2000

Exploiting Don't Cares to Enhance Functional Tests

Mark W. Weiss
University of Nebraska-Lincoln, mweiss@cse.unl.edu

Sharad C. Seth
University of Nebraska-Lincoln, seth@cse.unl.edu

Shashank K. Mehta
Pune University, skmehta@cse.iitk.ac.in

Kent L. Einspahr
Concordia University, Seward, NE, Kent.Einspahr@cune.edu

EXPLOITING DON'T CARES TO ENHANCE FUNCTIONAL TESTS

Mark W. Weiss and *Sharad C. Seth*
University of Nebraska-Lincoln
Lincoln, NE 68588-0115 USA
{mweiss,seth}@cse.unl.edu

Shashank K. Mehta
Pune University
Pune, 411007 India
skm@cs.unipune.ernet.in

Kent L. Einspahr
Concordia University
Seward, NE 68434 USA
eins@seward.cune.edu

Abstract

In simulation based design verification, deterministic or pseudo-random tests are used to check functional correctness of a design. In this paper we present a technique generating tests by specifying the don't care inputs in the functional specifications so as to improve their coverage of both design errors and manufacturing faults. The don't cares are chosen to maximize sensitization of signals in the circuit. The tests generated in this way require only a fraction of pseudo-exhaustive test patterns to achieve a high multiplicity of fault coverage.

1. Introduction

Design verification techniques range from conventional simulation to formal proofs [1]. In simulation based methods, tests are applied to an implementation and the results are checked against the design specification to uncover any design errors. As with program testing, simulation can only reveal the presence of errors but not prove their absence. Formal methods, on the other hand, attempt to provide such assurance by mathematical proofs. The proofs may show the equivalence of an implementation and its specification [2], or verify certain properties that must be satisfied by any implementation [3]. In between these two extremes are the several semi-formal methods, such as symbolic simulation and partial model checking [1], [4].

This paper focuses on simulation based testing, often used in practice to verify large design entities, such as microprocessors. The tests (or simulation vectors) may be produced by the designer to verify the basic functions, with possible assistance from a program to cover exceptional conditions ("corner cases") [5]. Automatic methods may use random test generation and produce a very large number of tests that can now be simulated/emulated on high-speed workstations or specialized hardware.

Real design errors can be quite complex and hard to capture accurately in an abstract model. Nevertheless, several design-error coverage metrics have been proposed to evaluate tests. These include HDL-based measures to indicate coverage of statements, branches, paths, and tags by the test [6], the FSM-based measures showing coverage of states, transitions, and outputs; and design-fault models at the gate level [7]. The coverage metrics also provide opportunities for targeted test generation [8], [9].

Common to all verification tests, independent of how they are produced, is that each test component can be viewed as an input/output (or i/o) pair. The input part represents the stimuli to be applied for verification; the output part represents the reference against which the output of the circuit is checked for equality. As a result, the test is a sequence or collection of test components. These i/o pairs may have been derived from a specification model [8], [10] or may correspond to the specification itself, e.g. the "cubes" for combinational logic.

A test component may contain don't-care inputs (which can independently be set to arbitrary values without affecting the specified output) and don't-care outputs whose values are not functionally significant for the given input. For example, in a finite state machine a particular input event may force a state transition independent of the value of other inputs. Similarly, a test component for the carry function (of a full adder) might specify the input = 11X and the output = 1 to indicate that the output is independent of the third input component. From a functional point of view, the manner in which the don't-care input is set is indeed of no consequence, however, this can be significant in testing. As illustrated in Figure 1, when the don't-care is set to 0 there are two paths sensitized from inputs to outputs (shown as bold lines) but when it is set to 1 no paths are sensitized. Therefore, the input = 110 covers all the faults covered by the two vectors included in 11X.

In this simple example, the designer (or design tool) can choose one of four options: (a) arbitrarily fill in the don't care and produce one simulation vector, (b) do a three-value simulation of the implementation for the given input, (c) expand the don't care and produce two

0-7803-6546-1/00 $10.00

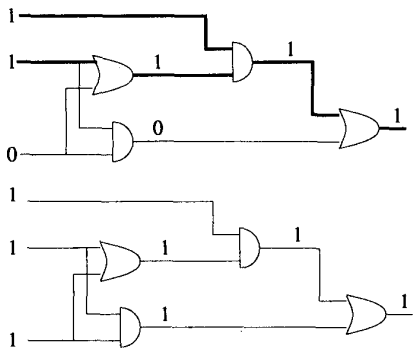


Fig. 1. Utilization of don't care inputs during testing.

simulation vectors, or (d) do a symbolic simulation. The last two options are logically equivalent. They do not scale up well when the number of don't-cares is large. Option (b) is computationally efficient but does not guarantee exact results because of the loss of information in three-value simulation. In option (a), on the other hand, an incorrect choice of the don't-care value can reduce the ability of the vector to detect manufacturing faults. It might diminish the coverage of design faults, as well, since the fault coverage metric is also used as a viable measure of design fault coverage [7].

