Universal Test Generation Using Fault Tuples*

Rao Desineni Kumar N. Dwarkanath R. D. (Shawn) Blanton
Center for Electronic Design Automation,
ECE Department
Carnegie Mellon University, Pittsburgh, PA 15213
email: (rhd, nari, blanton)@ece.cmu.edu

Abstract

A test generation tool for combinational circuits called FATGEN has been developed based on the notion of fault tuples. FATGEN can be used to simultaneously generate tests for many types of misbehavior that occur in digital systems. Individual experiments involving SSL, transistor stuck-open, path delay and bridging faults for the ISCAS85 benchmark circuits reveal an average speedup of nearly 32% and test set compaction of 60% when faults of all types are analyzed simultaneously. In addition, there is an average reduction of approximately 34% in the number of aborted faults.

1 Introduction

The continued reduction of device sizes in integrated circuits will introduce several non-conventional, and yet unpredicted failure types. No single fault model can be used to accurately represent all such misbehaviors. Instead, several fault models must be used depending on the characteristics and technology of the particular design under test. Current approaches to test generation and fault simulation would mandate the use of different methodologies for each fault type as shown in Figure 1a.

The classical single stuck-line (SSL) fault model continues to be the most commonly used fault model in digital systems testing. However, defects in next-generation CMOS technology are not expected to behave like SSL faults. Therefore, test generation algorithms based on the SSL fault model alone will no longer be sufficient. Past work [1][2][3][4] has shown that tests aimed at SSL faults can and do detect real failures. However, other experiments indicate that some realistic failures cannot be completely modeled by SSL misbehavior [5]. In addition, recent experiments like [6] and [7] prove that SSL tests alone are not sufficient for obtaining high defect coverage. As a result, other fault models that accurately reflect realistic circuit failures, and test generation techniques that target specific fault types have been explored [8][9][10][11].

Other work has proposed the use of general fault models. In [12], the authors describe test tools that can analyze SSL, transition, bridging, and stuck-open fault types. In addition, conditional faults can be combined with other fault types, where conditional faults are additional signal line assignments that must be satisfied for fault detection. In [13], the properties of the input pattern (IP) fault model are explored. The IP fault model is a generalization of the well-known cellular fault model [14]. The IP fault model allows for both complete or partial functional verification of every circuit module, where a module is a well-defined combinational subcircuit described at any level of the design hierarchy. In [15], the authors describe a fault/error simulator. Their tool can analyze input pattern faults as well as design errors that include wire routing mistakes and gate omissions. Few commercial tools use fault models other than the SSL fault model. One notable exception is the comprehensive test analysis tool TestBench from IBM [16]. TestBench's pattern fault is a rarely used fault type that allows an arbitrary set of signal lines to be grouped into activation conditions for an error site. Pattern faults themselves can be combined together using "ANDing" and "ORing" relationships to form more complex fault types. Although many misbehaviors in function can be described using the techniques of previous work, only limited types of timing failures can be captured. However, our fault modeling mechanism, known as fault tuples [17], can describe arbitrary changes in both function and cycle timing enabling a generic test analysis tool as shown in Figure 1b.

Figure 1: (a) Current test analysis tools have a separate engine for each fault type. (b) Our approach uses a single engine to simultaneously analyze all types of faults.

In this paper, we describe a novel test generation methodology based on fault tuples. Fault tuples are sim-
ple 3-element conditions for a circuit-signal line, its value, and clock cycle constraints. AND-OR expressions of fault tuples, which are called macrofaults, can be used to represent (many) arbitrary misbehaviors in function and cycle timing. Fault tuples:

- allow both existing and emerging fault types to be represented using a single mechanism;
- reduces analysis time when several fault types are analyzed simultaneously;
- increases test set compaction when several fault types are analyzed simultaneously;
- and enables fault collapsing across various fault types.

Most of these advantages are demonstrated via simulation experiments involving the ISCAS85 benchmark circuits [18].

