Combining Deterministic and Genetic Approaches for Sequential Circuit Test Generation *

Elizabeth M. Rudnick
Motorola, Incorporated
Austin, TX

Janak H. Patel
Center for Reliable &
High-Performance Computing
University of Illinois, Urbana, IL

Abstract—A hybrid sequential circuit test generator is described which combines deterministic algorithms for fault excitation and propagation with genetic algorithms for state justification. Deterministic procedures for state justification are used if the genetic approach is unsuccessful, to allow for identification of untestable faults and to improve the fault coverage. High fault coverages were obtained for the ISCAS89 benchmark circuits and several additional circuits, and in many cases the results are better than those for purely deterministic approaches.

I. INTRODUCTION

Sequential circuit test generation using deterministic algorithms is highly complex and time consuming [1-7]. Each target fault must be excited and the fault effects propagated to a primary output (PO); the required state must then be justified, and state justification is typically done by reverse time processing. Backtracing is a critical step and is used to determine the component input values required to obtain a particular output value. Handling components other than simple gates is especially difficult because of the backtracing step. Simulation-based test generation has been used to reduce the complexity of test generation. In a simulation-based approach, processing occurs in the forward direction only, and no backtracing is required. Therefore, complex component types are handled more easily. Candidate tests are generated, and a logic or fault simulator is used to select the best test to apply in a given time frame. Several faults are typically targeted simultaneously. Seshu and Freeman [8] first proposed simulation-based test generation, and several simulation-based test generators have since been developed using random [9], weighted random [10-12], and mutation-based [13, 14] pattern generators. Simulation-based test generators which use genetic algorithms (GAs) to generate candidate tests have also been developed [15-18]; very high fault coverages and fast execution times have been reported for several circuits.

*This research was supported by the Semiconductor Research Corporation under Contract SRC 94-DP-109.

A comparison of results for deterministic and GA-based test generators shows that each approach has its own merits. For some circuits, deterministic test generators provide higher fault coverages, while for other circuits, GA-based test generators provide higher fault coverages. The simulation-based approach is particularly well suited for data-dominant circuits, while deterministic test generators are more effective for control-dominant circuits. Untestable faults can be identified by using deterministic algorithms, but significant speedups can be obtained with the genetic approach. Hence, combining the two approaches could be beneficial. A straightforward solution would be to start with the GA-based test generator and then to use a deterministic test generator to improve the fault coverage and to identify untestable faults. Saab's hybrid test generator [19] switches from simulation-based to deterministic test generation when a fixed number of test vectors are generated without improving the fault coverage; simulation-based test generation resumes after a test sequence is obtained from the deterministic procedure. We will explore a different approach which uses deterministic algorithms for fault excitation and propagation, and a GA for state justification. Individual faults in a circuit are targeted, as is normally done in deterministic test generators.

Deterministic algorithms for combinational circuit test generation have proven to be more effective than genetic algorithms [17]. Higher fault coverages are obtained, and the execution time is significantly smaller. A hybrid test generator would then naturally include the deterministic algorithm for fault excitation and propagation within a single time frame. Since we have access to the HITEC [6] source code, we also chose to use the deterministic algorithms for fault propagation in successive time frames. State justification using deterministic algorithms is a much more difficult problem, however, and is prone to many backtracks, which can lead to high execution times. In our hybrid test generator, we use a simulation-based approach for state justification in which candidate sequences evolve over several generations, as controlled by a GA. When a sequence which justifies the desired state is found, execution of the GA terminates. Deterministic procedures for state justification are used only if the genetic approach is unsuccessful.

We begin with a brief description of GAs. An overview of our hybrid approach to test generation is given next, followed by a discussion of the application of GAs to state justification. Results are then presented in Section V for the ISCAS89 sequential benchmark circuits [20] and several synthesized circuits.