The rest of the paper is organized as follows. Generalizing from the simple example above, a formulation of the problem addressed in this paper appears in Section 2. Here we propose a unified fault model that can account for design errors and manufacturing faults within the context of i/o pairs. Section 3 addresses the test generation issues related to this fault model. Experimental results on benchmark circuits are given in Section 4. Section 5 relates our approach to earlier work in the area of test generation. Section 6 concludes the paper.

2. Problem Formulation

2.1 A Unified Fault Model

Test generation for detecting functional faults due to design errors or manufacturing defects requires a comprehensive fault model which combines both. Traditionally, stuck-at faults and bridging faults are considered a useful and effective representation of manufacturing defects. Similarly, design errors are modeled by representing the designed circuit as a good circuit (which functions as per the specifications) with the possibility of a localized error such as wrong-gate-substitution, missing-gate, extra-input, missing input, etc., [7].

The two models can be unified by describing the faulty circuit $\mathcal{G}'$ as the good circuit $\mathcal{G}$ with a collection of pairs $\mathcal{C} = \{(S_1, E_{S_1}), (S_2, E_{S_2}), \ldots, (S_k, E_{S_k})\}$, where each S_i is a collection of lines of the circuit $\mathcal{G}$ and E_{S_i} is the corresponding *environment* condition, or formally, $\mathcal{G}' = (\mathcal{G}, \mathcal{C})$. The behavior of this circuit is interpreted as follows:

> Whenever the (good) circuit $\mathcal{G}$ satisfies the condition E_S on some input, the lines of set S in $\mathcal{G}'$ have complementary values compared to their respective values in $\mathcal{G}$.

To understand this model let us consider some examples. The stuck-at-1 fault at line g can be expressed by $S = \{g\}$ and $E_S = \{g = 0\}$. A bridging fault in which line-a is forced to 1 by line b, which is itself at 1, is expressed by $S = \{a\}$ and $E_S = \{ab = 01\}$. The effect of substitution of an AND gate by an OR gate at site G can be captured by $S = \{g\}$ with $E_S = \{h_1h_2 = 10, h_1h_2 = 01\}$, where h_1 and h_2 are the inputs and g is the output of G. Most other design errors can be captured by this model.

This fault model is useful for testing manufactured chips because the correct circuit is available. By using this fault model a test set can be generated and applied to each chip. On the other hand, in the case of design verification it is possible that the circuit implementation might be faulty (nonconforming to the specifications). The design can be verified by treating the designed circuit as "good" and the specification as the behavioral description of the "bad" circuit because the fault model is reversible. Stated in simpler terms, if a "bad" circuit results from "good" by substituting an AND gate for an OR gate, then the "good" circuit results from the "bad" circuit by substituting an OR gate for an AND. The reverse role of the good and the bad circuits is valid because the objective of testing is only to distinguish between the two. A test set can be generated based on the designed circuit and the fault model. Then the response of the circuit can be compared with the specifications.

To test a fault S, two conditions should be satisfied:

> (1) At least one of the conditions of E_S should be realized to *excite* the fault, and
>
> (2) The situation should be created such that the faulty signal from at least one of the S lines is *propagated* to some primary output.

2.2 Testing Functional Behavior

Consider the 4-input circuit driving a 7-segment digit display. Its specification has 10 input/output pairs. Testing the complete functionality of the circuit requires one to try each of the 10 inputs and compare the 7-bit output with the specification. Such a specification leaves no choice to the test-generator to improve the speed of testing. This is a situation where all 10 (valid) input patterns have distinct output patterns. Fortunately, in most large circuits the output patterns are much fewer than the valid input patterns. This is expressed by forming cubes in the input space and assigning one output pattern to each cube. In other words, such specifications have partially specified inputs (having 0, 1, and X). For example, the functional specification of a 4-input priority-encoder can be expressed by 4 i/o pairs, namely, $1000/00, X100/01, XX10/10, XXX1/11$ instead of 15 pairs..

In these cases a test-generator can optimize the test set by selecting a subset of the vectors of each cube with the same fault testability as the entire cube. For example, in the priority-encoder described above, it may not be necessary to test all eight inputs embedded in XXX1 if say, 1001 and 0011 could test all the faults that could possibly be tested by the vectors of XXX1. In this case the eight vectors in a functional test-set could be replaced by the two vectors (1001 and 0011).

In this paper we assume that the circuit under test has a small number of i/o pairs specifying its functionality compared to the valid inputs ($O(2^{no.-of-inputs})$) and most inputs in the pairs are partially specified (i.e., have X's). The test-generation process proposed here takes each i/o pair independently and computes a small number of fully specified input vectors from the cube which can detect all or most of the faults that could be detected by all the vectors of the cube collectively.

