A Novel Framework for Functionally Untestable Transition Fault Avoidance during ATPG

Jeremy Lee[1], Nisar Ahmed[1], Mohammad Tehranipoor[1], Vinay Jayaram[2], and Jim Plusquellic[1]
[1]Dept. of CSEE, UMBC, Baltimore, MD, {jlee36,nisar1,tehrani,plusquel}@umbc.edu
[2] Texas Instruments, Dallas, TX, vjayaram@ti.com

Abstract— **Delay fault testing has proven to be a significant part of modern manufacturing testing. It has also become a source of overtesting due to detection of functionally untestable faults by invalid transitions that would not occur during functional operation of the chip. There has been previous work in the field that identifies these faults allowing them to be removed from active fault lists from ATPG tools. However, due to the random fill of don't-care bits in test patterns, incidental detection of functionally untestable faults becomes possible. In this paper, we propose a novel framework that will generate patterns using any commercial ATPG that avoid detection of these functionally untestable transition faults. Previous methods have required modification of the ATPG tool itself or designing a new ATPG to avoid such faults, making the immediate use of these tools very difficult. Our framework builds upon a commercial ATPG tool and modifies the netlist rather than the tool. By using the present functionality of the ATPG tool and a modified netlist, pattern generation is structurally constrained to avoid generating a pattern that will incidentally detect a functionally untestable fault. The proposed minimally affects test coverage of functionally testable faults and does not significantly increase the amount of effort needed by the ATPG tool.**

I. Introduction

Chip scaling and market demands continue to push designers to fit more complex designs into smaller areas. Not only does a higher design density correlate to a higher probability of defects, but as the feature size shrinks, the effective length of the interconnects become longer due to a shrinking width. This creates the potential for severe signal integrity problems in the chip, which often presents itself as signal delay. Delay fault testing detects many of these defects that cause gate and interconnect delay. However, design-for-testability (DFT) techniques like scan design that allow for easier delay fault coverage, also allow scan patterns that detect functionally untestable faults.

Functionally untestable faults create a difficult situation for many test engineers. If these faults are treated in a similar fashion as functionally testable faults, patterns are being applied to the chips that would never occur in the field, potentially failing chips unnecessarily. Overtesting can result in a potentially significant yield loss [1]. By avoiding these functionally untestable faults during pattern generation, we can prevent detection of these faults during test allowing only detection of functionally testable faults.

Current delay fault testing techniques already avoid a portion of the functionally untestable faults due to the method it is applied. Compared to launch-off-shift (LOS) [2] and enhanced scan [3], launch-off capture (LOC) [4] testing assists in avoiding the largest percentage of such faults due to the functional dependence between the initializing and transition launching patterns. However, functionally untestable faults can still be stimulated and detected by initializing the circuit-under-test (CUT) with a pattern that is not the functional response of the design. Avoiding these patterns during pattern generation will be the key to avoiding the respective untestable fault it stimulates.

Since it is possible to determine functionally untestable faults separately [5][6][7] before ATPG, a naive solution to avoid these faults is to exclude these faults from the active faults list during pattern generation. However, since most ATPG tools randomly fill any don't-care bits with a random value it may still be possible incidentally detect these faults. In order to ensure only functionally testable faults are detected, additional steps must be taken to constrain the ATPG tool from creating patterns with functionally invalid states.

Previous techniques that avoid functionally untestable faults during pattern generation have required custom modifications to ATPG tools or designing a new ATPG [8][9]. This makes the immediate application of their techniques to commercial ATPG tools rather difficult. The work in [8] constrains LOC by taking a random sample of patterns, determines which of those patterns will detect functionally untestable faults, and uses such patterns in conjunctive normal form (CNF) to constrain their custom ATPG tool. In [9], the custom ATPG tool contains a list of illegal state cubes that would detect functionally untestable faults. As LOC generation begins, the ATPG

* This work was supported in part by Semiconductor Research Corporation under contract SRC 2005-TJ-1322.

must check the list of illegal states to make sure the pattern it is generating does not contain such combinations.