II. GENETIC ALGORITHMS

The simple GA, as described by Goldberg [21], contains a population of *strings,* or individuals. Each string is an encoding of a solution to the problem at hand. Each individual has an associated *fitness,* which gives an indication of the quality of the corresponding solution and thus depends on the application. The population is initialized with random strings, and the evolutionary processes of *selection, crossover,* and *mutation* are used to generate an entirely new population from the existing population. This process is repeated for m generations. To generate a new population from the existing one, two individuals are selected, with selection biased toward more highly fit individuals. The two individuals are crossed to create two entirely new individuals, and each character in a new string is mutated with some small mutation probability p. The two new individuals are then placed in the new population, and this process continues until the new generation is entirely filled. At this point, the previous generation can be discarded. In our work, we use tournament selection without replacement and uniform crossover. In *tournament selection without replacement,* two individuals are randomly chosen and removed from the population, and the best is selected; the two individuals are not replaced into the original population until all other individuals have also been removed. In *uniform crossover,* characters from the two parents are swapped with probability 1/2 at each string position in generating the two offspring. A crossover probability of 1 is used; i.e., the two parents are always crossed in generating the two offspring. Also, a mutation probability of 1/64 is used. Because selection is biased toward more highly fit individuals, the average fitness is expected to increase from one generation to the next. However the best individual may appear in any generation, so we save the best individual found.

III. OVERVIEW

Test generation using our hybrid approach is illustrated in Fig. 1. An individual fault in the circuit is targeted. The fault is excited, and required values are backtraced to the primary inputs (PIs) and flip-flops. Next, the fault effects are propagated to a PO, either in the current time frame or in successive time frames. Again, required values are backtraced to the PIs and to flip-flops in time frame zero, in which the fault was excited. If any conflicts are found during fault excitation and propagation, the test generator backtracks to a decision point and makes an alternative choice. Finally the required state in time frame zero is justified by using GAs. Several candidate sequences are simulated, starting from the last state reached after any previous tests have been applied. If a sequence is found which justifies the state, then the sequence is added to the test set, along with the vectors required for fault excitation and propagation. If a sequence cannot be found to justify the desired state, then backtracks are made in the fault propagation phase, and attempts are made to justify the new state.

One drawback to this approach is that untestable faults cannot be identified. Even if a sequence exists which justifies a given state, the GA is not guaranteed to find it. Therefore, deterministic algorithms for state justification are still required in a complete test generator. Hence, our overall approach to test

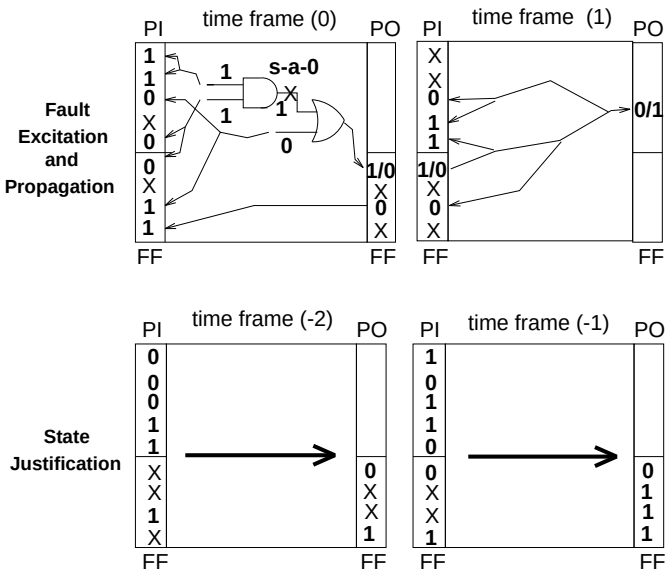

Figure 1: Test generation using GA for state justification.

generation includes both genetic and deterministic approaches for state justification, as indicated in Table I. The test generator makes several passes through the fault list, with different conditions and time limits imposed in each pass. Faults are removed from the fault list once they are detected. After each pass, the user is prompted as to whether to continue with another pass, and execution terminates when the user responds negatively.