In a functional specification where inputs are partially specified the values of the X inputs are not relevant for the output in the good circuit. But these values do affect the output values in a defective circuit. It is thus necessary that the unspecified PI values are set in such a way that in a faulty circuit the output values differ from those in the good circuit.

For some partially specified input i we can classify the faults in three classes:

C_N: the faults that cannot be propagated by any setting of X's,
C_A: the faults that are propagated by all settings of X's, and
C_P: the faults that are propagated by some but not all settings of X's.

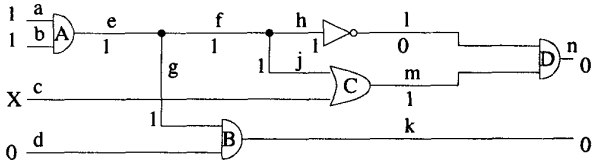


Fig. 2. Illustration of C classes.

For example, in Figure 2 under the input cube $\langle 11X0 \rangle$ $C_A = \{\{n\}, \{k\}, \{l\}, \{h\}, \{d\}, \{n,k\}, \ldots\}$, $C_N = \{\{m\}, \{j\}, \{g\}, \{l,m\}, \{g,d\}, \ldots\}$, and $C_P = \{\{f\}, \{e\}, \{a\}, \{b\}, \{a,b\}, \{e,d\}, \ldots\}$.

We only need to be concerned with the C_P faults in setting the unspecified PI values. Next, we attempt to determine the relationship between the faults and input settings that allow them to be tested.

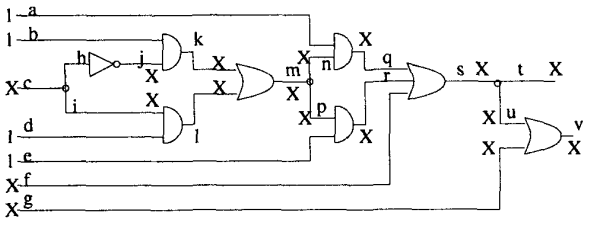
2.3 Border of the X-Domain

Consider the exact 3-valued simulation of some partially specified input i (see Figure 3). Simulation is called exact if a line has a binary value if and only if it is constant for all vectors of i, otherwise it is X. The values assigned to the X-inputs influence the binary values at the gates where X-values and binary values converge at the input and the output is a binary value. Any signal that does not pass through these gates cannot be affected by the values assigned to X's. Therefore in the faults of class C_P some of the faulty signals must enter these gates. If the X values are chosen so that they do not allow these signals to pass through these gates, then the fault cannot be detected.

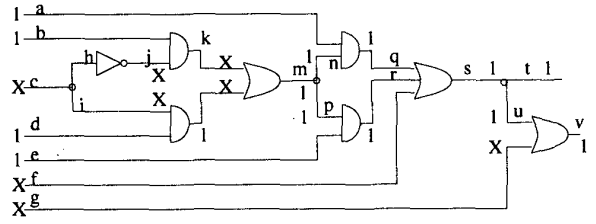
We formally define a *border*-gate as the gate which has a binary output and at least one X input in an exact 3-valued simulation. For input cube $\langle 11X0 \rangle$ in the circuit of Figure 2 the only border gate is C. A border gate is *enabled* if the values of the unspecified (i.e., X) primary inputs are set so that all the X inputs of the border-gate are set to the non-dominating value.

2.4 Test Generation Strategy

The difficulty of generating complete tests for such a general fault model is that considering each set of lines, S, and each value-assignment set, E_S, is computationally unviable. This situation requires judicious approximations. In our approach we consider the problems of excitation and propagation separately. Addressing first the propagation issue, we simplify the problem by restricting S to the singleton set only. The question that needs to be answered in order to generate an efficient test set is



(a)



(b)

Fig. 3. 3-valued vs. exact simulation

Given an input cube i, what is the minimum collection of singleton sets, C_i, with the property that any test set facilitating the propagation of all faults $S \in C_i$ also propagates all faults $S' \in C_P$?

Once we find C_i, a test set $\mathcal{T}_i$ is computed to propagate the faults of C_i. This test ensures propagation of all faults of C_P and if $\mathcal{T}_i$ is non-empty then it also takes care of C_A. Faults C_N do not have to be considered because they are not detectable by any vector of the cube i. If $\mathcal{T}_i$ turns out to be empty (i.e., when C_P would be empty), then any randomly selected vector of i is included in it to take care of C_A. The final test is $\cup_i \mathcal{T}_i$.

Fortunately it is easy to determine C_i from border-gate analysis. If a fault $S = \{l\}$ is in C_i, there exists a setting of unspecified PIs which enables the propagation of the fault from l to some PO(s), and there also exists a setting which blocks the propagation. Thus, there must exist a sensitization path starting from l and entering at least one border-gate. The sensitization path either (i) passes through no fanout-stem and enters input line l' of a border-gate, or (ii) it passes through a fanout-stem and the first such stem is l''.