In this paper, we propose a novel framework to be used as a wrapper around any commercial ATPG tool. Using a functionally untestable transition fault identification tool, we are able to determine which faults to avoid during ATPG. The avoided faults are then used as constraints by altering the netlist instead of modifying the ATPG algorithm. Transition fault pattern generation is then performed without the functionally untestable faults in the active list, but since the constraints are in place, patterns generated will not incidentally detect functionally untestable faults. The result of the ATPG will be a test pattern set that will only detect functionally testable faults.

The remainder of this paper is organized as follows. Section II covers an overview of the proposed framework. In Section III, we review the tool used to identify the functionally untestable faults. Section IV will discuss the constraint generation and minimization process. Sections V and VI describe the application and analysis, respectively, of the framework on the ISCAS'89 benchmarks using the Synopsys Design Tools [10]. Finally, we conclude our discussion in Section VII.

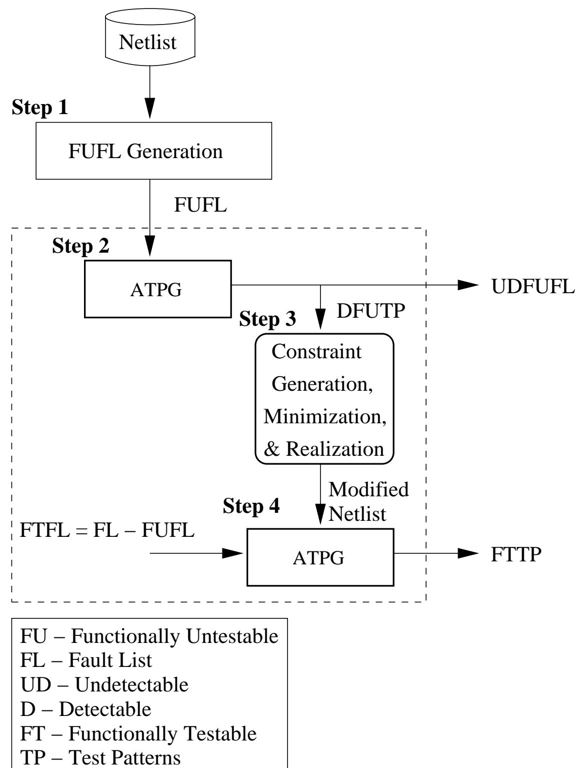

Figure 1. Flow of the proposed framework for functionally untestable fault avoidance.

II. Overview of Framework

The framework for functionally untestable transition fault avoidance consists of a four step process. Figure 1 outlines the general flow of the framework. First, a functionally untestable fault list (FUFL) is generated, which only needs to be performed once for each design. This list is generated using the technique described by Liu et al. [7], who extended FIRES [6], a sequential stuck-at redundant fault detection technique, towards application to transition faults.

Using the functionally untestable transition fault list, we can use existing ATPG tools to generate LOC patterns for each fault in the list without filling don't-care bits. The advantage of using LOC as the basis for pattern generation is that any fault that cannot be detected during LOC can be assumed to be functionally untestable due to the functional constraint LOC inherently applies to all test patterns. This pattern generation for functionally untestable faults will result in two lists (see Figure 1: *LOC detectable functionally untestable test patterns* (DFUTP) and *LOC undetectable functionally untestable fault list* (UDFUFL). Any faults that LOC ATPG can successfully generate a pattern for are LOC detectable and any fault that does not have a corresponding pattern are LOC undetectable. The UDFUFL are the existing 15–20% (according to our experimental results on ISCAS'89 benchmarks) of faults undetectable by LOC due to the inability to functionally stimulate and propagate the appropriate transition to an observable output. The remaining LOC detectable faults are a result of initializing the CUT with a pattern that is not a functional response during normal operation and cause chip overtesting.