Our ATPG tool FATGEN (FAult Tuple test GENerator) is capable of generating test vectors for any arbitrary misbehavior modeled in the fault tuple format. FATGEN is based on the D-algorithm [19][20], and generates tests by simultaneously satisfying AND-OR expressions of fault tuples. It should be noted that FATGEN does not identify the fault types for a particular circuit; similar to bridging and path delay fault analysis, our approach assumes that macrofaults are provided as input.

As illustrated in Figure 1, our approach is generic in nature and is capable of simultaneously generating tests for macrofaults of various types. Macrofaults (irrespective of the type of defect they represent) that share an objective (AND expressions of fault tuples) are detectable by a common test vector. This characteristic is exploited to generate tests for different fault types simultaneously. Experiments performed on the ISCAS85 benchmark circuits indicate that a test generated for a fault of one type of misbehavior is typically a test for other types of misbehavior. Because fault tuples enable a common representation mechanism, fault collapsing across various fault types is possible. This leads to reduction in overall test set size and test analysis time when several fault types are analyzed simultaneously. The common representation also precludes the need to evolve new fault analysis tools for each new fault type.

The rest of the paper is organized as follows. In Section 2, the definition of the fault tuple is presented. This section also illustrates how various misbehaviors can be represented using fault tuples. The automatic test pattern generation methodology and algorithm are described in Section 3. In Section 4, we present our experimental methodology and the results obtained for the ISCAS85 benchmark circuits. Finally, in Section 5 we summarize our work and describe our current work.

2 Fault Tuples

A fault tuple \( f \) is a 3-tuple \(<l, v, t>\), where \( l \) is a circuit signal line, \( v \) is a value, and \( t \) is a clock cycle constraint. The value set for each element is:

\[
\begin{align*}
  l & \in \{ \text{lines} \} \\
  v & \in \{ 0, 1, D, D' \} \\
  t & \in \{ i, i+N, i-, etc. \}
\end{align*}
\]

A tuple \( f=<l, v, t> \) is said to be satisfied if and only if the signal line \( l \) is controlled to the value \( v \) in a clock cycle described by \( t \) and the corresponding error (if any) is propagated to an observable point. The tuple elements \( l \) and \( v \) and their value sets are well-known [19], however the element \( t \) needs elaboration. If \( v \in \{ 0, 1 \} \), then the signal \( l \) must be controlled to value \( v \) within the clock cycle range described by time element \( t \). For example, \( t=i \) means that the tuple is satisfied if \( l \) is controlled to \( v \) in any clock cycle \( i \). If \( v \in \{ D, D' \} \), then \( t \) describes the range of clock cycles where a signal line must be activated, and the resulting error (\( D \) or \( D' \)) manifests. Satisfaction of a tuple in this case also requires that the error be propagated to an observable point. Note, observation can take place in any clock cycle, that is, \( t \) describes the time for activation, not observation. The value \( t=i+N \), where \( N \) is some integer, means the tuple must be satisfied in the \( N \)th clock cycle after a reference clock cycle \( i \). Tuple satisfaction must occur in a clock cycle \( j \) before a reference clock cycle \( i \). Other values such as \( i, i+, \text{etc.} \) are similarly defined. The values assignable to the tuple element \( t \) for a particular fault tuple \( f \) are variables themselves that can be instantiated either by another fault tuple (discussed later) or set to a particular value \( N \) by the user.

The utility of fault tuples stems from their combination. A product \( P \) is defined as a conjunctive (AND) expression of tuples. For example, product \( P_1=<f_1><f_2> \) is an AND expression of tuples \(<f_1>\) and \(<f_2>\). A product \( P \) is satisfied if and only if all its tuples with \( v \in \{ 0, 1 \} \) are satisfied and one or more of its tuples with \( v \in \{ D, D' \} \) are satisfied. A macrofault \( M \) is a disjunctive (OR) expression of products, and therefore is an AND-OR expression of tuples. For example, macrofault \( M_1=\{<f_1><f_2>|<f_3><f_4>\} \) is an OR expression of the two product terms, \( P_1=<f_1><f_2> \) and \( P_2=<f_3><f_4> \), where the symbol ‘|’ is used to denote the OR relation between products. A macrout fault
M is detected if and only if one or more of its products are satisfied. Thus, M₁ is detected if and only if the product P₁ = p₁ < f₁,₂ > is satisfied or the product P₂ = p₂ < f₂,₄ > is satisfied, or both P₁ and P₂ are satisfied.

