Asian Test Symposium, 2000, pp. 254-259

Embedded Core Testing Using Genetic Algorithms

Ruofan Xu and Michael S. Hsiao ({ruofanxu, mhsiao}@ece.rutgers.edu)
Department of Electrical and Computer Engineering, Rutgers University, Piscataway, NJ

Abstract

Testing of embedded cores is very difficult in SOC (system-on-a-chip), since the core user may not know the gate level implementation of the core, and the controllability and observability of the core are limited by other cores and the user defined logic surrounding the core. One simple but expensive method to solve this problem is to add a wrapper around each core in the SOC, and shift in/out every bit at the core input, output, and possibly its internal state. An approach to remove part of these wrappers using controllability and observability evaluation via random inputs is proposed at the high level (i.e. no gate-level information needed). To achieve better results than the random input vectors, genetic algorithm is used in this paper to justify the test patterns provided by the core designer. Several high level benchmarks are experimented and results show that with the test patterns generated by the genetic algorithm, both the wrapper size and the test application time are further reduced, while the fault coverage of each core is improved.

I Introduction

The design time for today's chips is becoming longer due to the increasing size and complexity of circuits. To cut down the design time, third party cores can be used, resulting in what is known as a system-on-a-chip (SOC) design. However, since the gate level information is not available to the core user, conventional ATPG is not applicable, and testing of these cores becomes a challenging task [1, 2].

One basic approach to solve this problem is to route primary inputs and primary outputs of the SOC to core's I/O's using multiplexers [3]. But this scheme is limited by the number of primary I/O's of the SOC against the number of core I/O's and may not help testing the UDL (User Defined Logic) around the core. Another solution is the use of isolation ring, or the wrapper [4–7]. In using isolation ring, a flip-flop is added to each core input and output, and all the added flip-flops are connected in series. In this way, with only a few entries from the SOC primary I/O, we can route the PI's to the wrapper and provide all the test vectors needed by the core and shift out all the results generated by the core. The disadvantage of this technique is that it will result in high hardware and test application time overhead. Another possibility for this is to use a TAMBUS, which is a bus routed to every core I/O's in the chip [6, 8, 9]. Similar to the wrapper, TAMBUS can provide full controllability and observability to all the cores. However, the width of the TAMBUS may be limited to fully covered cores with large number of I/O's.

In a recent work [10], a method is developed to evaluate the controllability and observability for core inputs and outputs at the high level (i.e. no gate-level information needed). Based on these evaluations, some of the core I/O's can be quickly removed from the wrapper as well as from the TAMBUS, so that both the area and performance overhead will be reduced. Furthermore, it is made sure that the resulting cores in the SOC are still fully testable by performing a validation check.

Since these testability evaluations are all based on pseudo random input vectors, the fault coverages may result in an decrease (up to 10%) after the removal of the wrappers for some cores that are hard to reach. To avoid this, we propose to generate intelligent patterns by a genetic algorithm instead. In doing so, justification of core vectors is no longer a hit-or-miss situation; rather, we are able to intelligently search for each justification vector. As a result, both the wrapper size and test application time are reduced, with the fault coverages for the core's improved.

The remainder of this paper is organized as follows. Section II gives the preliminaries on controllability and observability evaluation. Section III shows how to generate a test set which can justify as many test patterns of each core in the SOC as possible using genetic algorithms. We validate this approach in section IV by getting the fault coverage of each core. In section V we present the results of both the evaluation and the validation. Finally, we conclude the paper in Section VI.

II Preliminaries

When full wrappers are used in a SOC, the size of a full wrapper is the number of inputs plus the number of outputs of the corresponding core. The test application time and area overhead are proportional to the size of core I/O's. So if we can justify the entire test set for a core from the primary inputs of the SOC and can observe the core outputs through some existing logic to the primary

*This research was supported in part by the NJ Commission on Science and Technology under contract 00-2042-007-14.

outputs of the SOC, then the wrapper would not be necessary, resulting in both test application time and area reduction. For example, in Figure 1, if we can justify the test set for core 2 via core 1, and if we can propagate any faulty value generated at the output of core 2 through core 3 or core 5, we can eliminate the wrapper around core 2.

But in cases that are not as ideal, we may be able to justify only a part of the whole test set for the core under test, or we may be able to observe only some core outputs at the SOC's output. In such cases we cannot remove the entire wrapper, but we may still be able to remove part of the full wrapper, which may still save some test application time and area, depending on how much can we remove.

