Pattern Generation for Mutation Analysis using Genetic Algorithms

Yen-Chi Yang, Chun-Yao Wang, Ching-Yi Huang, and Yung-Chih Chen[†]
Department of Computer Science, National Tsing Hua University, HsinChu, Taiwan 300
[†]Department of Computer Science and Engineering, Yuan Ze University, ChungLi, Taiwan, 320

Abstract—**Mutation Analysis (MA) is a fault-based simulation technique that is used to measure the quality of testbenches for mutant detections where mutants are simple syntactical changes in the designs. A mutant is said living if its error effect cannot be observed at the primary outputs. Previous works mainly focused on the cost reduction in the process of MA, because the MA is a computation intensive process in the commercial tool. For the living mutants, to the best of our knowledge, the commercial tool has not addressed the pattern generation issue yet. Thus, this paper presents a Genetic Algorithm to generate patterns for detecting living mutants such that the quality of the verification environment is improved. The experimental results show that more living mutants can be detected after adding the generated patterns in the testbench.**

I. Introduction

Functional verification, a process to ensure that the design specification and the implementation are consistent [6] [8] [24]∼ [27] [29], is an important process in the design flow. As designs become larger, functional verification process becomes increasingly more complex, and consumes more than 70% of the design effort. This percentage continues to increase with the growth of design complexity. Therefore, functional verification is a critical issue for designers.

Coverage metrics were proposed to assess the verification quality of a design. It can be divided into two categories: structural coverage (code coverage) [19] [22] and functional coverage [11] [12] [22]. Structural coverage identifies the percentage of the design that has been executed by the testbench. This indicator can help designers improve the testbench quality. However, structural coverage does not consider stimuli's abilities for propagating the error effects to observation locations. Therefore, the higher scores in structural coverage metrics do not indicate the better qualities of the patterns.

On the other hand, functional coverage considers the semantic interpretation of the functionality to exercise the design. The functional coverage metric is defined and evaluated by designers based on the details in the design specification. However, functional coverage does not check unexpected or inappropriate operations. Thus, it is a metric that provides a feedback on how well the stimuli cover the operations described in the specification, but is not a good measure for the evaluation of the quality and completeness of the verification environment.

Mutation Analysis (MA) is a fault-based technique that can improve the quality of the verification environment, which originated from software engineering field [9] [14] in the early 1970s. By injecting artificial faults into the original implementation, designers can check whether the verification environment can differentiate the faulty and original designs [5]. The MA approach is based on two hypotheses: the Competent Programmer Hypothesis [4], and the Coupling Effect Hypothesis [9]. The Competent Programmer Hypothesis states that programmers usually develop programs that perform the intended tasks with some errors. However, these errors are simple and can be corrected by small syntactical changes. The Coupling Effect Hypothesis states that complex errors are coupled from simple errors in such a way that the pattern set for detecting all simple errors in a program will also detect a high percentage of complex errors [20].

A mutant is an operation that is changed in the program. Given a program statement: a = b & c, if we accidentally type "|" instead of "&" in this statement, a mutant is injected into the original program. We say that a pattern is able to activate a mutant if the output value of the original statement is not equal to that of the mutated statement. If these different values are observed at the primary outputs (POs), we say the mutant is killed; otherwise, living, which indicates the weakness of the verification environment.

There is some research related to MA that mainly aims for cost reduction [17]. This is because MA needs high computation expenses. The cost reduction usually deals with simulation reduction or mutant reduction. The work [18] proposed a preprocessing technique for accelerating MA process. It analyzed the error propagation ability of each mutant before performing the simulation in RTL designs. The analyzed results can be used to reduce the simulation cost of MA. The work [15] proposed a mutation-based mutant ordering heuristic to reduce the effective mutant numbers for Simulink models.

There are other works related to MA. The work [3] implemented an MA approach over RTL designs for testbench qualification. It measures and drives the quality improvement of all aspects of functional verification. In [23], the authors proposed an approach for system testability verification based on an adaptation of the Weak Mutation analysis technique [16]. Weak Mutation assumes that each statement in a design is an observation point. Thus, if a mutant causes the output of a statement changed, the mutant is also said killed under the Weak Mutation. The work [23] translated the mutants in the RTL designs as the faults in the gate-level netlist and generated the patterns for detecting the faults based on the Weak Mutation. The work [28] proposed a functional test generation approach where simulation results are used to guide the generation of additional test patterns.