Fault tuples can be used to represent a significant number of digital misbehaviors. Consider an SSL macrofault M = a/l (i.e. signal line a is permanently stuck-at logic 1). Macrofault M can be represented using the single tuple {<a, D', i>}. This macrofault is detected if a good value 0 is applied to line a in any clock cycle i and the resulting error discrepancy D' is propagated to an observable point, which are the necessary and sufficient conditions for detecting the SSL macrofault M.

Other fault types can similarly be represented. The following examples highlight the representation capability of fault tuples:

- The MSL fault shown in Figure 2a can be detected in multiple ways as indicated by the disjunction of the tuple products. Specifically, the MSL fault M = (A/1, C/0) is detected if and only if both faults A/1 and C/0 are detected, or C/0 is detected with A=1, or A/1 is detected with C=0.

- The AND-bridging fault AND-NFBF (B, C) shown in Figure 2b is detected if and only if the SSL fault B/0 is detected with C=0, or the SSL fault C/0 is detected with B=0 [21]. The conditions for the detection of AND-BF (B, C) are easily described using the products listed in Figure 2b.

- The pattern fault 11 → (1, 0) affecting the OR gate shown in Figure 2c is detected if E and C are set to 1 and the discrepancy D appearing at the output of the OR gate is propagated to an observable point. The use of the same clock cycle variable τ = i for the tuples (Figure 2c) indicates that the tuples are not independent, that is, all the tuples of a product must be satisfied in the same clock cycle i for the fault to be detected.

- The macrofault representation of a transistor stuck-open (TSO) fault [11] is based on the 2-test sequence required for detection. A TSO fault is an example of a macrofault utilizing a relational value for the tuple element t. For the simple circuit of Figure 2d (assuming a full-CMOS transistor implementation), the pull-down transistor to which input B is connected is considered to be stuck-open. To detect this fault, the output of the NAND gate (line E) must be initialized to a logic 1 by the first test pattern. The second test pattern must activate the error discrepancy D', and D' must be propagated to an observable point. The detection requirements are captured by the time element values i and i+1. Tuples with t=i describe the initialization conditions for the macrofault, while tuples with t=i+1 describe conditions for activation and observation of the error discrepancy D' exactly one clock cycle after initialization conditions have been met in clock cycle i. Note that fault tuples within a macrofault are not limited to pairs of adjacent clock cycles. In other words, any combination of t values of i, i+1, i+2, i+3, etc., are allowed in a macrofault and are supported by FATGEN.

- Timing faults can also be easily captured by the fault tuple mechanism. Figure 2e illustrates a transition fault. This macrofault also uses relational values for the tuple element t. The slow-to-rise NAND gate transition fault E↑; (f) Slow-to-fall robust path delay fault ↓AEF.
tion is initialized \((E=0)\) by first test vector and the slow transition represented by the error discrepancy \(D\) on line \(E\) is propagated by the second test vector. This sequence of test vectors is captured by the time values \(i\) and \(i+1\).

- Path delay fault descriptions are similar to transition faults except require additional conditions for values on the side inputs along the path under test. The path delay fault shown in Figure 2f requires the sensitization of a slow-to-fall transition along the path shown. This fault is detected by a pair of test vectors. The first test vector must initialize the path, while the second initiates the transition and robustly sensitizes the path. Sometimes it is possible to optimize the tuple representation for a macrofault. For example, the path delay fault for the simple circuit of Figure 2f can use the single tuple \(<A, D', i+1>\) for the second clock cycle.