To reduce the size of the full wrapper, we must maintain the testability for both the core itself and the UDL around the core. To achieve this, we evaluate the core I/O's in two perspectives. From the input point of view, we evaluate core inputs by their controllability. From the output point of view, we evaluate core outputs by their observability, both at the high level. During our evaluation, we assume that there are multiple cores in the SOC and evaluation is performed for one core at a time. So at any time there is only one core under test, (shaded in Figures 1 and 3). We also assume that the functionality of the cores are available, which is reasonable because the SOC integrater has to know how the cores behave in order to use them properly. We assume all the cores are fully scanned. Finally, for both the UDL around the core and the core-under-test, only the RTL information is needed in our approach. No gate level information for either the UDL or the core under test is required.

A Controllability evaluation

For each core i in the SOC, let us denote the designer-provided test set T_i. When evaluating controllability of inputs of core i (core 2 in Figure 1), we apply the set of test vectors from the primary inputs of the SOC, and obtain a set of vectors that we can justify at the input of core i. We call these vectors at the inputs of core i, S_i. For any partial wrapper of size n, we denote the corresponding n input bits in the wrapper as $I = \{I_1, I_2, ..., I_n\}$. An expanded set E_i (derived from S_i) is used to check whether all the vectors in T_i can be justified. The expanded set E_i is built by replacing $I_1, I_2, ..., I_n$ of each vector in S_i with don't-cares, while leaving other bits unchanged. Then we check whether E_i can cover every vector in T_i by a greedy approach. If it can, then this partial wrapper can justify the test set and more input bits may be possibly removed from it. If it cannot, more input bits should be added to the partial wrapper until the test set can be justified. This process is repeated until a minimal partial wrapper is found. For example, let us consider the designer-provided test set is $T_i = \{000, 010, 111\}$, and the vector set we can

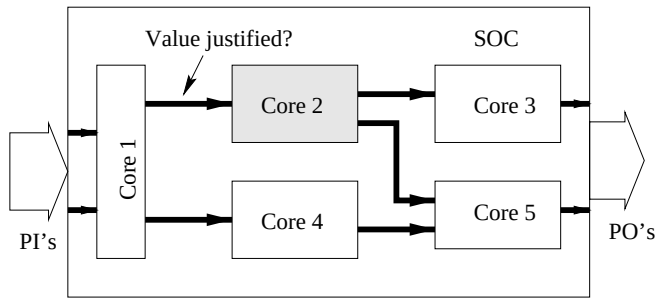

Figure 1: Controllability Evaluation.

justify at the input of core i directly from the SOC inputs is $S_i = \{010, 011\}$. Thus, two vectors in T_i cannot be fully justified: $\{000, 111\}$. For a partial wrapper which includes only the first bit ($I = \{I_1\}$), we would get the expanded set $E_i = \{x10, x11\}$. This set can not cover T_i since the vector 000 is not included in E_i. So we expand the partial wrapper to $I = \{I_1, I_2\}$, which contains the first two bits. Now E_i becomes $\{xx0, xx1\}$, which can fully cover all the vectors in T_i. This process is repeated until no more expansion or reduction is available. The resulting partial wrapper would be the minimal partial input wrapper. In our example, we have reduced the full wrapper of size 3 to only 2 bits. The algorithm for finding the minimal partial wrapper I without reducing the controllability of the core is shown in Figure 2.

```
Set I = all core inputs
while (I is still reducible) {
    set E_i = φ;
    for each vector v in S_i {
        for j from 1 to n
            set the I_j^th bit in v = don't-care;
        put v in E_i;
    }
    if (E_i can cover T_i)
        reduce I;
    else
        expand I;
}
```

Figure 2: Controllability Evaluation

B Observability evaluation

The aim of our observability evaluation is to identify a maximal set of core outputs that can be easily observed at the primary outputs. The basic idea is to try to propagate the faulty values of the core to the primary output of the SOC [11]. To get the evaluation for every single output of the core, each output must be evaluated separately.