In case (i) the fault l can be observed only if fault l' can be observed. In case (ii) the fault l can be observed only if the fault l'' can be observed because the sensitization path did not fork before entering l''. This fact leads to the conclusion that it is sufficient to consider faults at the fanout-stems and at the dominating inputs of the border-gates.

Observation *For any input-cube i, the singleton faults C_i that cover all faults of C_P of cube i in propagation is the union of the set of fanout stems in the input-cones of border-gates and the set of dominating inputs to the border-gates.*

For the example circuit of Figure 2 $C_{\langle 11X0 \rangle}$ is $\{\{e\}, \{f\}, \{j\}\}$.

Finally we turn to the fault excitation problem. As observed earlier, in general, any set of assignments could trigger the fault at the line under consideration. Thus, we may be left with no choice but to consider all value assignments of non-specified primary inputs, leaving no room for test set optimization. Consequently, it is required that each singleton fault be considered only in a few environmental conditions. The binary settings of lines are constant for vectors of the same input-cube. Therefore in our experiments we have considered each fault of C_i only once for test generation for each input cube. But if the same fault occurs in C_i and C_j, then the test is generated for it in both cases. Significant savings can be achieved by considering only C_i, as the example in Figure 3 illustrates. Of the eight possible test vectors in the input cube, two vectors are sufficient to test all singleton faults.

3. Test generation

As an illustration of the strategy just outlined, we discuss in detail the test generation process for the single-line (singleton) faults. The circuit and the i/o pairs are assumed to be available. The process independently considers every i/o pair and carries out the following steps in sequence:

(a) logic simulation of the input cube to justify each specified output,

(b) identification of border gates by analyzing the results of the logic simulation,

(c) generation of a list of target singleton faults corresponding to the collection C_i discussed above,

(d) generation of tests under input constraints to cover all target faults, and

(e) compaction of the tests generated in the previous step.

These steps are further elaborated below.

3.1 Logic Simulation

As is well known, the commonly used 3-value simulation trades information loss for speed (linear-time complexity); it may set some signal values as X that should be binary. In an exact 3-value simulation every signal that is binary (zero or one) independent of

the settings of the unspecified primary inputs should be correctly marked. Exact simulation is NP-complete since the boolean satisfiability problem can be reduced to it. However, this theoretical complexity can be encapsulated in a line justification procedure that is commonly used in automatic test pattern generation [11].

Let us assume that for a node (line) N in the circuit, Justify(N, v) determines if there is an input vector contained in the input cube that would set node N to the binary value v. For each node N with a X value after 3-value simulation, if the call to Justify$(N, 0)$ fails we can immediately change the X value to 1 because it is not possible to justify a 0 value at node N by *any* setting of the unspecified inputs. Otherwise, we make the call Justify$(N, 1)$. If this fails, the node can be set to 0, otherwise, it must remain as X. Since the number of X values is bounded by the circuit size, at most a linear number of calls to Justify is necessary for the exact simulation.

This idea is incorporated in the exact logic simulation algorithm shown in Figure 4. After the 3-value simulation, the algorithm collects all gate output nodes with X value that are in the cone of the specified outputs. These are tested for a constant value as above in order of their level from input to output. Whenever a node value changes, deterministic implications of the change are propagated to other nodes in the circuit and the list of remaining X nodes is pruned accordingly. In the final step, the algorithm checks for any discrepancies in the values at a primary output between the specification and exact simulation. If this happens a design error is detected independent of the settings of X values on the input.

Example: The circuit shown in Figure 3 will be used as a running example. For the input cube shown in the figure, assume both outputs are specified to be 1. Figure 3(a) shows the signal values after the (inexact) 3-value simulation upon which the following sorted list L will be created:

$$L = k, l, m, q, r, s,$$

It is possible to justify both 0 and 1 on k. Therefore this node retains its X value. The same is true of node l. However, $Justify(m, 0)$ fails therefore m is assigned constant 1 and by deterministic implication, lines n, p, q, r, s, t, u, and v are also assigned 1. As a result, the list L is pruned and becomes null, completing the while loop. The result is shown in Figure 3(b). The primary-output check in the last step succeeds as the PO values after exact simulation match the specification, hence no design errors are revealed at this stage.

```
Logic_Simulate(C:circuit, B:input cube) {

  3_value_simulate(C,B);

  For all the gates in the cone of specified outputs {
    Create a list L of gate output nodes with X value
    sorted in order of their level from input to output
  }

  While L is non-empty{

    Remove node N at the head of the list L

    If ¬Justify(N, 0) then {
      Assign 1 to N;
      Carry out deterministic implications and update L;
    }

    Else If ¬Justify(N, 1) then {
      Assign 0 to N;
      Carry out deterministic implications and update L;
    }
  }

  For each primary output Z with specified value v {
    If ¬Justify(Z, v̄) then report design error
  }
}
```

Fig. 4. Algorithm: A high-level description of the logic simulation algorithm.