In the first pass through the fault list, state justification is performed using a GA. A time limit of one second per fault is imposed, which limits the number of backtracks in the deterministic fault propagation phase. A small population size of 64 is used, and the number of generations is limited to four to reduce the execution time. A sequence length of $\frac{1}{2}x$ is used, where x is supplied by the user. Many of the testable faults are detected in this pass, but untestable faults are identified only if conflicts are found without doing state justification. In the second pass through the fault list, GAs are again used for state justification, but the search space is expanded. In particular, the population size is increased to 128, the number of generations is increased to eight, and the sequence length is doubled. Also, the time limit is increased to 10 seconds per fault to enable more backtracking in the fault propagation phase.

Table I: TEST GENERATION APPROACH

Pass	State Justification Approach	Conditions
1	GA	1-second limit per fault population size = 64 4 generations sequence length = 1/2x
2	GA	10-second limit per fault population size = 128 8 generations sequence length = x
3	deterministic	100-second limit per fault

Finally, deterministic algorithms are used for state justification for any additional passes through the fault list. Required values at the flip-flops are backtraced to the PIs in previous time frames through reverse time processing. An untestable fault is identified when all possible choices at decision points prove unsuccessful in generating a test to detect the fault. The time limit per fault is increased to 100 seconds in the third pass and multiplied by ten in successive passes to expand the search space. In this manner, tests are generated for many of the testable faults by using the GA for state justification. The deterministic algorithms for state justification are used to identify untestable faults and to generate tests for hard-to-detect faults only.

IV. APPLICATION OF GA'S TO STATE JUSTIFICATION

In applying GAs to state justification, we use each string in the population to represent a candidate test sequence. A binary coding is used, and successive vectors in the sequence are placed in adjacent positions along the string. Sequences are evolved over several generations, with the fitness of each individual being a measure of how closely the final state reached matches the desired state. If any sequence is found which produces the desired state, the search is terminated, and the sequence is added to the test set, along with the fault excitation and propagation vectors. Otherwise, the GA runs to completion for a limited number of generations. The test sequence length used is typically a multiple of the sequential depth of the circuit.

A. Fitness Function

Since the fitness of an individual sequence indicates how closely the state it produces matches the desired state, simulation is required. The presence of a particular fault may affect the state; thus, both good and faulty circuit simulations are required for an accurate result. If tests have already been added to the test set, then the current good circuit flip-flop values may already be known. However, the state is not known for the faulty circuit unless a faulty circuit simulation is performed using all previously generated test vectors. Instead of simulating the faulty circuit, we initialize the faulty circuit flip-flops to unknown values. Before the search is begun for a sequence to justify a required state, the desired good circuit state is compared to the current good circuit state, and the desired faulty circuit state is compared to the all-unknown state. (Note that separate values are maintained for the good and faulty circuits during the fault excitation and propagation phases.) If the states match, no justification is required.

If the current state does not match the desired state, then several candidate sequences are simulated for both the good and faulty circuits. Fault injection is performed by modifying the circuit description, as is done in PROOFS [22]; eg., an OR-gate is inserted to simulate a stuck-at-one fault, and the second input of the OR-gate is set to zero for the good circuit and one for the faulty circuit. The bitwise parallelism of the computer word is used, which allows 32 sequences to be simulated in parallel. Two bits are required to represent the three possible logic values: one, zero, and X (unknown). Thus, two computer words are used at each node to simulate the good circuit, and two computer words are used at each node to simulate the faulty circuit. PI values are mapped from the sequences in the GA to the respective bit positions at the PI nodes. Simulation is done in an event-driven manner, with good and faulty circuit simulations done together.

The test sequence length is set to a fixed value, but the state is checked after each test vector is simulated to determine whether it matches the desired state. If it does, the search is terminated. Therefore, the length of the actual test sequence used may be less than the given value. However, for the purposes of the GA, the fitness function measures how closely the final state matches the desired state:

$$fitness = \frac{9}{10}(\#\ matching\ flip\ flops\ in\ good\ circuit) + \frac{1}{10}(\#\ matching\ flip\ flops\ in\ faulty\ circuit).$$

A flip-flop is considered to match if it requires no particular value or if the desired and actual values are equal. If the states match in both the good and faulty circuits, then the fitness will equal the number of flip-flops in the circuit. The two terms in the fitness function correspond to the two goals of the GA: finding a state justification sequence for the good circuit and finding a state justification sequence for the faulty circuit. Unequal weights are used in order that the GA can be targeted to one goal at a time. When a GA has two or more goals, the optimum fitness function does not necessarily weight the goals equally. If equal weights are used, the GA jumps back and forth among the goals, and none of the problems gets solved quickly. A heavy weighting of one goal ensures that the strings evolve steadily in one direction. Experiments on several circuits confirmed that the weights chosen work better than equal weights of 1/2.