Since the gate level information is not available for the core, its exact gate-level faulty values are unknown. So we use random values as the faulty output values (output of core 2 in Figure 3); that is, we assign a random value to

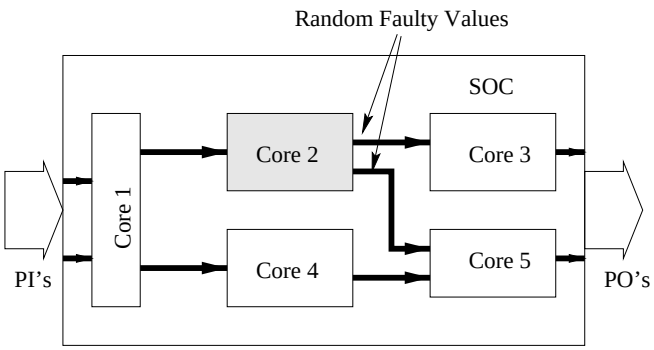

Figure 3: Observability Evaluation.

our current processing core output and then apply high-level fault simulation to the SOC [11].

If the faulty values can be observed at the primary output or any of the partial wrapper, which are different from the fault free values, we assume this output can be easily observed and can be potentially removed from the full-size wrapper. For instance, if a number of random faulty values placed at the output bits of core 2 in Figure 3 can be observed at the primary outputs of the SOC, then we can remove these bits from the output wrapper of core2 and assume that they are fully observable. On the other hand, if no faulty value could be observed after simulating all the vectors in a given test set, this output is assumed to be hard to observe and is left in the wrapper. Figure 4 shows our method of observability evaluation.

```
For each output bit in core i {
    simulate a given test set
    and for each vector {
        set a random faulty value to the output bit;
        high-level simulate to the rest of the ckt;
        if (faulty value observed) {
            remove this bit from the wrapper;
            stop and go to evaluate the next bit;
        }
    }
    // faulty value cannot be observed
    leave this bit in the wrapper;
}
```

Figure 4: Observability Evaluation

C Genetic algorithms

Genetic algorithm (GA) is an adaptive method which can be used to solve search and optimization problems. In this paper it is used to justify the test patterns provided by the core designer from the primary input of the SOC. The GA framework used in the paper is similar to the simple GA described in Goldberg [12]. The GA contains a population of strings, also called chromosomes or individuals, in which each individual represents a test vector at the primary input of the SOC. A binary coding is used, and therefore, each character in a string represents the logic value to be applied to the corresponding PI. The population size used depends on the number of PI's. Larger populations are needed to accommodate larger number of PI's in order to maintain diversity.

Each individual has an associated fitness, which measures the test vector quality in terms of the number of bits justified at the input of the core under test. The population is initialized with random strings. A high level simulator is used to compute the fitness of each individual by measuring the Hamming distance of obtained input vector for the core with the desired vector. Then the evolutionary processes of selection, crossover, and mutation are used to generate an entirely new population from the existing population. Two individuals are selected from the existing population, with selection biased toward more highly fit individuals. The two individuals are crossed by randomly swapping bits between them to create two entirely new individuals, and each character in a new string is mutated with some small mutation probability. The two new individuals are then placed in the new population, and this process continues until the new generation is entirely filled. Evolution from one generation to the next is continued until a test vector found to fully justify the target test pattern or until a maximum number of generations is reached. 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.

III Justifying Test Patterns Using GA's

In testing cores in SOC's, a major problem is how to justify the test set provided by the core designer at the input of the core if full-wrapper was not used. Because there may be multiple cores in a SOC and the gate level implementation of each core is not available to the SOC designer, conventional way of justifying the test patterns via backtracing is not possible. Genetic algorithms on the other hand, are used to avoid the backtracing and involve forward simulation only.

In our controllability evaluation procedure, we want to remove as many core inputs as possible from the full wrappers. The number of inputs that can be removed is highly dependent on the number of test patterns we can justify, and, for those test patterns that cannot be fully justified, the number of bits that differs between the justified vector and the target test pattern.

Based on these facts, we select Hamming distance between the test pattern for the core and the vector justified at the core input as the fitness in our GA process. And a smaller Hamming distance implies a higher fitness. For example, if a target test pattern for the core under test is {10101}, and when applying a certain vector at the pri-

mary input, we get a {11010} at the input of the core, then the Hamming distance between these two vectors is 4 (bit positions 2, 3, 4, and 5 differ).

The population size in our GA process is set equal to $4 \times \sqrt{\#PI's}$, i.e. a function of the number of primary inputs. Thus for SOC's with a large number of inputs, there would be a larger population to maintain the diversity. An additional technique to maintain diversity is to mutate each bit in the offspring vectors with a certain mutation probability, which is 1% in our process.