In this work, we aim at generating the patterns for detecting the living mutants. However, we do not directly target the living mutants at RTL designs due to high complexity[1]. On the contrary, since the living mutants in the RTL designs represent the undetected faults in the gate-level netlist after the synthesis [23], we generate patterns for the undetected faults in the gate-level netlists instead. Furthermore, we exploit the divide and conquer concept to deal with the pattern generation problem. To detect a fault, there are many conditions in the fault propagating path that have to be simultaneously satisfied. For an undetected fault, however, some conditions may have been satisfied via random simulations. Thus, we propose a genetic algorithm (GA)-

This work is supported in part by the National Science Council of R.O.C. under Grant NSC 101-2628-E-007-005, NSC 100-2628-E-007-031-MY3, and NSC 101-2221-E-155-077.

[1] Complex conditions and large ranges of variables in the RTL designs increase the complexity of mutant detection.

978-1-4673-5762-3/13/$31.00 ©2013 IEEE

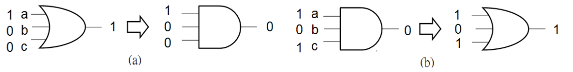

Fig. 1. (a) Modeling the OR to AND mutant as a stuck-at 0 fault at the input a. (b) Modeling the AND to OR mutant as a stuck-at 1 fault at the input b.

based method, which collects/combines/evolves these useful sub-patterns that satisfy some conditions, to obtain new patterns such that more living mutants can be detected after adding the generated patterns in the testbench.

II. Background

A. Mutant Modeling

A mutant is an injected error in the RTL designs, and it could be translated as stuck-at faults in the gate-level netlist [23]. Thus, a living mutant escaping from detection in the testbench can be represented as hard-to-detect stuck-at faults in the netlist. Furthermore, according to the Coupling Effect Hypothesis in the MA, we can only consider the single stuck-at fault model instead of the multiple stuck-at fault model.

Here we propose a method that models a mutant as a stuck-at fault. Here we only consider three types of mutants.

1) OR to AND mutant: We assume an OR gate is replaced with an AND gate. To distinguish these two operations, their output values have to be different. In Fig. 1(a), assume we set the output of the OR gate as 1, and set that of the AND gate as 0, then we can derive an input pattern abc = 100 that satisfies the output values of the OR gate and the AND gate. Thus, we can model this OR to AND mutant as the input a stuck-at 0 fault. This is because after detecting this stuck-at 0 fault, we can get the pattern abc = 100 and detect this OR to AND mutant.

2) AND to OR mutant: Similarly, in Fig. 1(b), we can model this AND to OR mutant as the input b stuck-at 1 fault. This is because after detecting this stuck-at 1 fault, we can get the pattern abc = 101 and detect this AND to OR mutant.

3) NOT to BUF mutant: Similarly, we model this NOT to BUF mutant as the input stuck-at 1 fault.

B. Error Propagation Analysis

Error Propagation Analysis (EPA) is a technique that statically analyzes the structures of the designs and evaluates the error propagation ability for a given mutant. Before detailing the EPA concept, we first introduce the Mutant Controllability (MC) and Decreasing Rate (DR), which will be used in the EPA. The MC represents the changing probability of each signal. For the signal on the output of the mutated gate, its MC value is set 1. The MC value calculation can be found in [18]. When the MC value of the signal becomes 0, that means the mutant effect does not influence this signal. The higher MC value of a signal means that the mutant effect is able to change this signal with a higher probability. After calculating the MC value for every signal on every path, we derive the DR value of each signal from its MC value. The DR value represents the degree of the error-masking effect from this signal. The greater DR value means that the error-masking effect is stronger. We calculate the DR value for every signal along each path to the POs, and find out a path that has smaller DRs for better propagating the mutant effect. The DR value calculation can be found in [18].

C. Mutant Propagation Path Ranking

For a living mutant that has many propagation paths to the POs, we need to focus on finding a “good” path for propagating the mutant effect. Thus, we apply the EPA technique [18] as mentioned to rank the paths such that a path having a higher probability to propagate the mutant effect will be selected first.

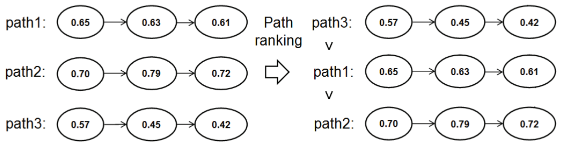

Fig. 2. An example for ranking paths based on the value of DR.