Squaring of the fitness function has been used previously to amplify the differences between individuals [21]. Such a measure should be considered if a proportionate selection scheme is used, but since tournament selection is being used in our GA, this operation would have no effect on the result.

B. GA Parameters

Since 32 sequences can be evaluated in parallel, the population size should be a multiple of 32. Initially, we use a small population size of 64 to limit the execution time. We increase it to 128 in the second pass through the fault list, expanding the search space. The number of generations is initially limited to four, again to reduce the execution time. We increase the number of generations to eight in the second pass, when expanding the search space. Tournament selection and uniform crossover are used, since these schemes worked well in simulation-based test generation [18]. Crossover and mutation probabilities of one and 1/64, respectively, are used. Nonoverlapping generations are used, since exploration of the search space is paramount.

V. RESULTS

A hybrid test generator, GA-HITEC, was implemented using the existing HITEC [6] source code and 2700 additional

lines of C++ code. Tests were generated for several of the ISCAS89 sequential benchmark circuits [20] on a SUN SPARCstation 20 with 64 MB memory. Test generation results are shown in Table II. Results for HITEC are shown for comparison. The three lines of results for each circuit correspond to three passes through the fault list with time limits and parameter settings as shown in Table I for GA-HITEC. One exception is that a population size of 32 was used for passes one and two for circuit s35932 to speed up the execution. Test sequence lengths of four and eight times the sequential depth were used in passes one and two, respectively, for all circuits except s5378 and s35932. Test sequence lengths were one-quarter and one-half the sequential depth for these circuits. Higher fault coverages might be obtained with longer test sequences, but the execution time would increase. HITEC also makes several passes through the fault list. The time and backtrack limits are initially set to one second and 10,000 backtracks, respectively, and they are multiplied by ten in each successive pass. Further improvements in fault coverage and untestable fault identification are possible for both GA-HITEC and HITEC if a fourth pass is made through the fault list using a time limit of 1000 seconds per fault; however, execution times would increase. The number of faults detected (**Det**), the number of test vectors generated (**Vec**), the total execution time, and the number of untestable faults identified (**Unt**) at the end of each pass are shown for both GA-HITEC and HITEC.

For many circuits, more faults are detected by GA-HITEC than HITEC at the end of each of the first two passes. In most cases, the GA-HITEC fault coverage at the end of the third pass is greater than or equal to that of HITEC. These results show that the GA is effective in searching for state justification sequences, especially when it is combined with the deterministic approach. In the first two passes, GA-HITEC is able to make use of the current good circuit state, i.e., the state reached after all previous sequences in the test set have been applied. In contrast, HITEC always backtraces to a time frame in which all flip-flops are set to unknown (don't care) values. However, GA-HITEC is not a superset of HITEC. Although GA-HITEC uses the same algorithms as HITEC after the second pass, the HITEC fault coverage is sometimes higher after the third pass. For example, HITEC detects 34,898 faults in circuit s35932 after the third pass, while GA-HITEC detects only 34,862 faults. This discrepancy occurs because the algorithms used in HITEC are partially nondeterministic. Many of the faults are incidentally detected by the test sequences generated; these faults are identified by the fault simulator, and they are never targeted by the test generator. The fault coverage thus depends on the fault list ordering and the time limit imposed on each fault.

While fewer untestable faults are generally identified in the first two passes with GA-HITEC, approximately the same number are identified at the end of the third pass. Note that in some cases, such as circuit s832, HITEC declares some testable faults to be untestable. Comparison of execution times shows that GA-HITEC is faster for some circuits, while HITEC is faster for other circuits. GA-HITEC wastes time targeting untestable faults in the first two passes, a result especially apparent for circuit s386. If these untestable faults can be filtered out in advance, significant speedups can be obtained. The higher HITEC execution time for some circuits is due to the fact that HITEC has lower fault coverage for these circuits and is repeatedly targeting the same testable faults unsuccessfully.