Figure 2 reveals that macrofaults share tuples, that is, many fault types can be described using common fault tuples. For example, the tuple \(<C, D, i>\) is present in the expression for the MSL and the AND-BF macrofault of Figures 2a and 2b. This common tuple information among various fault types can be exploited while performing test generation and fault simulation.

3 Test Generation for Fault Tuples

A test generation tool called FATGEN (FAult Tuple test GENerator) has been developed for generating tests for macrofaults described using fault tuples. A variation of D-algorithm is the core of the test generation procedure. FATGEN’s input is a circuit netlist description and a list of macrofaults in the tuple format; its output is a set of test patterns for the macrofaults and other information that includes coverage, runtime, etc.

Test generation for macrofaults described as fault tuples is accomplished by simultaneously satisfying all tuples in a product. If all tuples in one macrofault product cannot be satisfied simultaneously, the "next" product \((OR\) term) in the macrofault is tried. The test generation process continues until (a) all objectives in one macrofault product are satisfied, indicating that a test has been generated, or (b) all product terms in the macrofault are exhausted, in which case the macrofault is redundant. As an example, the circuit in Figure 2b shows an AND-bridging fault represented as \(\text{M}=<B, D, i><C, 0, i><B, 0, i><C, D, i>\). In this case, FATGEN will try to simultaneously apply a logic value 1 on signal line \(B\), a logic value 0 on signal line \(C\), and propagate the discrepancy \(D\) on signal line \(B\). If, due to circuit constraints, these assignments are not possible, FATGEN will try to satisfy the next product term \(<B, 0, i><C, D, i>\). The primary input assignment that satisfies these objectives is reported as a test vector for the macrofault \(\text{M}\).

In the rest of this section, we will describe the macrofault data structure (which is essential in explaining the test generation algorithm), the algorithm used in FATGEN, and our complete system (that includes fault simulation) for fault tuples.

3.1 Macrofault Data Structure

We use the macrofault data structure illustrated in Figure 3. Each product term in a macrofault is partitioned into sub-products based on the time elements or "clock cycles" of its tuples. The number of clock cycles for a product is simply equal to the number of different time element values appearing in the product. For example, the products of the MSL fault shown in Figure 2a has only one clock cycle, represented by the tuple element \(i\). On the other hand, the transition fault shown in Figure 2e and the path delay fault shown in Figure 2f, each has two clock cycles. As shown in Figure 3, the path delay fault has two clock cycles: \(i\) and \(i+1\). Both clock cycles each have three fault tuples, and are sub-products of the product \(P\).

![Figure 3: Macrofault data structure for the path delay fault of Figure 2f.](image)

3.2 FATGEN

Like all fault-oriented test generation algorithms, FATGEN consists of two major steps: justification and error propagation. The basic structure of FATGEN closely resembles the D-algorithm. However, several significant issues make FATGEN quite different from the classical D-algorithm. First, there may be more than one fault site, compared to the typical assumption of a
single site in the D-algorithm. Second, a macrofault product may not contain a discrepancy \( D, D' \), as in the case for a path delay fault. Finally, a test vector may have to satisfy multiple objectives simultaneously across both time (clock cycles), and space (different sites in the circuit).

Test generation within FATGEN starts by pre-processing the fault tuple list within a macrofault to determine if error propagation is required. If the fault tuples do not contain any discrepancies \( D \) or \( D' \), the algorithm does not enter the error propagation phase. Otherwise, error propagation is given priority over justification problems. The pseudo-code in Figure 4 outlines FATGEN; it is an alteration of the D-algorithm pseudo-code given in [21].

