and three open faults.**

## 7. Conclusion

In this paper, we describe a model-free diagnostic algorithm for multiple open-interconnect faults. We use a worst-case model for the logic effect open faults may have on the circuit and argue that the independence from any explicit fault models is beneficial in dealing with physical defects that are difficult to model. The diagnostic algorithm is discussed in its two separate phases. The algorithm is applied to combinational and full-scan sequential circuits injected with one, two, or three open faults. The results exhibit the robustness of the method as it achieves good resolution within short computational time. In the future we plan to apply this approach to diagnosis of other types of faults as well as different types of faults present together.

## References


[1] M. Abramovici, M. A. Breuer, and A. D. Friedman. *Digital Systems Testing and Testable Design*. IEEE Press, Piscataway, New Jersey, 1994.

[2] V. Boppana and M. Fujita. Modeling the unknown!towards model-independent fault and error diagnosis. *Proc. ITC*, pages 1094–1101, 1998.

[3] M. L. Bushnell and V. D. Agrawal. *Essentials of Electronic Testing for Digital, Memory & Mixed-Signal VLSI Circuits*. Kluwer Academic Publishers, 2000.

[4] V. Champac, A. Rubio, and J. Figueras. Electrical model of the floating gate defect in cmos ics: Implication on iddq testing. *IEEE Trans. CAD*, pages 359–369, 1994.

[5] C. Di and J. A. G. Jess. On accurate modeling and efficient simulation of cmos opens. *Proc. IEEE ITC*, pages 875–882, 1993.

[6] I. Hamzaoglu and J. H. Patel. New techniques for deterministic test pattern generation. *Proc. IEEE VTS*, pages 138–148, 1998.

[7] S. Y. Huang and K. T. Cheng. Errortracer: Design error diagnosis based on fault simulation techniques. *IEEE Trans. CAD*, 18(9):1341–1352, September 1999.

[8] H. Konuk. Fault simulation of interconnect opens in digital cmos circuits. *Proc. IEEE ICCAD*, pages 548–554, 1997.

[9] H. Konuk and F. J. Ferguson. Oscillation and sequential behavior caused by opens in the routing of digital cmos circuits. *IEEE Trans. CAD*, 18:1200–1210, 1998.

[10] D. B. Lavo, T. Larrabee, and B. Chess. Beyond the byzantine generals: Unexpected behavior and bridging fault diagnosis. *Proc. IEEE ITC*, pages 611–619, 1996.

[11] J. B. Liu, A. Veneris, and M. S. Abadir. Efficient and exact diagnosis of multiple stuck-at faults. *Proc. IEEE LATW*, 2002.

[12] S. M. Reddy, H. Tang, I. Pomeranz, S. Kajihara, and K. Kinoshita. On testing of interconnect open defects in combinational logic circuits with stems of large fanout. *Proc. European Test Workshop*, pages 127–128, 2002.

[13] E. Sentovich, K. Singh, C. Moon, H. Savoj, R. Brayton, and A. Sangiovanni-Vincentelli. Sequential circuit design using synthesis and optimization. *Proc. Int'l Conference on Computer Design*, pages 328–333, 1992.

[14] N. Sridhar and M. S. Hsiao. On efficient error diagnosis of digital circuits. *Proc. IEEE ITC*, pages 678–687, 2001.

[15] A. Veneris and I. N. Hajj. Design error diagnosis and correction via test vector simulation. *IEEE Trans. CAD*, 18(12):1803–1816, December 1999.

[16] A. Veneris, J. B. Liu, M. Amiri, and M. S. Abadir. Incremental diagnosis and debugging of multiple faults and errors. *Proc. IEEE DATE*, pages 716–721, 2002.

[17] S. Venkataraman and S. B. Drummonds. A technique for logic fault diagnosis of interconnect open defects. *Proc. IEEE VTS*, pages 313–318, 2000.

[18] S. Venkataraman and S. B. Drummonds. Poirot: Applications of a logic fault diagnosis tool. *IEEE Design and Test of Computers*, pages 19–30, Jan.-Feb. 2001.

[19] S. Venkataraman and W. K. Fuchs. A deductive technique for diagnosis of bridging faults. *Proc. IEEE ICCAD*, pages 562–567, 1997.

[20] J. Wu and E. M. Rudnick. Bridge fault diagnosis using stuck-at fault simulation. *IEEE Trans. CAD*, 19(4):489–495, 2000.