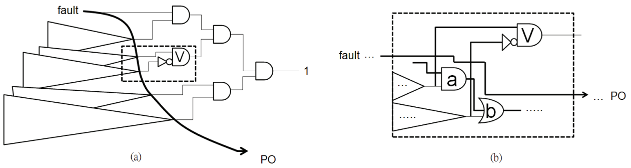

Fig. 3. (a) The construction of the AND gate tree. (b) The details in the square of (a).

For example, in Fig. 2, the DR values are shown on the nodes. For every path, we select the node with the largest DR value, and rank the paths based on this value in an ascending order. With having this path ranking, then we sequentially examine each path within a run time limit. If one path fails, we select the next one until the fault effect is propagated out. In Fig. 2, the largest DRs in $path1$ to $path3$ are 0.65, 0.79, and 0.57, respectively. Thus, the paths are ranked as $path3 > path1 > path2$.

III. The proposed approach

A. AND Gate Tree Construction

After modeling a living mutant as a fault and selecting a path for fault effect propagation, we construct an AND gate tree with respect to this path. This AND gate tree is used for justifying whether the fault effect is propagated out. To propagate the fault effect, two conditions have to be held simultaneously. First, the fault has to be activated. Second, all side inputs on the propagating path have to be set as the non-controlling values with respect to the gate types. $1\{0\}$ is the non-controlling value of an AND$\{$OR$\}$ gate.

In Fig. 3, assume the bold line is the fault propagation path from the fault to a PO. We build an AND gate tree from the side inputs of the fault propagation path as follows: for a side input of an AND gate a on the fault propagation path, we directly connect it to the AND gate *V*. However, for a side input of an OR gate b on the fault propagation path, we connect it to the AND gate *V* with an inverter. Based on this construction, we can realize that the 1 value at the root gate of the AND gate tree indicates that the fault is detected; otherwise, the fault is undetected. However, by analyzing the internal values of this AND gate tree, we can identify the side inputs that cause the fault undetectable. That is, the leaf nodes having 0 value in the AND gate tree. On the other hand, we can also know that which of the side inputs have been assigned the non-controlling values in the random simulations. Thus, we can collect the sub-patterns with respect to these side inputs for further use in the GA algorithm. The details will be discussed in the next subsection.

B. Genetic Algorithm

GA [13] is a searching technique that emulates biological evolution for having an optimal solution, and has been used in many research directions [7] [21]. The proposed GA is applied to create patterns that can detect the fault.

The proposed GA consists of five steps, and they are Record, Creation, Inversion, Crossover, and Mergence & Mutation.

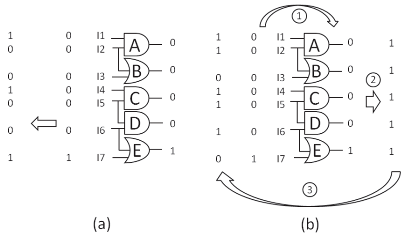

Fig. 4. (a) Creation. (b) Inversion.

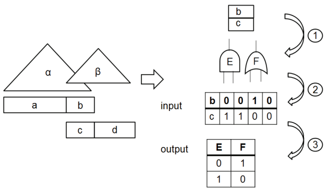

Fig. 5. An example for the step of Crossover.

1) Record: After the random simulations, without loss of generality, we assume that the random patterns cannot detect the fault. However, based on the internal values of the AND gate tree, we can record some sub-patterns that *propagate* the fault to the next gate. We name these sub-patterns *recorded-positive-sub-patterns*. We also record some sub-patterns that *do not propagate* the fault to the next gate, which are also named *recorded-negative-sub-patterns*.

2) Creation: In this step, we generate additional new sub-patterns from the recorded-positive-sub-patterns. For example in Fig. 4(a), assume the recorded-positive-sub-pattern is <I1∼I7> = <0000001>. A new sub-pattern <I1∼I7> = <1001001> that causes the same side inputs in the AND gate tree can be created based on the output values of gate A ∼ gate E.