\[
\text{FATGEN}(S) \\
S: \text{sub-product} \\
\text{begin} \\
\text{set values on all lines in the tuples of the sub-product} \\
\text{if (signal line values conflict) return FAIL} \\
\text{if (tuple list contains D or D') then} \\
\text{begin} \\
\text{if (Imply_and_check() == FAIL) then return FAIL} \\
\text{if (error not at PO) then} \\
\text{begin} \\
\text{if (D-frontier == NULL) then return FAIL} \\
\text{repeat} \\
\text{begin} \\
\text{select an untried gate (G) from D-frontier} \\
\text{propagate error through G} \\
\text{if (D-alg() == SUCCESS) then return SUCCESS} \\
\text{until all gates in D-frontier have been tried} \\
\text{return FAIL} \\
\text{end} \\
\text{end} \\
\text{// an error has been propagated or no error in tuple list */} \\
\text{if (J-frontier == NULL) then return SUCCESS} \\
\text{select a gate (G) from J-frontier} \\
\text{repeat} \\
\text{begin} \\
\text{justify the value at the output of G} \\
\text{if (D-alg() == SUCCESS) then return SUCCESS} \\
\text{until all gates in J-frontier have been tried} \\
\text{return FAIL} \\
\text{end} \\
\text{end} \\
\text{end}
\]

Figure 4: Pseudo-code of FATGEN for test generation based on fault tuples.

\[
\text{TEST\_GENERATION\_FOR\_MACROFAULT}(M) \\
M: \text{macrofault} \\
\text{begin} \\
\text{while (all products P of M)} \\
\text{while (all sub-products S of P)} \\
\text{if (FATGEN(S) == FAIL) then} \\
\text{try next product P} \\
\text{else} \\
\text{/** FATGEN() == SUCCESS */} \\
\text{/* A test \( t, k \) found for the sub-product */} \\
\text{store test \( t, k \) */ For fault simulation later */} \\
\text{increment \( K */ \text{Count of sub-products */} \\
\text{try next sub-product \( S \) */} \\
\text{end while on \( S \) \\
\text{if \( (K == 0) \) then /* Test generation failed for product */} \\
\text{try next product \( P \) \\
\text{else} \\
\text{/** Test vector(s) found for product \( P \), \( \text{M} \) detected */} \\
\text{for \( K: \text{all tests for different sub-products of } \text{P} \) \\
\text{call \text{FATSIM}(t, k) */} \\
\text{end for on \( K \) \\
\text{return SUCCESS} \\
\text{end while on \( P \) */} \\
\text{/** No test vector(s) found; macrofault \( \text{M} \) is redundant. */} \\
\text{return FAIL} \\
\text{end} \\
\text{end}
\]

Figure 5: Complete test generation pseudo-code for a macrofault \( \text{M} \).

4 Experimental Results

FATGEN has been implemented in the C language in about 5000 lines of code. FATGEN combined with FATSIM was used to perform test generation on the ISCAS85 benchmark circuits using a 300 MHz SUN-SPARC workstation with 1GB of memory. In order to demonstrate feasibility of the proposed fault tuple test generation methodology, the following fault types were analyzed: SSL, AND-NFBF (non-feedback bridging faults), TSO, and path delay faults. An equivalent fault collapsed set of SSL faults, generated using the fault simulator HOPE [22], for each of the benchmark circuits was used. The TSO fault list was created using SOPRANO, a test generator for TSO faults [11]. A small subset of all possible AND non-feedback bridging faults was selected by randomly choosing pairs of signal lines. Slow-to-rise and slow-to-fall path delay faults were considered for a random subset of all possible paths.

Simulation experiments involving the ISCAS85 benchmark circuits were performed in two stages: (1) tests were generated for each fault type individually, and (2) tests were generated for all fault types simultaneously. The reduction in run time by performing test analysis on all fault types simultaneously over the sum of run times for test analysis for each fault type individ-
ually is reported as speedup. The reduction in test set size by performing test analysis on all fault types simultaneously over the sum of test set sizes obtained by performing test analysis for each fault type individually is reported as test compaction. Similarly, the reduction in the number of aborted faults by performing simultaneous test analysis is reported as fault abort reduction.