Results of running GA-HITEC on several circuits synthesized from high-level descriptions are shown in Table III. The **Am2910** circuit is a 12-bit microprogram sequencer similar to the one described in [23]; **div** is a 16-bit divider which uses repeated subtraction to perform division; **mult** is a 16-bit two's complement multiplier which uses a shift-and-add algorithm; and **pcont2** is an 8-bit parallel controller used in DSP applications. Test sequence lengths of 24 and 48 were used in the first two passes through the fault lists. For larger circuits, smaller test sequence lengths could be used and the GA parameters could be adjusted to speed up the execution, but lower fault coverages might then be obtained. Results for HITEC are shown for comparison. The two or three lines of results for each circuit correspond to the individual passes through the fault list. GA-HITEC yielded higher fault coverages than HITEC for all four circuits, and the GA-HITEC execution times were also smallest.

VI. CONCLUSIONS

Deterministic algorithms for fault excitation and propagation have been combined with a genetic algorithm for state justification in a new hybrid sequential circuit test generator, GA-HITEC. GA-HITEC makes several passes through the fault list, targeting individual faults, with time limits increasing in successive passes. GAs are used for state justification in the first two passes, while a deterministic algorithm is used in any additional passes. Results for the ISCAS89 benchmark circuits demonstrate the effectiveness of GAs for state justification. Higher fault coverages are obtained for GA-HITEC as compared to HITEC for many circuits. Approximately the same number of untestable faults are identified for the two test generators, and GA-HITEC executes more quickly for many of the circuits. Significant speedups can be obtained for GA-HITEC by identifying untestable faults in a preprocessing step.

While the hybrid test generation approach is effective for benchmark circuits, it may be even more useful for real circuits from industry. Real circuits may impose constraints on the test generator which are difficult to satisfy with deterministic approaches. Backtracing is used during the fault excitation and propagation phases in the hybrid test generator, but processing is restricted to the forward direction during state justification. Thus, constraints are more easily imposed on the test sequences generated.

Finally, this research can be extended to justification of module output values in architectural-level test generation. Backtracing required values through high-level modules is a difficult problem, but a genetic approach could be used in place of traditional approaches to simplify the test generator.

ACKNOWLEDGMENT

The authors would like to thank Prof. David Goldberg for providing several useful suggestions.