To generate a new population, first two pairs of vectors from the current population are selected randomly. In each pair the vector with higher fitness (i.e. less Hamming distant) is selected as the parent vector. Then the offspring vectors are generated by these two parent vectors and added to the new population. This process is repeated until the new population is completely filled. Figure 5 shows the procedure to generate a test set for the entire SOC.

```
For each core in the SOC {
  For each test patter V_i of the core under test {
    Generate a vector P_i at the SOC PI's using GA's
    Add P_i to P;
  }
}
```

Figure 5: Test generation using GA

As shown in Figure 5, for a given test set at the core input ($\{V_1, V_2, ..., V_s\}$ at the input of Core B in Figure 6), we need to derive a set of vectors that can justify the input vector set $\{V_1, V_2, ..., V_s\}$ for Core B. The GA is called to justify each test pattern V_i. The resulting vectors P_i ($\{P_1, P_2, ..., P_s\}$ in Figure 6) are recorded as the test set P that we are going to use in our controllability and observability evaluation. Compared to the random vectors generated to try to justify the core vectors, this new test set derived from GA can greatly reduce the number of bits necessary to shift in via the partial input wrapper of each core. Consequently, the test application time is significantly reduced.

IV Validation

In order to validate our results, we must show that after those input and output bits are removed from the wrapper, the testability of each core is not reduced. To do this, we take gate-level descriptions of the cores being considered and use a gate level fault simulator and perform fault simulation within the context of SOC. In doing so, we can assess the exact fault coverage of every core before and after the test enhancement.

As described previously, we assume only one core (the core under validation) is faulty at a time and all other cores are fault free. The application of the test set obtained from Section III at the SOC primary inputs will result in attaining input bits not included in the wrapper for the core-under-test. Then, for the inputs included in the wrapper, we shift in corresponding values to match the expected test vector provided by the core vendor. Based on our controllability evaluation method, all the test patterns specified by the core vendor can be justified this way. Next, we perform fault simulation for the core in the context of the SOC. The fault is detected if its faulty value is observed at either the primary output or at any of the partial output wrappers.

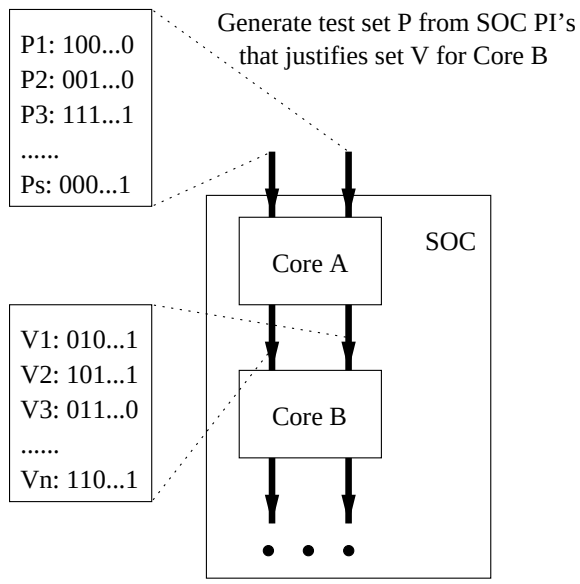

Figure 6: Justify Vectors for Core B.

```
for each fault in core i {
    apply the test set obtained from Section III;
    for each vector {
        if (fault is detected) {
            DetectedFaults++;
            break;
        }
        shift in data from the partial input wrapper
        if (faulty value observed) {
            DetectedFaults++;
            break;
        }
    }
}
```

$Fault\ Coverage = \frac{DetectedFaults}{Total\ Faults}$

Figure 7: Validation Algorithm

Since we are injecting single stuck-at-faults into the core, we use the gate level description of each core during our validation process. However, this information is only used when we attempt to show that our evaluation is effective. It is not necessary in our evaluation process.

High-level fault simulation may be used in place of gate-level fault simulation. Figure 7 shows the algorithm.

V Experimental Results

We implemented our algorithm in C++, and experiments were conducted on a Sun Ultra 10 workstation with 256MB of RAM for two synthesized benchmark circuits and two manually constructed SOC's. The manual SOC's are constructed by combining several ISCAS85 and ISCAS89 [13] benchmark circuits (Figures 8 and 9). The two synthesized benchmarks are am2910 and divckt. Am2910 is a microprogram address sequencer, and divckt is a 16-bit divider.