3.2 Border Gate Identification

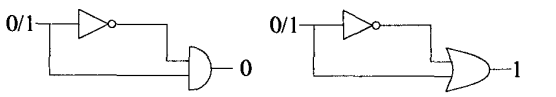
Based on the results of simulation, the gates with constant output and at least one X input are identified as border gates. By definition, the constant output is the dominating value for the gate. For the example in Figure 3(b) three border gates are identified, namely, the OR gates with output lines m, s, and v.

3.3 Fault List Generation

As discussed in Section 2, it is enough to consider the faults in border gates and certain fanout stems. The specific faults are determined as follows:

(a) For a border gate, if there is just one dominating input, say, p with value v, then we include the fault $S = \{p\}$ and $E_S = \{p = v\}$ (i.e., $p_{\overline{v}}$ in stuck-at model notation). If there is more than one dominating input, we need not include any faults at the border gate because no singleton fault can be propagated through the gate. If there are no dominating inputs, we must have X inputs that are correlated to produce a constant value at the gate output. In this case, we include the fault $S = \{p\}$ and $E_S = p_{\overline{v}}$ for each X input p.

(b) For each constant fanout stem s (with binary value v) that is in the input cone of a dominating value in a border gate, we include the fault $S = \{s\}$ and $E_S = p_{\overline{v}}$.



(a) Line Constrained to 0 (b) Line Constrained to 1

Fig. 5. Structural change to constrain input value.

For the three border gates in the running example, the following faults will be included: k_0, l_0, q_0, r_0, and u_0. In addition, because the constant-valued stem m is in the input cone of q and r, the fault m_0 will also be in the fault list.

3.4 Constrained Test Generation

The test generation must be carried out under input constraints; only the unspecified values in the input cube can be changed during test generation. It is possible to modify a PODEM-like algorithm that searches for a solution on a decision tree to allow branching and backtracking only on the unconstrained inputs. We accomplish the same goal by running a standard test generator [12] on a modified circuit that constrains the inputs internally (see Figure 5). A greedy approach is used to cover as many faults as possible by a single test vector before considering another vector in the input cube.

For the running example, the fault k_0 is detected by the test cube $abcdefg = 110110X$ which also detects m_0. Further expanding the test cube to 1101100 detects the fault u_0. Similarly, the test 1111100 detects l_0 and also detects m_0 and u_0. The faults q_0 and r_0 on the fault list are not detectable by any vector in the original input cube. Therefore only two vectors in the input cube cover all the faults detectable by all eight vectors included in the cube. There are 12 such faults: $b_0, d_0, h_1, i_0, j_0, k_0, l_0, m_0, s_0, t_0, u_0, v_0$.

3.5 Test Compaction

Most available ATPG tools provide the ability to compact the generated test set, e.g., by reverse fault simulation. The test vectors produced for an i/o pair may be compacted further by using this facility. In the running example no further compaction of the generated test set is possible.

4. Experimental Results

We implemented the test generation described in the last section and conducted experiments using a representative sample of 30 industrial PLA circuits included in the release of the Espresso tool [13]. In this section, we describe the experimental process and present the results.

As previously described, our approach requires both a structural representation and a behavioral representation for the circuit under test. Our experiment begins with the generation of these necessary specifications. The behavioral specifications were produced using the Espresso tool to generate the on-set and the off-set from the original PLA definition. The structural representations were produced using SIS [13] to simplify and synthesize the circuits from the original PLA definition. The synthesis step used the *rugged script*. Technology mapping was limited to four-input simple gates in a BDNET format. Finally, we convert the BDNET format to ISCAS-89 netlist format for use as the structural specification.

Table I provides the characteristics of the 30 circuits and summarizes the test generation results. Following the name of the circuit, the next three columns, from left to right, show the number of primary inputs, the number of primary outputs, and the number of i/o pairs in the minimized behavioral specification obtained using Espresso. The column labeled "Avg. Fully Specified Vectors per I/O Pair" gives the average size of a pseudo-exhaustive test for an i/o pair. The entries in this column were computed as follows. First, we determined the *relevant inputs* of each i/o pair as those primary inputs which occur in the cones of influence of the specified outputs and counted the number of don't care inputs, say m, among them. Then, 2^m is the length of an exhaustive test for the given i/o pair. The fifth column, labeled "Avg. Effective DCs per I/O Pair" is the logarithm to the base 2 of column 6. The eighth column, labeled "Total Tests" shows the number of tests obtained with our algorithm to cover all of the functionally non-redundant faults. The same number is shown normalized in column 7 labeled "Avg. Tests per I/O Pair". This can be compared with the number in column 6. Finally, the last column shows the inverse of the fraction of pseudo-exhaustive patterns used in our test, i.e., avg-fully-spec-vector-per-io/Avg-tests-per-io.