Table II: GA-HITEC TEST GENERATION RESULTS

Circuit	Seq Depth	Total Faults	GA-HITEC				HITEC			
			Det	Vec	Time	Unt	Det	Vec	Time	Unt
s298	8	308	255	216	49.5s	0	261	142	41.8s	21
			264	391	5.96m	0	265	281	3.86m	26
			265	415	34.7m	26	265	281	32.3m	26
s344	6	342	327	163	16.6s	0	317	103	20.4s	9
			327	163	2.35m	0	320	119	2.89m	9
			328	169	8.47m	11	324	139	17.6m	11
s349	6	350	334	182	18.6s	2	323	77	21.2s	11
			334	182	2.19m	2	325	85	2.82m	11
			335	188	6.67m	13	334	111	11.5m	13
s382	11	399	73	55	6.89m	0	76	59	6.03m	1
			310	403	22.2m	0	296	1352	25.0m	3
			328	716	2.39h	10	301	1665	3.05h	10
s386	5	384	291	223	1.06m	0	314	275	11.2s	70
			297	295	6.58m	0	314	275	11.2s	70
			314	359	6.70m	70	314	275	11.2s	70
s400	11	426	78	73	6.78m	6	80	98	6.21m	7
			345	566	20.1m	6	337	1356	21.3m	9
			346	704	2.22h	16	342	1669	2.31h	17
s444	11	474	63	77	7.96m	14	85	107	7.24m	16
			370	547	32.7m	14	320	1159	33.1m	17
			381	880	2.62h	25	378	2060	2.84h	25
s526	11	555	67	103	11.4m	1	51	34	10.8m	7
			367	629	44.9m	3	51	34	1.63h	17
			376	873	5.38h	21	346	680	10.7h	22
s641	6	467	404	292	28.5s	41	404	184	6.44s	63
			404	292	3.08m	41	404	184	6.44s	63
			404	292	3.12m	63	404	184	6.44s	63
s713	6	581	476	294	36.5s	82	476	190	9.95s	105
			476	294	3.90m	82	476	190	9.95s	105
			476	294	3.98m	105	476	190	9.95s	105
s820	4	850	460	283	6.80m	0	773	804	2.39m	19
			474	420	51.5m	0	814	1113	4.76m	34
			814	1108	54.2m	36	814	1113	6.01m	36
s832	4	870	446	321	7.05m	14	673	550	4.56m	35
			549	467	48.9m	14	816	1169	7.33m	51
			818	1064	52.5m	52	817	1181	8.72m	53
s1196	4	1242	1238	375	11.1s	3	1239	460	6.34s	3
			1239	377	17.7s	3	1239	460	6.34s	3
s1238	4	1355	1283	409	14.5s	72	1283	469	9.97s	72
s1423	10	1515	787	359	24.4m	9	570	95	19.7m	9
			879	448	2.94h	10	570	95	2.96h	11
			928	477	19.6h	14	776	177	27.5h	14
s1488	5	1486	1132	357	8.52m	0	763	76	14.7m	7
			1235	633	52.2m	0	1438	1085	24.6m	32
			1444	1369	1.03h	41	1444	1138	31.0m	41
s1494	5	1506	1154	384	7.89m	12	1149	336	6.96m	20
			1220	548	54.1m	12	1445	1123	13.2m	45
			1453	1224	1.05h	52	1453	1178	18.3m	52
s5378	36	4603	2993	384	35.8m	74	3238	941	24.6m	128
			3088	497	6.61h	78	3238	941	3.84h	171
			3238	683	39.6h	224	3238	941	36.3h	225
s35932	35	39094	33,427	219	4.01h	3856	34,798	364	1.39h	3856
			33,694	315	12.6h	3984	34,898	439	2.10h	3984
			34,862	425	19.5h	3984	34,898	439	8.07h	3984

Det: # of faults detected Vec: # of test vectors generated Unt: # of untestable faults identified

Table III: GA-HITEC TEST GENERATION RESULTS: SYNTHESIZED CIRCUITS

Circuit	Total Faults	GA-HITEC				HITEC			
		Det	Vec	Time	Unt	Det	Vec	Time	Unt
Am2910	2391	2172	890	3.23m	158	1886	334	8.32m	164
		2181	1119	13.0m	159	2132	624	27.1m	170
		2190	1214	1.11h	173	2166	1286	2.16h	173
div	2147	1717	239	23.8m	136	1677	208	6.93m	136
		1740	356	1.79h	136	1677	208	1.04h	136
		1741	359	9.18h	136	1679	212	10.2h	136
mult	1708	1601	179	4.08m	3	1148	80	13.1m	8
		1633	421	22.8m	3	1351	101	1.21h	14
		1633	421	1.93h	23	1551	122	5.04h	24
pcont2	11300	6757	208	1.33h	2639	3354	7	3.52h	2646
		6757	208	8.51h	2770	3354	7	18.3h	2773

Am2910: 12-bit microprogram sequencer
div: 16-bit divider
mult: 16-bit two's complement multiplier
pcont2: 8-bit parallel controller for DSP applications

Det: number of faults detected
Vec: number of test vectors generated
Unt: number of untestable faults identified

References

[1] R. Marlett, "An effective test generation system for sequential circuits," *Proc. Design Automation Conf.*, pp. 250-256, 1986.

[2] W. -T. Cheng, "The BACK algorithm for sequential test generation," *Proc. Int. Conf. Computer Design,* pp. 66-69, 1988.

[3] H. -K. T. Ma, S. Devadas, A. R. Newton, and A. Sangiovanni-Vincentelli, "Test generation for sequential circuits," *IEEE Trans. Computer-Aided Design,* vol. 7, no. 10, pp. 1081-1093, October 1988.