The patterns generated for the LOC detectable faults (DFUTP) are then used in the next step, Constraint Generation, Minimization, and Realization, which will be used for ATPG. This step is done to make sure that running ATPG on functionally testable faults will not incidentally detect functionally untestable ones when filling in don't-care bits in the final patterns. Unlike previous techniques, we apply the constraints to the design netlist instead of modifying the ATPG tool to handle an entirely new set of constraints. For each care-bit of a LOC detectable fault pattern, there is a corresponding reachable input. A simple combinational logic tree can be added to the design on each of these reachable inputs for each pattern, which can then be ORed together with the trees of the other LOC detectable patterns. The single OR gate will be constrained by the ATPG to prevent generation of a pattern that will detect any faults in the FUFL. Most ATPG tools (e.g. Synopsys TetraMax [10]) can easily constrain a single net to a zero or a one. This will be further discussed in Section IV.

For designs that have a large number of DFUTPs, a form of constraint minimization must be used to reduce some of the burden that may be placed on the ATPG tool.

To alleviate this problem, in this framework, we reduce the size of constraints such that if many of the patterns have similar bit combinations, one logic tree is used to apply many of the constraints.

The modified netlist will then be used for ATPG. The fault list input into the ATPG tool consists of the functionally testable faults only, i.e. the original fault list minus those faults determined to be functionally untestable generated in Step 1 (FTFL = FL-FUFL). By constraining the output of the OR gate that was mentioned above to a logic zero, we prevent the ATPG tool from generating a pattern that will detect any faults in the FUFL since the don't-care bits of the final patterns are only filled with values that will not incidentally detect a functionally untestable transition fault.

III. Functionally Untestable Fault Identification

The first step of our framework is based on the functionally untestable transition fault identification technique in [7]. This technique uses static logic implication (static learning) to expand on the work in [5], which developed a technique called FIRE.

The efficiency of the identification technique is determined by the number of implications performed for each circuit. As was done in [7], we limit our program to direct, indirect, and extended backward implications. These implications are stored as a graph for efficient searching and easy traversal through the circuit. Direct implication is straight forward and can be learned based on the function of the gate. If we are given a simple circuit as in Figure 2, we can immediately connect the implications for $C = 1$ as shown in Figure 3(a). Indirect implications are derived based on the direct implications. For example, if we again imply a 1 on net C, the direct implications imply nets A and B are both 1, which each directly imply $D = 1$. So an indirect implication can be made with $C = 1$ and $D = 1$, implying $E = 0$, which itself has a set of direct implications and becomes part of the implication graph of $C = 1$. Extended backward implication is used in cases where the output of a gate is known, but the inputs cannot be directly implied based on the current output. One example of this is when the output of an AND gate is 0, since it is not directly implied as to whether one input is 0 or both are 0. Extended backward implication will determine whether there are any common implications when implying the dominant value on each of the inputs of the current gate. After direct, indirect and backward implications are completed, the final implication graph for net D implied to 0 for the circuit shown in Figure 2 will be as shown in Figure 3(b).

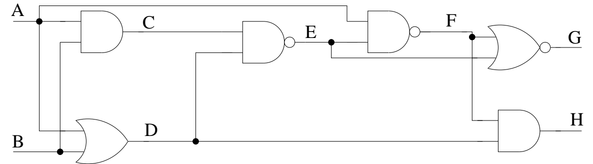

Figure 2. A simple combinational circuit used to demonstrate static logic implication.

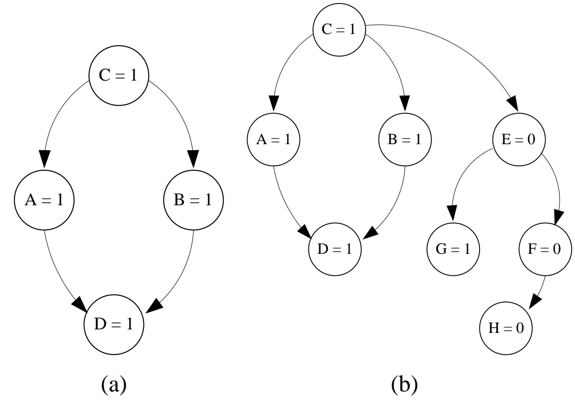

Figure 3. (a) Direct implications graph for C = 1. (b) Graph after direct, indirect, and backward implications have been performed.