3) Inversion: In this step, we use the recorded-negative-sub-patterns to generate new sub-patterns to increase the diversity of the sub-patterns. For example in Fig. 4(b), assume the recorded-negative-sub-pattern is <I1∼I7> = <0000001>, the corresponding output values of gate A ∼ gate E are <00001>. We then sequentially invert these output values and backward justify the input values. Thus, the input values of I1 and I2 are both changed to 1 when the output of gate A is changed from 0 to 1. However, there might exist some situation that the input values are conflict when performing this backward justification. For example, if gate E is changed from 1 to 0, then both I6 and I7 have to be assigned 0. However, I6 = 0 is conflict with I6 = 1 that is a necessary assignment of gate D's 1 value. Hence, we will keep the output value of gate E intact under this situation.

4) Crossover: For these generated sub-patterns, they may have some overlapped inputs. Thus, we use a crossover operation to obtain more sub-patterns in the overlapped inputs for increasing the diversity. For example in Fig. 5, b and c are the overlapped inputs of two sub-patterns and they are 0010 and 1100, respectively. Assume the gates in this overlapped region are gates E and F, then EF = 01 for b and EF = 10 for c. Our crossover operation changes the values in the gates E and F from 01 to 00, and from 10 to 11 as shown in Fig. 6. Then we can obtain four additional sub-patterns, 0000, 1000, 1111, and 1110 in the overlapped inputs.

5) Mergence & Mutation: After generating a lot of sub-patterns, we then merge multiple sub-patterns to form a complete pattern. If the overlapped inputs of two sub-patterns are identical, we directly merge these two sub-patterns as shown in Fig. 7(a). Otherwise, we randomly set the conflict bits as 0 or 1 as shown in Fig. 7(b).

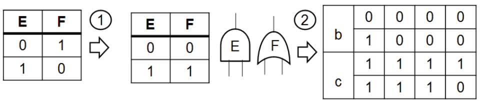

Fig. 6. An example for the step of Crossover.

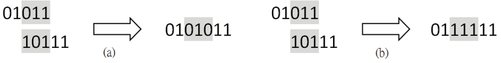

Fig. 7. An example for the step of Mergence & Mutation.

After having patterns in the population from these five steps of GA, we then verify whether these patterns can detect the fault or setting the non-controlling values to more side inputs of the fault propagation path from the AND gate tree. If the fault is detected, we terminate the algorithm. If more side inputs are assigned the non-controlling values, the pattern quality is improved and we proceed to the next iteration; otherwise, we remove the patterns that make less number of side inputs having the non-controlling values from the population.

C. Overall Flow

Fig. 8 is the overall flow of our work. The inputs are a circuit and a fault. We first select a path with higher probability to propagate the fault and build the corresponding AND gate tree. Then we perform random simulations and check the output value of root gate of the tree. If the output value is 1, the fault is detected; otherwise we perform the GA. If the new patterns from the GA is able to detect the fault, the algorithm is terminated; otherwise the next GA iteration will be performed again.

IV. Experimental Results

We implemented our algorithm in C++. The benchmarks are from the ISCAS'85 [1]and ITC'99 [2]. The experiments were conducted on an Intel Core$^{\text{TM}}$2 Quad 2.5 GHz Linux platform (Ubuntu 10.10). We conducted two experiments in this work. The first one is to show whether the generated new patterns can detect the living mutants based on our mutant modeling in the environment of a commercial tool, Certitude$^{\text{TM}}$ [3]. The second one is to show whether the proposed GA approach can be used as an effective pattern generator for hard-to-detect faults in testing. The flow of the first experiment is shown in Fig. 9. We first generate random patterns to detect the mutants in the design under the Certitude environment. The considered mutants are those discussed in Section II.A. For the living mutants, we model them as faults as mentioned and generate additional patterns by the GA. Then we add these generated patterns into the testbench again and check if these mutants are detected or not. The experimental results are shown in Table I. Column 2 lists the number of the total faults after the mutant modeling. Column 3 lists the number of faults detected by the random patterns and the random patterns plus GA patterns. Column 4 lists the pattern numbers. According to Table I, we find that more living mutants are detected in the Certitude report after adding the generated GA patterns into the testbench.

For the second experiment, we first perform the PODEM algorithm to collect hard-to-detect faults. These hard-to-detect faults are the fault that are not detected under the condition of backtrack limit = 10^8. Thus, only a few hard-to-detect faults are identified and listed in Table II. Columns 2 and 3 list the number of the PIs and the number of the gates in the circuit. Column 4 lists the fault and its faulty value. Column 5 lists the total number of the side inputs on the selected propagation path. Column 6 lists the CPU time by applying the PODEM and the proposed GA algorithm to detect the fault.