[4] M. H. Schulz and E. Auth, "Essential: An efficient self-learning test pattern generation algorithm for sequential circuits," *Proc. Int. Test Conf.*, pp. 28-37, 1989.

[5] A. Ghosh, S. Devadas, and A. R. Newton, "Test generation for highly sequential circuits," *Proc. Int. Conf. Computer-Aided Design,* pp. 362-365, 1989.

[6] T. M. Niermann and J. H. Patel, "HITEC: A test generation package for sequential circuits," *Proc. European Conf. Design Automation*, pp. 214-218, 1991.

[7] D. H. Lee and S. M. Reddy, "A new test generation method for sequential circuits," *Proc. Int. Conf. Computer-Aided Design*, pp. 446-449, 1991.

[8] S. Seshu and D. N. Freeman, "The diagnosis of asynchronous sequential switching systems," *IRE Trans. Electronic Computing,* vol. 11, pp. 459-465, August 1962.

[9] M. A. Breuer, "A random and an algorithmic technique for fault detection test generation for sequential circuits," *IEEE Trans. Computers,* vol. 20, no. 11, pp. 1364-1370, November 1971.

[10] H. D. Schnurmann, E. Lindbloom, and R. G. Carpenter, "The weighted random test-pattern generator," *IEEE Trans. Computers,* vol. 24, no. 7, pp. 695-700, July 1975.

[11] R. Lisanke, F. Brglez, A. J. Degeus, and D. Gregory, "Testability-driven random test-pattern generation," *IEEE Trans. Computer-Aided Design,* vol. 6, no. 6, pp. 1082-1087, November 1987.

[12] H.-J. Wunderlich, "Multiple distributions for biased random test patterns," *IEEE Trans. Computer-Aided Design,* vol. 9, no. 6, pp. 584-593, June 1990.

[13] T. J. Snethen, "Simulator-oriented fault test generator," *Proc. Design Automation Conf.,* pp. 88-93, 1977.

[14] V. D. Agrawal, K. T. Cheng, and P. Agrawal, "A directed search method for test generation using a concurrent simulator," *IEEE Trans. Computer-Aided Design,* vol. 8, no. 2, pp. 131-138, February 1989.

[15] D. G. Saab, Y. G. Saab, and J. A. Abraham, "CRIS: A test cultivation program for sequential VLSI circuits," *Proc. Int. Conf. Computer-Aided Design*, pp. 216-219, 1992.

[16] M. Srinivas and L. M. Patnaik, "A simulation-based test generation scheme using genetic algorithms," *Proc. Int. Conf. VLSI Design*, pp. 132-135, 1993.

[17] E. M. Rudnick, J. G. Holm, D. G. Saab, and J. H. Patel, "Application of simple genetic algorithms to sequential circuit test generation," *Proc. European Design and Test Conf.*, pp. 40-45, 1994.

[18] E. M. Rudnick, J. H. Patel, G. S. Greenstein, and T. M. Niermann, "Sequential circuit test generation in a genetic algorithm framework," *Proc. Design Automation Conf.,* pp. 698-704, 1994.

[19] D. G. Saab, Y. G. Saab, and J. A. Abraham, "Iterative [simulation-based genetics + deterministic techniques] = complete ATPG," *Proc. Int. Conf. Computer-Aided Design*, pp. 40-43, 1994.

[20] F. Brglez, D. Bryan, and K. Kozminski, "Combinational profiles of sequential benchmark circuits," *Int. Symposium on Circuits and Systems*, pp. 1929-1934, 1989.

[21] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning,* Reading, MA: Addison-Wesley, 1989.

[22] T. M. Niermann, W. -T. Cheng, and J. H. Patel, "PROOFS: A fast, memory-efficient sequential circuit fault simulator," *IEEE Trans. Computer-Aided Design*, pp. 198-207, February 1992.

[23] Advanced Micro Devices, "The AM2910, a complete 12-bit microprogram sequence controller," in *AMD Data Book*, AMD Inc., Sunnyvale, CA, 1978.