There are many additional implications that can be performed to more fully describe the circuit [11], which result in a more complete description of the circuit and potentially yield a larger set of identified functionally untestable faults. However, this increases the complexity of the program and the time required to perform the implications. Since the fault identification process is not the direct focus of this work we limited the number of implications to reduce the time requirement of the program.

The implication graphs learned from static learning help to easily identify functionally untestable faults, which will become the FUFL. In order to identify these faults, a single-line conflict technique similar to FIRE is used to identify any functionally untestable faults. However, the original FIRE must first be extended to include static learning [12]. Extending FIRE basically results in the intersection of the implication graphs of a net implied to 0 and 1. If the intersection yields a non-empty set, any implications in this non-empty set can be identified as functionally untestable faults. This technique determines the faults that are sequentially uninitializable and sequentially uncapturable faults. Sequentially uninitializable faults cannot be detected due to redundancies in the design that prevent nets from being set to a logic one or zero. Sequentially uncapturable faults are due to an inability to propagate a transition to an observable output.

In order to extend FIRE further for application towards functionally untestable transition faults, sequentially constrained faults must also be considered. Sequentially constrained faults can be individually initialized and captured,

but are functionally constrained by LOC such that the two faults cannot occur sequentially. Similar to the sequentially uninitializable and sequentially uncapturable faults, this relation is dependent on an intersection operation. This set considers the union of the initializable set with the capturable set of a net implied to 0 intersected by the union of the initializable set with the capturable set of the same net implied to 1. Any implications in the non-empty set are considered sequentially constrained and functionally untestable.

IV. Constraint Generation, Minimization, and Realization

Constraint generation is based on the faults that were identified in the first step of the functionally untestable fault avoidance framework described in Section III. To determine these constraints, a commercial ATPG tool is used to generate LOC transition fault patterns for the FUFL, which is then minimized and then transformed into constraints that are temporarily added to the final design solely for the purpose of transition fault pattern generation in the final step of the framework.

Under the assumption that the faults identified by Step 1 of the framework are functionally untestable transition faults, we also assume that those faults will never be sensitized during normal operation and are only sensitizable due to states that are initialized during test mode through the scan chain. So, it is only during the initialization phase of LOC that functionally untestable fault patterns must be avoided since the launch pattern is a functional response of the circuit and is based on the initialization pattern. In other words, we make sure the initialization pattern is a valid, reachable states, which makes the functional response state valid.

A. Constraint Generation

In order to prevent the detection of these faults during LOC ATPG, the scan cells must be constrained from any state that would place an LOC initialization pattern for any faults in the FUFL into the scan chain. To realize these constraints, LOC ATPG is performed on the FUFL (Step 2 of framework). During this ATPG step, don't-care bits must remain unfilled in order to isolate only those cells that are necessary for detection of functionally untestable faults. This is easily done in Synopsys' TetraMax by removing the random fill option during ATPG.

LOC ATPG will determine test patterns for a fraction of all faults in the FUFL, leaving the remaining faults as undetectable functionally untestable faults (UDFUF). Since LOC in general leaves approximately 20% of the total faults of a design undetected, the set of faults declared as UDFUF should be the same faults as the 20% of total faults that are LOC undetectable.

B. Constraint Minimization

Although the UDFUFs make up a majority of the functionally untestable faults, for larger designs, the number of patterns generated during the second step of the framework (DFUTP) can become rather large. If constraints were based solely from this large set of patterns, it could unnecessarily constrain the ATPG tool too much and require significantly more effort and time to avoid the faults in the FUFL. Therefore, a constraint minimization strategy is used to minimize the time the ATPG tool spends on constraints.