For example in *B20*, the hard-to-detect fault is n16234→n16227, stuck-at 1. There are 19 side inputs on the propagating path. Our GA approach spent 358.89 seconds to detect the fault while the PODEM did not detect the fault within 1537.10 seconds. According to Table

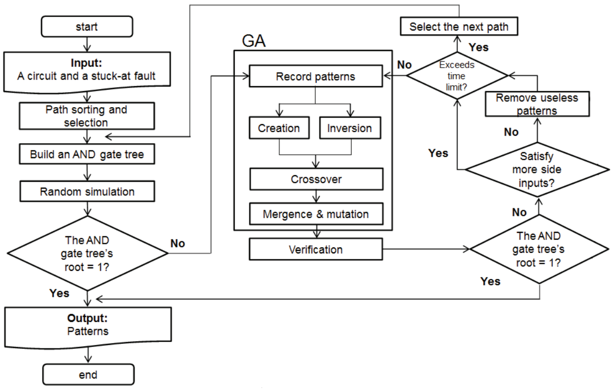

Fig. 8. Our overall flow.

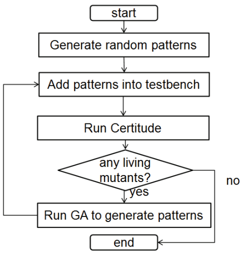

Fig. 9. The flow of the mutant detection.

II, our GA approach can generate effective patterns for these hard-to-detect faults.

V. Conclusion

Living mutant detection is an important task in the Mutation Analysis. However, commercial EDA tools have not addressed this issue yet. In this paper, we propose a Genetic Algorithm to generate patterns for detecting living mutants in the designs. The experimental results show that the GA approach can generate effective patterns to kill the living mutants in the designs.

Acknowledgement

The authors would like to thank the supports of Certitude tool from Dr. Kai Yang in SpringSoft.

TABLE I
The experimental result for mutant detection.

Circuit	|fault|	|Detect|		|Pattern|	
		Ran.	Ran.+GA	Ran.	Ran.+GA
C17	28	21	28	20	27
C499	1240	714	1234	1000	1123
C880	1157	856	862	1000	1074
C1355	1208	128	786	1000	1075
C1908	1601	108	1092	1000	1054
C2670	4080	2901	3446	1000	1167
C3540	3850	1703	1718	1000	1100
C5315	5627	2073	3326	1000	1201
C6288	9696	9677	9682	10000	10015
C7552	11788	11768	11788	10000	10020

TABLE II
The experimental results for hard-to-detect faults.

Circuit	|PI|	|gate|	fault/faulty value	|SI|	CPU time (s)	
					PODEM	GA
B20	522	10802	$n16234 \rightarrow n16227/1$	19	>1537.10	358.89
B21	522	10977	$n11374 \rightarrow n11400/1$	12	>2453.87	8.96
			$n15763 \rightarrow n16922/1$	23	>2644.32	162.45
			$n15419 \rightarrow n16922/1$	23	>1947.67	277.88
			$n13339 \rightarrow n13338/1$	12	>9835.98	30.03
			$p1\_ir\_reg\_13\_ \rightarrow n18561/1$	13	>7675.01	128.20
			$p1\_ir\_reg\_15\_ \rightarrow n18561/1$	12	>7729.62	193.19
			$p2\_ir\_reg\_11\_ \rightarrow n10521/1$	7	>7520.15	55.75
			$p2\_ir\_reg\_8\_ \rightarrow n13654/1$	8	>10801.20	74.18
			$p2\_ir\_reg\_12\_ \rightarrow n10563/1$	7	>6625.60	56.51
B22	767	16689	$n23133 \rightarrow n23039/1$	10	>11545.20	8.54

References

[1] *ISCAS'85 Benchmarks* [Online].
Available: http://www.iscas.net/

[2] *ITC'99 Benchmarks* [Online].
Available: http://www.cerc.utexas.edu/itc99-benchmarks/bench.html

[3] *SpringSoft.* [Online]. Available: http://www.springsoft.com/

[4] A. Acree, et al, "Mutation analysis," Georgia Institute of Technology, Technical Report GIT-ICS-79/08, 1979.

[5] A. Benso, et al,"A functional verification based fault injection environment," *in Proc. Defect and Fault-Tolerance in VLSI Systems*, 2007, pp. 114 - 122.