The results are shown in Table 1. The first two columns give the names of the SOC's and the cores. For each core, the number of inputs and outputs are listed in the third and fourth columns. Column 5 shows the size of the test set provided by the core designer. The next two columns are the results of our controllability evaluation, using random primary inputs and the test set given by the GA process, respectively. Column 8 gives the result of our observability evaluation, i.e. the size of the partial output wrapper for each core. After the easily controllable and easily observable core I/O's are removed from the full size wrappers, the size of the partial wrapper in terms of the percentage of the full wrapper is shown in columns 9 and 10 for the random and GA approaches, respectively. The total number of faults of each core is listed in column 11. Finally, the last two columns give the gate-level fault coverages with the full size wrappers and the partial wrappers.

For example, for the core s820 in the manually constructed SOC2, there are 18 inputs and 19 outputs, and 187 vectors in the provided test set. Using the random inputs, the resulting partial input wrapper size is 14. When genetic algorithms are applied, the size of the partial input wrapper is reduced to 0, which implies all the test vectors for this core can be justified directly from the SOC inputs, and thus all the inputs of this core can be removed from the wrapper. The size of the partial output wrapper for this core is 8, which indicates 11 outputs are observable either at the SOC outputs or at the wrapper of some subsequent cores, and thus they can be removed from the wrapper. After the removal, (14+8)/(18+19) = 59.5% of all the core I/O's remained in the wrapper if using random inputs, while only 21.6% would remain if GA's were used. The total number of faults in s820 is 850. The original fault coverage using full wrappers was 100%. When the partial wrappers were used, the fault coverage drops to 98.82%, slightly lower than the original. The slight drop in fault coverage is not due to failing to justify the core vectors, but due to the optimism in propagation of faulty effects to SOC outputs. Another example is for the core REGCNT in benchmark am2910, 6 out of 15 core inputs and 8 out of 12 core outputs are needed in the partial wrappers using random inputs. When GA's are used, only 2 bits are needed in the input wrapper, which further reduced the total wrapper size from 51.9% of the full size to only 37.0%. With the partial wrappers, the fault coverage drops slightly to 88.80% from the original 89.30%. Similar results were observed for other cores in various SOC's as well.

It is interesting to note that for cores s208 and s510 of SOC1, no wrapper was needed at all using the GA approach, resulting in a 72.7% and 61.5% wrapper reduction when compared to the random approach. In addition, the resulting fault coverages remained 100% even without any wrapper. Finally, the execution time of the GA process

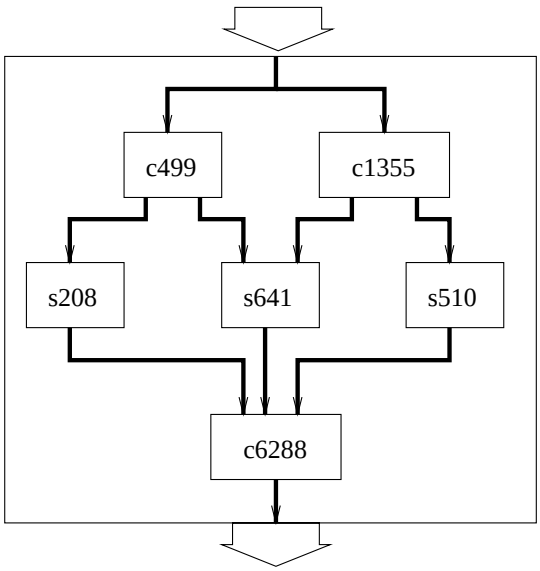

Figure 8: Manually Constructed SOC1.

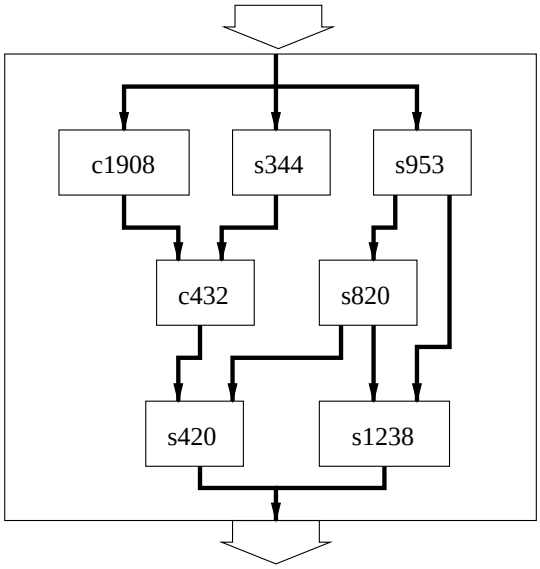

Figure 9: Manually Constructed SOC2.