The method used to minimize the number of constraints is based on *pattern dominance*. For example, assume there is a design with 8 scan cells and with four patterns in the format $b_7b_8b_6b_5b_4b_3b_2b_1b_0$ that detect some functionally untestable faults: XXX1X0XX, XX11X0XX, XX0XXX0X, XXX100XX. There are clearly similarities in the first, second, and fourth pattern. Assuming each of these patterns detect different, but intersecting fault sets, the pattern with the fewest number of care bits can be used to determine whether the other patterns can be eliminated from constraint consideration. In this example, the first and third patterns have the fewest number of care bits. Starting with the first pattern, searching for all other patterns that also contain the same care bits yield the second and fourth patterns, so these two patterns can then be eliminated from constraint consideration. Since the third pattern does not have the same care bits in the same positions as the first pattern, it cannot be removed from consideration. As a result only XXX1X0XX and XX0XXX0X are used as constraints during ATPG.

The reason the second and fourth patterns could be eliminated from the pattern set is due to pattern dominance. The first pattern will detect at least one functionally untestable fault with the given care bits. The second pattern will also detect that same faults as the first pattern plus those detected due to the additional care bits in the pattern. However, a necessary condition to detect those additional faults are the care bits of the first pattern. So, if the first pattern is constrained from occurring, then those faults detected by the second and fourth patterns will not be detected either. Because of this constraint, the ATPG tool will not generate a pattern that contains a 1 at b_4 and a 0 at b_2.

C. Constraint Realization

After constraint minimization, the constraints are finally realized into a netlist that will be used for the final design. Since the position of the care bit in the pattern stream and the scan cell order is known, a one to one mapping between care bit and scan cell can be made. For every pattern that remains after generation and minimization, a

single behavioral equation can be formulated based on the sequence of care bits.

Using the first pattern from the example above, XXX1X0XX, the constraint created by the pattern can be realized as $pattern_1 = cell_4 \cdot \overline{cell_2}$, where the dot represents the logical AND operation. The equation represents the case when b_4 and b_2 are 1 and 0, respectively, excluding the remaining bits since they are don't-care states. Equation 1 is a generalized form of the constraints created from the DFUTP, where n is the total number of scan cells in the design.

$$pattern_j = \prod_{i<n} c_i, c_i = \begin{cases} cell_i & \text{if } b_i = 1 \\ \overline{cell_i} & \text{if } b_i = 0 \end{cases} \quad (1)$$

$$constraint = \sum_{j<k} pattern_j \quad (2)$$

To ease the application of the ATPG constraints, rather than constraining each of the $pattern$ signals to 0, if all the signals are ORed together, a single net will have to be constrained to a 0 to ensure the ATPG will not generate patterns that contain a state that will incidentally detect a functionally untestable fault. Equation 2 shows the general form for performing the logical OR operation between all of the $pattern$ signals, where k is the total number of patterns after minimization.

Once constraint generation, minimization, and realization is performed, there is now a functional description of states to avoid during LOC ATPG. It must now be integrated into the current design in order to be effective. A more detailed implementation of constraint integration with the targeted design is described in the following section.

V. Framework Implementation

We implemented our framework to easily wrap around existing commercial synthesis and test tools. The framework was implemented using a C program to fully integrate each of the four steps together. Synopsys DFT Compiler and TetraMax [10] were the commercial tools used to synthesize and generate patterns. We have listed the framework implementation below in an easy to follow flow.

Functionally Untestable Fault Avoidance Flow

---

1) Synthesize design and insert scan chains using DFT Compiler.
2) Run functionally untestable fault identification program on synthesized netlist. (Framework Step 1)
   - Functionally untestable fault list (FUFL) is generated by the program.
3) Use FUFL as fault list in TetraMax and perform LOC ATPG on synthesized design. (Framework Step 2)
   - TetraMax generates patterns to detect faults that are LOC detectable but functionally untestable (DFUTP).
4) DFUTP are extracted and minimized
   - Behavioral model of constraints is generated. (Framework Step 3)
5) Constraint model is synthesized and optimized using DFT Compiler and connected to already synthesized design.
6) The final netlist is used for LOC ATPG in TetraMax with a functionally testable fault list. (Framework Step 4)

The functionally untestable fault identification program was implemented in C and followed the techniques explained in Section III. A Perl script was also used to extract the DFUTP from the STIL file, minimize the number of patterns, and generate the behavioral model.