The average number of tests per i/o cube for all 30 circuits is just 4.41. This can be compared with the corresponding number, $7.81E+09$, for the fully specified vectors per i/o pair. We note that even in cases where the average number of don't cares per i/o pair is very large, e.g. the circuits *xparc* and *ibm*, the number of tests per i/o pair is still quite small.

We observed in Section 2 that because each i/o pair is considered independently for test generation, a given line in the circuit is likely to be observed multiple times with differently specified inputs. This was verified for the circuit *chkn* as follows. The circuit was fault simulated for the 2411 unique tests generated by

TABLE I
RESULTS FOR PLA CIRCUITS.

PLA Name	# PIs	# POs	# I/O Pairs	Avg. Eff. DCs per I/O Pair	Avg. Fully Spec. Vectors per I/O Pair	Avg. Tests per I/O Pair	Total Tests	Reduction
mish	94	43	158	3.52	1.1E+01	0.92	145	1.1E+01
misg	56	23	120	11.24	2.4E+03	1.13	136	2.1E+03
ibm	48	17	499	37.73	2.3E+11	3.77	1882	6.1E+10
misj	35	14	55	9.20	5.9E+02	1.15	63	5.1E+02
xparc	39	73	3226	30.98	2.1E+09	11.97	38609	1.7E+08
jbp	35	57	402	13.74	1.4E+04	5.14	2065	2.7E+03
x6dn	38	5	310	31.01	2.2E+09	4.66	1446	4.7E+08
in3	34	29	341	23.47	1.2E+07	4.39	1497	2.7E+06
in6	33	23	317	20.17	1.2E+06	4.35	1380	2.7E+05
b3	32	20	621	24.56	2.5E+07	6.0	3763	4.1E+06
b4	33	23	680	20.50	1.5E+06	4.29	29116	3.4E+05
in4	32	20	603	24.66	2.6E+07	6.25	3767	4.1E+06
exep	29	63	643	22.48	5.8E+06	12.69	8157	4.5E+05
in7	26	10	142	16.16	7.3E+04	2.77	394	2.6E+04
chkn	29	7	370	19.85	9.5E+05	6.88	2545	1.3E+05
vtxl	27	6	305	21.83	3.7E+06	3.12	953	1.1E+06
xldn	27	6	305	21.83	3.7E+06	3.12	953	1.1E+06
x9dn	27	7	315	21.69	3.4E+06	3.00	945	1.1E+06
in5	24	14	348	15.61	5.0E+04	4.87	1696	1.0E+04
vg2	25	8	304	19.69	8.5E+05	2.67	812	3.1E+05
t1	21	23	210	7.12	1.4E+02	3.12	656	4.4E+01
ts10	22	16	262	14.58	2.5E+04	4.10	1074	6.0E+03
shift	19	16	200	6.67	1.0E+02	4.86	971	2.0E+01
bc0	21	11	688	13.25	9.8E+03	7.83	5389	1.2E+03
in2	19	10	399	13.73	1.4E+04	5.25	2094	2.6E+03
t2	17	16	180	8.82	4.5E+02	3.14	565	1.4E+02
al2	16	46	141	7.19	1.5E+02	1.82	257	8.2E+01
alcom	15	38	90	3.30	9.9E+00	1.42	128	6.9E+00
b12	15	9	72	6.03	6.6E+01	2.17	156	3.1E+01
bcd	16	38	1590	12.05	4.2E+03	5.55	8824	7.5E+02

our method; no faults were dropped during simulation so we could count how many times each individual fault was detected by the tests. Next, for comparison purposes, random tests of equal length were generated as follows. For each i/o pair, the don't care inputs were filled randomly as many times as the number of tests produced by our method for that i/o pair and the resulting vectors were accumulated. A duplicate vector was detected as it was produced and replaced by another randomly-generated vector. This process ensured that the random tests were functional and of equal length for each i/o pair and overall.

The results are shown in Figure 6 for the two kinds of tests. Along the x-axis is the *detectability* of a fault for each test, defined as the number of times the fault is detected by the test. This number is normalized to lie between zero and one by dividing it by the common test length. The y-axis shows $F(x)$, the fraction of the faults with detectability greater than or equal to the value indicated by the x-value. Both curves start at (0,1). However, in between, they exhibit markedly different behavior, particularly, in the initial parts which correspond to faults with low detectability. Here, the tests generated by the border-gate approach are seen to catch low-detectability faults much more frequently than the random tests.