Table 1 lists the results for each fault type for the larger ISCAS85 benchmark circuits. For each fault type analyzed, the total number of faults are listed in the column labeled (1). Column (2) shows the number of faults that are determined to be redundant. To limit test pattern generation time, a backtrack limit of 400 was used to abort test generation for a particular macrofault. The number of aborted faults for each fault type is shown in column (3). The total CPU run times (in seconds) are shown in column (4). Column (5) shows the size of the final test set. The last five columns in Table 1 show the results obtained by analyzing all the fault types simultaneously. The speedup, test compaction, and fault abort reduction for each benchmark and the average values obtained by performing simultaneous analysis are listed in Table 2.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>SSL faults</th>
<th>AND-NFBB faults</th>
<th>TSO faults</th>
<th>Path delay faults</th>
<th>All fault types together</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total faults (1)</td>
<td>Red. (2)</td>
<td>Abt. (3)</td>
<td>Time (sec) (4)</td>
<td>Total faults (1)</td>
</tr>
<tr>
<td>c432</td>
<td>524</td>
<td>1</td>
<td>3</td>
<td>0.7</td>
<td>85</td>
</tr>
<tr>
<td>c499</td>
<td>758</td>
<td>8</td>
<td>35</td>
<td>2.6</td>
<td>77</td>
</tr>
<tr>
<td>c880</td>
<td>942</td>
<td>0</td>
<td>2</td>
<td>1.1</td>
<td>83</td>
</tr>
<tr>
<td>c1355</td>
<td>1574</td>
<td>8</td>
<td>138</td>
<td>10.8</td>
<td>115</td>
</tr>
<tr>
<td>c1908</td>
<td>1879</td>
<td>7</td>
<td>9</td>
<td>5.9</td>
<td>155</td>
</tr>
<tr>
<td>c2670</td>
<td>2747</td>
<td>97</td>
<td>24</td>
<td>10.8</td>
<td>253</td>
</tr>
<tr>
<td>c3540</td>
<td>3428</td>
<td>140</td>
<td>12</td>
<td>13.7</td>
<td>192</td>
</tr>
<tr>
<td>c5315</td>
<td>5350</td>
<td>59</td>
<td>0</td>
<td>11.9</td>
<td>165</td>
</tr>
<tr>
<td>c6288</td>
<td>7744</td>
<td>34</td>
<td>45</td>
<td>29.4</td>
<td>43</td>
</tr>
<tr>
<td>c7552</td>
<td>7550</td>
<td>62</td>
<td>74</td>
<td>30.5</td>
<td>279</td>
</tr>
</tbody>
</table>

Table 1: Simulation results for SSL, AND-NFBB, TSO, and path delay faults for the ISCAS85 benchmark circuits.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Sum of run time for various fault types</th>
<th>Run time for simultaneous test generation</th>
<th>Speedup (%)</th>
<th>Sum of test set sizes for various fault types</th>
<th>Test set size for simultaneous test generation</th>
<th>Test set compaction (%)</th>
<th>Total number of aborted faults for various fault types</th>
<th>Number of aborted faults for simultaneous test generation</th>
<th>Fault abort reduction (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>c432</td>
<td>2.3</td>
<td>1.7</td>
<td>33.3</td>
<td>234</td>
<td>88</td>
<td>62.4</td>
<td>10</td>
<td>9</td>
<td>10.0</td>
</tr>
<tr>
<td>c499</td>
<td>11.7</td>
<td>4.4</td>
<td>62.5</td>
<td>433</td>
<td>182</td>
<td>58.0</td>
<td>119</td>
<td>20</td>
<td>83.2</td>
</tr>
<tr>
<td>c1355</td>
<td>3.3</td>
<td>2.5</td>
<td>27.0</td>
<td>344</td>
<td>195</td>
<td>43.3</td>
<td>2</td>
<td>0</td>
<td>100.0</td>
</tr>
<tr>
<td>c1908</td>
<td>27.5</td>
<td>13.2</td>
<td>51.8</td>
<td>607</td>
<td>222</td>
<td>63.4</td>
<td>215</td>
<td>56</td>
<td>74.0</td>
</tr>
<tr>
<td>c2670</td>
<td>20.1</td>
<td>17.9</td>
<td>11.2</td>
<td>641</td>
<td>269</td>
<td>58.0</td>
<td>15</td>
<td>6</td>
<td>60.0</td>
</tr>
<tr>
<td>c3540</td>
<td>30.9</td>
<td>26.5</td>
<td>14.2</td>
<td>603</td>
<td>318</td>
<td>47.2</td>
<td>83</td>
<td>82</td>
<td>1.2</td>
</tr>
<tr>
<td>c5315</td>
<td>45.6</td>
<td>31.1</td>
<td>31.1</td>
<td>544</td>
<td>216</td>
<td>60.3</td>
<td>25</td>
<td>21</td>
<td>16.0</td>
</tr>
<tr>
<td>Average</td>
<td>38.8</td>
<td>26.5</td>
<td>31.7</td>
<td>491.9</td>
<td>206.3</td>
<td>59.7</td>
<td>100.9</td>
<td>44.0</td>
<td>34.1</td>
</tr>
</tbody>
</table>