DFT Compiler was used to incorporate the generated constraints into the synthesized design. By passing the constraints netlist through the synthesis tool as a separate module, many redundancies in the constraint netlist were removed and the constraints were reduced to a structural netlist that maintained the original function of the behavioral model. Adding the constraint module to the design was straightforward and maps the the output net of all the scan cells to the input ports of the constraint module. Since the STIL file that extracted the patterns contained the order of the scan cells, the constraint module inputs were placed in the same order as the scan cells. The output signal of the constraint module is tied to the output of the final logical OR operation of all the $pattern$ signals to simply searching for the signal when applying the single constraint during ATPG.

When using this final netlist with the included constraints with TetraMax, the ATPG is constrained to always hold the single output of the constraint logic to 0. Although this is functionally equivalent to constraining the scan chain to individually prevent the generation of functionally untestable fault states, it is significantly easier to apply the constraint on a single net as opposed to constraining the scan cell values individually or on a per pattern basis.

VI. Analysis

The framework was run on the ISCAS'89 benchmarks using a 3.2 GHz Pentium 4 with 1 GB of memory running the Linux Operating System. In Table I, we have listed the

TABLE I
IDENTIFIED FUNCTIONALLY UNTESTABLE FAULTS.

Bench Name	Total # of Faults	FUFL based on [7]
s1423	3028	5
s5378	6822	118
s13207	15534	25
s15850	18240	324
s38417	56490	3010

number of functionally untestable faults that were identified by the first step of our framework. The first column lists the benchmark name and the second column lists total number of faults identified by TetraMax. Finally, in the third column, we list the number of faults identified as functionally untestable by the technique referenced in Section III.

The number of functionally untestable faults found by our implementation of the technique have found significantly fewer faults than those discovered in [7]. We believe this is due to a combination of circuit optimizations performed by DFT Compiler and the possibility of our own tool performing fewer implications than what was done in [7]. We intend to study this matter further in future work in order ensure all functionally untestable faults are correctly being identified by this tool.

We show the results of the proposed framework in Table II. Columns 2 and 3 show the results of the ATPG using TetraMax for conventional LOC and the coverage when applying the framework. For both LOC ATPG and ATPG with our framework, we included the entire fault list of the design to show the constraints filtering out those faults in the FUFL. Column 4 shows the number of patterns used after minimization to generate the constraints for the final ATPG. Finally, column 5 is the overall time of the framework, which does not include the time taken to identify the functionally untestable faults since that is only performed once for each design.

The number of constraints listed in Column 4 of Table II correlates closely with the number of functionally untestable faults identified in Column 3 of Table I. Since the FUFL identification program did not identify as many functionally untestable faults for each of the benchmarks as desired, the number of constraints were quite limited. The number of constraints grows linearly with the number faults in the FUFL and, for the cases we have provided, remains roughly half the number of faults identified as functionally untestable.

As can be compared between Columns 2 and 3, the fault coverage between LOC and our framework is directly related to the number of constraints used. For s1423 and s13207, since our program was not able to identify many functionally untestable faults, the number of constraints were few, and very few functionally untestable faults were filtered out during pattern generation.

For the remaining three cases, there was a sufficient number of constraints to clearly show a substantial drop in fault coverage. For s5378 and s15840, there was almost a 20% drop in coverage. In each case, the fault identification tool only indicated approximately 2% of the total faults as functionally untestable. However, if the percentages are compared with the percentage of functionally untestable faults for those two benchmarks in [7], we can assume the additional faults excluded during the framework ATPG are also functionally untestable.

Since we are essentially determining invalid/unreachable states using the framework, we assume any fault that requires initialization with an invalid state is functionally untestable. Due to this, by first finding a subset of functionally untestable faults and the invalid states that would detect them, additional faults will also potentially be filtered out by this process since these additional faults also require the same functionally unreachable states to be detected. So, with a greater number of constraints, the more states that can be concluded as invalid.

However, as can be seen with s38417, since there are so many constraints that still remain after minimization, the ATPG tool cannot effectively reach a high fault coverage and is obviously impaired. This problem most likely can be alleviated with a better constraint minimization technique, which will be pursued further in future work. An appropriate balance obviously must be reached between constraining the ATPG to effectively prevent detection of functionally untestable faults and the ATPG effort load.