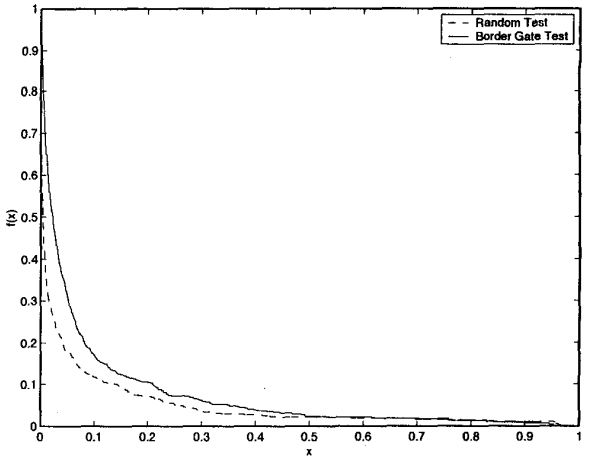


Fig. 6. Detectability of border-gate vs. random tests.

5. Related Prior Work

The key idea of our paper, setting input don't cares to maximize path sensitization in the circuit, is closely

related to earlier papers on automatic test pattern generation for manufacturing faults.

RAPS (Random Path Sensitization) [14] and SMART [15] have a similar goal of generating tests that deliberately sensitize a large number of signal paths towards the primary outputs (POs) without targeting specific faults. Unlike this paper, however, they assume no primary input constraints.

The RAPS test generator repeatedly computes one new test vector by executing the following steps to generate a test set which is better than a random set, by analyzing the circuit structure. To generate a test vector, it iteratively sets one unspecified (X) PO to a randomly selected binary value and justifies. At this point, if the circuit still has gate inputs with the X value at this point, one is randomly selected, set to the non-controlling value (relative to the gate driven by it) and justified. Then the next iteration begins if at least one unspecified PO remains.

SMART is an extension of RAPS. The difference between RAPS and SMART comes from exploiting "restart-gates" to extend the critical paths. Consider the circuit simulation of a partially specified input. A gate is called a "restart-gate" if it has one controlling input, its output is critical, and none of its inputs are critical. This can happen only if some of the inputs to the gate are unspecified and the output is specified. Thus, restart-gates are border-gates but the converse is not true. For example, gate C in Figure 2 is a border-gate but not a restart-gate because its output is not critical. RAPS takes every gate with unspecified inputs and attempts to justify the corresponding non-controlling value on such lines.

SMART, like RAPS, proceeds in an iterative fashion. It selects one of the unspecified POs and sets it to a randomly selected value (actually it discriminates between 0 and 1 when information from preprocessing suggests that one of them has a better chance to detect faults). Subsequently, it justifies that value. At this stage it adds a feature which is not present in RAPS. It takes one restart gate at a time and attempts to justify the non-controlling value on all the unspecified inputs of the gate. The success in justification leads to extension of the critical path to the controlling input of the gate. After all restart gates are considered the next iteration is started. The process terminates if no additional justification is possible.

The approach presented in this paper is similar to the SMART approach in using border (restart) gates to help extend sensitized paths. The main difference is that SMART ignores multi-branch sensitization paths, which appear more frequently in larger and more complex designs. The multiple branches may pass through the same gate when gates have more than one controlling input so such cases cannot be ignored. Further, treating one restart gate individually independent of the others cannot handle the sensitized paths with branches in different restart gates.

Another similarity between PODEM-X [16], FAST, and our algorithm is "dynamic compaction" which is referred to in this paper as "constrained test generation". The idea behind it is to set the X-bit(s) of the PI(s) of a test-vector i (which has some unspecified bits) so that more faults might be detected. Since only the X bits are changed to binary bits, the new (more specified) test vector j detects all the faults that were detected by i. FAST uses the same approach as SMART and generates j by justifying non-controlling values on the unspecified inputs of restart gates. On the other hand, in the present approach we start from i (input of the i/o pair) and attempt to generate as few fully specified vectors from i as possible which can collectively test all faults of $\mathcal{C}_i$. These test vectors are generated from i by a test generator which is restricted from changing the values of the signals with binary values. Unlike our approach (of finding relevant faults from border-gate analysis) RAPS and SMART generate test vectors from the initial input cube by extending the sensitization paths (generally without consideration to order and completeness).

Finally, we mention an earlier work with a complementary focus, namely, determining the maximum areas of *desensitization* for a given input vector. This problem is important when fault simulation is done by injecting single faults in the good circuit. The fast fault simulator (FFSIM) in PODEM-X uses a technique, called X-propagate [17] that allows rapid evaluation of gates which are unobservable. The X-algorithm [18] does the same using a more sophisticated approach.

6. Conclusion

The fault model and the border-gate approach to test generation allows a unified perspective on tests to detect design errors and manufacturing faults. Test generation is carried out independently for each i/o pair with the goal of covering all the faults in the chosen model that can be detected by *any* input vector in the input cube. The number of such vectors depends on the number of don't cares in the input cube. As the results on the benchmark circuits show, this number can be very large, yet, the same fault coverage is achievable with only a small subset of these vectors.