Table 1: Results

SOC	Core	# core inputs	# core outputs	# vec	Input Wrapper		Output Wrapper	Final Wrapper Size (%)		Fault Coverage (%)	
					RND	GA		RND	GA	Full	Partial
am2910	REGCNT	15	12	46	6	2	8	51.9	37.0	89.30	88.80
	MUX4	51	12	37	41	27	1	66.6	44.4	100	100
	Cntrl	7	10	37	1	1	2	17.6	17.6	94.95	94.95
divckt	MUX16A	33	16	16	23	8	11	69.4	38.8	100	100
	ALU16	38	17	52	29	3	1	54.5	7.2	98.32	94.45
	Cntrl_div	3	6	8	1	1	2	33.3	33.3	100	100
SOC1	c499	41	32	57	0	0	18	24.7	24.7	98.94	98.94
	c1355	41	32	91	0	0	3	4.1	4.1	99.49	99.49
	s208	10	1	39	8	0	0	72.7	0	100	100
	s510	19	7	69	16	0	0	61.5	0	100	100
	s641	35	24	68	26	6	0	44.1	10.2	100	100
	c6288	32	32	40	25	21	0	39.1	32.8	99.56	99.56
SOC2	c1908	33	25	119	0	0	13	22.4	22.4	99.47	99.47
	c432	36	7	49	27	19	5	74.4	55.8	99.24	99.24
	s344	9	11	30	0	0	5	25	25	100	100
	s953	16	23	119	0	0	12	30.8	30.8	100	100
	s820	18	19	187	14	0	8	59.5	21.6	100	98.82
	s420	18	1	84	16	14	0	84.2	73.7	98.24	98.24
	s1238	14	14	205	12	8	0	42.9	28.6	94.76	94.76

for each SOC was less than 20 minutes, and the evaluation time for each core in any SOC was less than 5 minutes.

VI Conclusion

From this work we observe that there are many easily controllable and easily observable I/O's in most cores. After they are removed from the full size wrapper, the total area and test application time overhead for the DFT can be greatly reduced. Compared to the random justification approach, the GA's can justify many more test patterns directly at the core inputs, and thus further decrease the overhead. The validation results show that with the partial wrappers, the fault coverages of the cores remain the same for most cores, and in a few cases, a slight drop in fault coverage resulted.

References

[1] S. Bhatia, T. Gheewala and P. Varma, "A unifying methodology for intellectual property and custom logic testing," *Proc. Intl. Test Conf.,* pp. 639-648, 1996.

[2] K. De, "Test methodology for embedded cores which protects intellectual property," *Proc. VLSI Test Symp.,* pp. 2-9, 1997.

[3] V. Immaneni and S. Raman, "Direct Access Test Scheme-Design of Block and Core Cells for Embedded ASICS," *Proc. Intl. Test Conf.,* pp. 448-492, 1990.

[4] B. Pouya and N. A. Touba, "Modifying user-defined logic for test access to embedded cores," *Proc. Intl. Test Conf.,* pp. 60-68, 1997.

[5] N. A. Touba and B. Pouya, "Testing core-based designs using partial isolation ring," *Proc. VLSI Test Symp.,* pp. 10-16, 1997.

[6] Y. Zorian, E. J. Marinissen, and S. Dey, "Testing embedded-core-based system chips," *IEEE Computer,* June 1999, pp. 52-60.

[7] I. Ghosh, N.K. Jha, S. Dey, "A low overhead design for testability and test generation technique for core-based systems," *Proc. Intl. Test Conf.,* pp. 50-59, 1997.

[8] Y. Zorian, "Test requirements for embedded core-based systems and IEEE P1500," *Proc. Intl. Test Conf.,* pp. 191-199, 1997.

[9] E. J. Marinissen, Y. Zorian, R. Kapur, T. Taylor and L. Whetsel, "Towards a standard for embedded core test: an example." *Proc. Intl. Test Conf.,*pp. 616-627, 1999.

[10] R. Xu and M. Hsiao, "Evaluation for controllability and observability of embedded cores in SOC," *Intl. Test Synthesis Wkshop,* 2000.

[11] M. S. Hsiao and J. H. Patel, "A new architectural-level fault simulation using propagation prediction of grouped fault-effects," *Proc. Intl. Conf. Computer Design,* pp. 628-635, 1995.

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

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