[6] J. Bergeron,*Writing Testbenches-Functional Verification of HDL Model.* Norwell, MA: Kluwer Academic, 2000.

[7] Y.-H. Chang, et al, "GA^{2}CO: Peak temperature estimation of VLSI circuits," *in Proc. Int. SoC Design Conf.*, 2009, pp. 345 - 348.

[8] Y.-C. Chen, et al, "Fast Node Merging with Dont Cares Using Logic Implications," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems,* vol. 29, pp. 1827-1832, Nov. 2010.

[9] R. DeMillo, et al, " Hints on test data selection: Help for the practicing programmer," *IEEE Trans. Computer*, vol. 11, pp. 34 - 41, April 1978.

[10] R. DeMillo, "Test adequacy and program mutation," *International Conf. on Software Engineering*, 1989, pp. 355 - 356.

[11] L. Drucker, "Functional coverage metrics-the next frontier," in *EE times*, Aug. 2002.

[12] A. Gluska, "Coverage-oriented verification of banias," *in Proc. Design Automation Conf.*, 2003, pp. 280 - 285.

[13] D. E. Goldberg, "Genetic algorithm in search, optimization, and machine learning," MA: Addison-Wesley, 1989.

[14] R. Hamlet, "Testing programs with the aid of a compiler," *IEEE Trans. Software Engineering*, vol. SE-3, no. 4, pp. 279 - 290, July 1977.

[15] N. He, et al, "Test-Case Generation for Embedded Simulink via Formal Concept Analysis," *in Proc. Design Automation Conf.*, 2011, pp. 224 - 229.

[16] W. Howden, "Weak mutation testing and completeness of test sets," *IEEE Trans. Software Engineering*, vol. SE-8, pp. 371 - 379, July 1982.

[17] Y. Jia, et al, "An analysis and survey of the development of mutation testing," *IEEE Trans. Software Engineering*, vol. 35, pp. 1 - 32, 2010.

[18] H.-Y. Lin, et al, "A probabilistic analysis method for functional qualification under mutation analysis," *in Proc. Design, Automation and Test in Europe*, 2012, pp. 147 - 152.

[19] J. C. Miller, et al, "Systematic mistake analysis of digital computer programs," *Commun. ACM*, vol. 6, pp. 58 - 63, Feb. 1963.

[20] A. Offutt, "The coupling effect: fact or fiction," *ACM SIGSOFT Software Engineering Notes*, vol. 14, no. 8, pp. 131 - 140, Dec. 1989.

[21] J. Ouyang, et al, "Power optimization for FinFET-based circuits using genetic algorithms," *in Proc. Int. SOC Conf.*, 2008, pp. 211 - 214.

[22] S. Tasiran, et al, "Coverage metrics for functional validation of hardware designs," *IEEE Design and Test of Computers*, vol. 18, no. 4, pp. 36 - 45, July/Aug. 2001.

[23] F. Vargas, et al, "Testability Verification of Embedded Systems Based on Weak Mutation Analysis," *in Proc. Int. Workshop on Testing Embedded Core-Based System-Chips,* 1999.

[24] C.-Y. Wang, et al, "On Automatic Verification Pattern Generation for SoC with Port Order Fault Model," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems,* vol. 21, pp. 466-479, April 2002.

[25] C.-Y. Wang, et al, "An Automorphic Approach to Verification Pattern Generation for SoC Design Verification using Port Order Fault Model," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems,* vol. 21, pp. 1225-1232, Oct. 2002.

[26] C.-Y. Wang, et al, "Automatic Interconnection Rectification for SoC Design Verification based on the port order fault model," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems,* vol. 22, pp. 104-114, Jan. 2003.

[27] C.-Y. Wang, et al, "An AVPG for SoC Design Verification with Port Order Fault Model," *in Proc. Int. Symposium on Circuits and Systems,* pp. 259-262, 2001.

[28] H.-P. Wen, et al, "Simulation-based functional test generation for embedded processors," *IEEE Trans. Computer*, vol. 55, no. 11, pp. 1335 - 1343, Nov. 2006.

[29] S.-C. Wu, et al, "Novel probabilistic combinational equivalence checking," *IEEE Tran. Very Large Scale Integration Systems*, vol. 16, no. 4, pp. 365 - 375, April 2008.