As the tests are independently generated for each i/o pair, each line in the circuit is likely to be observed multiple times in the context of different input settings. Recent research shows that the multiple observations

improve the coverage of non-modeled faults in manufacturing testing [19] We have found that typically the test set size increased by a factor of 3 to 30 over that generated by ATPG. However, because our tests are functional they can be applied at a much higher clock rate.

The multiple observations should also improve the coverage of design error faults. We intend to verify this for our test vectors by means of a new simulator [20] which is able to evaluate coverage of both design error and manufacturing faults.

Acknowledgments: This work was supported by the NSF Grant No. CCR-9971167 and the University of Nebraska-Lincoln Center for Communication and Information Science. We are grateful to Miron Abramovici for bringing some past related work to our attention.

References

[1] D. Dill, "Embedded tutorial: What's between simulation and formal verification," in *Proceedings of the Design Automation Conference*, June 1998.

[2] H. Hulgaard, P. F. Williams, and H. Reif, "Equivalence checking of combinational circuits using boolean expression diagrams," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, vol. 18, pp. 903–917, July 1999.

[3] J. Burch, E. Clarke, D. Long, K. McMillan, and D. Dill, "Symbolic model checking for sequential circuit verification," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, vol. 13, pp. 401–424, April 1994.

[4] C. W. Barrett, D. L. Dill, and J. R. Levitt, "A decision procedure for bit-vector arithmetic," in *Proceedings Design Automation Conference*, pp. 522–527, 1998.

[5] A. Chandra, V. Iyengar, D. Jameson, R. Jawalekar, I. Nair, B. Rosen, M. Mullen, J. Yoon, R. Armoni, D. Geist, and Y. Wolfsthal, "AVPGEN–a test generator for architecture verification," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 3, pp. 188–199, June 1995.

[6] D. Ince, "Software testing," in *Software Engineer's Reference Book* (J. McDermid, ed.), London, England: Butterworth-Heinemann, 1991.

[7] M. Abadir, J. Ferguson, and T. Kirkland, "Logic design verification via test generation," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, vol. 7, pp. 138–148, January 1988.

[8] R. Vemuri and R. Kalyanaraman, "Generation of design verification tests from behavioral VHDL programs using path enumeration and constraint programming," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 3, pp. 201–214, June 1995.

[9] H. Al-Asaad and J. Hayes, "Design verification via simulation and automatic test pattern generation," in *Proceedings International Conference on Computer-Aided Design*, pp. 174–180, 1995.

[10] R. C.-Y. Huang and K.-T. Cheng, "A new extended finite state machine (EFSM) model and its application to functional vector generation," *Third IEEE International High Level Design Validation and Test Workshop (HLDVT '98)*, 1998.

[11] P. Goel, "An implicit enumeration algorithm to generate tests for combinational logic circuits," *IEEE Transactions on Computers*, vol. C-30, pp. 215–222, March 1981.

[12] H. K. Lee and D. S. Ha, "On the generation of test patterns for combinational circuits," Technical Report 12-93, Virginia Polytechnic Institute and State University, Department of Electrical Engineering, College Station, TX 77840 USA, 1993.

[13] E. Sentovich, K. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli, "SIS: A system for sequential circuit synthesis," Memorandum UCB/ERL M92/41, University of California, Berkeley, University of California, Berkeley, CA 94720 USA, May 1992.

[14] P. Goel, "RAPS test pattern generator," *IBM Technical Disclosure Bulletin*, vol. 21, pp. 2787–2791, December 1978.

[15] M. Abramovici, J. J. Kulikowski, P. R. Menon, and D. T. Miller, "SMART and FAST: Test generation for VLSI scan-design circuits," *IEEE Design and Test*, pp. 43–54, August 1986.

[16] P. Goel and B. C. Rosales, "PODEM-X: An automatic test generation system for VLSI logic structures," in *Proceedings 18th Design Automation Conference*, pp. 260–268, June 1981.

[17] P. Goel, H. Lichaa, T. E. Rosser, T. J. Stroh, and E. Eichelberger, "LSSD fault simulation using conjunctive combinational and sequential methods," in *Proceedings IEEE Test Symposium*, pp. 371–376, November 1980.

[18] S. B. Akers and B. Krishnamurthy, "Why is less information from logic simulation more useful in fault simulation," in *Proceedings International Test Conference*, pp. 786–800, 1990.

[19] J. Dworak, M. R. Grimaila, S. Lee, L.-C. Wang, and M. Mercer, "Modeling the probability of defect excitation for a commercial IC with implications for stuck-at fault-based ATPG strategies," in *Proceedings International Test Conference*, pp. 1031–1037, 1998.

[20] H. Al-Asaad and J. P. Hayes, "ESIM: A multimodel design error and fault simulator for logic circuits," in *Proceedings of the VLSI Test Symposium*, pp. 221–228, 2000.