Overall, the framework did not increase the amount of time to complete pattern generation by an exorbitant amount, even on s38417. The increase in time was mainly due to the ATPG accounting for over 1000 constraints. Reducing the number of constraints will potentially reduce the pattern generation time in addition to restoring the fault coverage to an acceptable level. Even with almost 100 constraint for s15850, the entire framework flow took less than one minute with the majority of the time spent on constraint generation, minimization, and realization instead of pattern generation with the constrained ATPG.

VII. CONCLUSION

We have presented a novel framework for avoiding functionally untestable faults during pattern generation that can be used in conjunction with a commercial ATPG tool. Rather than altering a custom tool as previous implementations have done, the netlist is modified to include additional logic that is constrained during ATPG. The additional constraint ensures the ATPG does not generate an

TABLE II

FUNCTIONALLY UNTESTABLE FAULT AVOIDANCE FRAMEWORK RESULTS

Benchmark Name	LOC Fault Coverage(%)	Framework Fault Coverage(%)	# of Constraints	Framework Time(s)
s1423	95.72	95.55	2	6.5
s5378	89.06	70.62	41	9.5
s13207	89.99	88.04	10	34.5
s15850	89.46	71.50	94	29.1
s38417	96.80	49.14	1105	539

LOC test pattern that will detect a functionally untestable fault. Application of the framework is straightforward and does not significantly increase pattern generation time nor hinder the ATPG from reaching reasonable coverage levels if there are a manageable number of constraints that are applied. Initial results show that a current commercial ATPG tool without significant modification can be used to avoid incidental detection of faults that have already been identified as functionally untestable and potentially identify additional faults.

REFERENCES

[1] J. Rearick, "Too Much Delay Fault Coverage Is a Bad Thing," in *Proc. of International Test Conference (ITC)*, 2001, pp. 624–633.

[2] J. Savir, "Skewed-Load Transition Test: Part I, Calculus," pp. 705–713, 1992.

[3] J. Savir and S. Patil, "On Broad-Side Delay Test," pp. 284–290, 1994.

[4] B. Dervisoglu and G. Stong, "Design for Testability: Using Scanpath Techniques for Path-Delay Test and Measurement," pp. 365–374, 1991.

[5] M. A. Iyer and M. Abramovici, "FIRE: A Fault-Independent Combinational Redundancy Identification Algorithm," *IEEE Transactions on VLSI Systems*, vol. 4, no. 2, pp. 295–301, June 1996.

[6] M. A. Iyer, D. E. Long, and M. Abramovici, "Identifying Sequential Redundancies Without Search," in *Proc. of Design Automation Conf. (DAC)*, 1996, pp. 457–462.

[7] X. Liu and M. S. Hsiao, "On Identifying Functionally Untestable Transition Faults," in *IEEE Intl. High-Level Design Validation and Test Workshop*, 2004, pp. 121–126.

[8] ——, "A Novel Transition Fault ATPG That Reduces Yield Loss," *IEEE Design & Test of Computers*, pp. 576–584, 2005.

[9] Z. Zhang, S. M. Reddy, and I. Pomeranz, "On Generating Pseudo-Functional Delay Fault Tests for Scan Designs," in *IEEE Intl. Symposium on Defect and Fault Tolerance in VLSI Systems*, 2005, pp. 398–405.

[10] Synopsys Inc., "User Manuals for Synopsys Toolset Version 2005.09," Synopsys Inc., 2005.

[11] M. Syal, R. Arora, and M. S. Hsiao, "Extended Forward Implications and Dual Recurrence Relations to Identify Sequentially Untestable Faults," in *Intl. Conf. on Computer Design*, 2005.

[12] J.-K. Zhao, J. A. Newquist, and J. H. Patel, "A Graph Traversal Based Framework for Sequential Logic Implication with an Application to C-Cycle Redundancy Identification," in *Proc. of Intl. Conf. on VLSI Design*, 2001, pp. 163–169.