Table 2: Speedup, test set reduction, and reduction in the number of aborted faults obtained by performing simultaneous test generation on all fault types.
Results in Table 1 illustrate that our ATPG system can be used as a generic tool for many different fault types. Table 2 indicates an average reduction of 60% in test set size when all faults are considered simultaneously. Simultaneous analysis also results in substantial reductions in the run times (nearly 32% on average) and the number of aborted faults (nearly 34%). From the abort reduction percentage, it is easy to see that fault coverage improves for most of the benchmark circuits; the only exceptions are c5315 and c7552. We believe these initial improvements result from two factors. First, a significant number of macrofaults (of both the same and different fault type) happen to share fault tuples. Thus, the analysis performed on a single tuple is actually amortized over several macrofaults. Second, the fault simulation of a generated vector is performed over all macrofaults instead of a smaller subset of only one type. We expect that the introduction of heuristics for backtracking and search space pruning would further reduce the test analysis time, test set size, and the number of aborted faults. Currently, no heuristics are used.

To analyze the potential impact of fault ordering on test analysis, we performed experiments using SSL, TSO, path delay, and AND non-feedback bridging faults. Specifically, we analyzed all $4! = 24$ fault orderings for the ISCAS85 benchmark circuits. The ordering that resulted in the “best” results (which are reported in Tables 1 and 2) occurred for a fault list that analyzed AND-NFBBFs first, followed by path delay, SSL and TSO faults. The worst-case performance occurred for the ordering: path delay, TSO, AND-bridging, followed by SSL faults, which resulted in an average speedup of only 14.2%, and a test set size reduction of only 43.0%. This coarse-grain analysis reveals that there exists a large amount of fault dominance and/or equivalence across the macrofault types. The development of heuristics that exploit these relationships at the macrofault and tuple levels should further improve the results reported here.

5 Conclusions

High defect coverage requires good coverage of different fault types. In this paper, we presented a comprehensive test generation technique that can be used to generate tests for many arbitrary misbehaviors that can occur in digital systems, thus providing a single test generation solution. The technique is based on the fault tuple modeling mechanism which can be used to represent many misbehaviors in terms of simple 3-element primitives.

Experimental results show the viability of the methodology for generating tests for various fault types. It is also shown how common information among various fault types can be exploited to reduce the total test generation time, test set size, and the number of aborted faults. Results generated from the benchmark circuits produced a reduction in CPU run times for test generation by an average of 32% when compared to the sum of the CPU run times for each individual fault type. A significant reduction in the test set sizes and number of aborted faults is also achieved. Simultaneous analysis for various fault types resulted in an average test set compaction of 60% and a 34% reduction in the number of aborted faults.

Current work focuses on optimization and performance enhancement of the test and fault simulation algorithms. Techniques for dynamic fault tuple ordering and fault collapsing are being explored. In addition, we are adding capability for handling sequential circuits, and using the ATPG system for hierarchical testing.

References


