Intelligent Random Vector Generator Based on Probability Analysis of Circuit Structure

Yu-Min Kuo, Cheng-Hung Lin, Chun-Yao Wang, Shih-Chieh Chang, and Pei-Hsin Ho*
*Department of CS, National Tsing Hua University, Hsinchu, Taiwan, and *Synopsys, Inc.*

Abstract

Design verification has become a bottleneck of modern designs. Recently, simulation-based random verification has attracted a lot of interests due to its effectiveness in uncovering obscure bugs. Designers are often required to provide the input probabilities while conducting the random verification. However, it is extremely difficult for designers to provide accurate input probabilities. In this paper, we propose an iterative algorithm that derives good input probabilities so that the design intent can be exercised effectively for functional verification. We conduct extensive experiments on both benchmark circuit and industrial designs. The experimental results are very promising.

1. Introduction

With the exponential growth of design complexity, verification has become a bottleneck of modern designs. About 70% of project effort for complex ICs is spent on verification [5]. Among verification techniques, simulation-based verification remains one of the most important techniques to uncover design errors. Normally, verification engineers need to manually write testbenches. However, the manually-writing process is not only time-consuming but also error-prone. Recently, random verification has attracted a lot of interests because randomly generated vectors may uncover some obscure bugs which are not easy to be discovered by designers. It was reported that there are more than three-quarters of the bugs were found using pseudo-random techniques [8]. Many industrial companies [11][14][15] and researches [1][4][7] have demonstrated the success of applying biased random verification.

The random verification requires careful implementation of a simulation environment. First, proper input probabilities must be provided to generate quality random vectors. Ineffective input probabilities may cause long simulation time with poor coverage. Secondly, because inputs to a design are often correlated, some impossible or illegal input sequences should not be applied at a circuit's inputs; otherwise, the design under verification (DUV) may enter an unexpected situation. Normally, to prevent illegal vectors, constraint equations are provided by designers and randomly generated vectors must satisfy those equations. Several previous works [2][3][6][12][13] attempt to solve the constraint problem.

Since the effectiveness of random verification is directly affected by the input probabilities applied at inputs, as far as we know, most previous works assume that input probabilities are provided by designers. In this paper, we propose a novel way to automatically determine effective input probabilities so that as many states as possible are visited. Our method backward calculates from outputs as an initial input probability assignment and iteratively refines this assignment to a better one based on the input/output relations of probabilities. First, we propose efficient methods for a combinational circuit so that many combinational outputs can be reached by few random input patterns. Then, we extend these methods to sequential circuits by generating good state-dependent probabilities of inputs. We have performed our methods on a large set of MCNC and ISCAS-89 benchmarks, a public domain TV80 microprocessor core, and an industrial AES design. The experimental results show that our results are significantly better than the random verification.

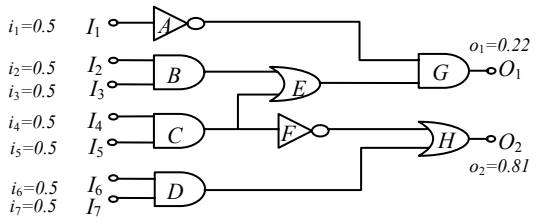
We would like to mention that although ATPG has provided an effective method to generate vectors for combinational circuits, it is still considered to be difficult for sequential circuits. The difficulty is relevant to the huge state space, which causes the explosion problem when employing the timeframe expansion technique. Therefore, it may not easily generate input vectors to reach all possible states or state transitions. An alternative to ATPG is the random verification, which serves as the important role to compensate the insufficiency problems.

The remainder of this paper is organized as follows. Section 2 discusses the overview of our approach. Section 3 provides the cost function for determining input probabilities of combinational circuits and describes the algorithm for combinational circuits. Section 4 augments the discussion on dealing with sequential circuits. Section 5 shows the experimental results. Section 6 concludes this paper.

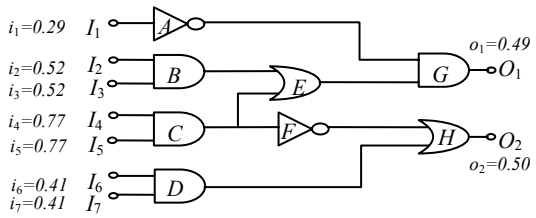
2. An Example of Effective Input Probabilities

In this section, we illustrate how input probability can affect the efficiency of random verification. Throughout this paper, we use the upper-case symbol (character) to represent a gate/wire and use the lower-case symbol (character) to represent the 1's probability of the gate/wire. Consider the example of applying probabilities at inputs in Figure 1. A random verification is said to be *uniform random* if the probability of each primary input is 0.5 as in Figure 1(a). Given the uniform random as input probabilities, One can find that we have output probabilities $o_1 = 0.22$, and $o_2 = 0.81$. On the other hand, if input probabilities are re-assigned to the values as in Figure 1(b), we have output probabilities $o_1 = 0.49$,

and o_2 = 0.50. If our objective is to reach as many output combinations as possible using random verification in a given period of time, it is very likely that the probability assignments of inputs in Figure 1(b) are superior to those in Figure 1(a). This is due to the fact that the probabilities of all outputs are close to 0.5 in Figure 1(b) so the chances of reaching all output combinations are balanced. Therefore, by smartly biasing input probabilities, it is possible to improve the effectiveness of random verification.



(a) Uniform random verification



(b) Biased random verification

Figure 1: Example of the probability assignments

3. Generation of Input Probabilities for Combinational Circuits

Remember that our goal is to derive input probabilities to reach as many states as possible in a short period of time for a sequential circuit. In this section, we first show a technique to find input probabilities to reach many output combinations for a combinational circuit and later extend the technique to sequential circuits.

3.1 Evaluation of Input Probabilities

To derive a good input probability, it is important to determine whether a set of input probabilities is better than another set. Intuitively, if all output probabilities are close to 0.5, the effectiveness of random verification is better as that in Figure 1. Given a set of output probabilities, we use a cost function called *random_quality* to evaluate effectiveness.

$$random\_quality = \sum_{i=1}^{|PO|} (po_i - 0.5)^2$$

Consider Figure 1(a). The random_quality of the output probabilities is $(o_1–0.5)^2 + (o_2–0.5)^2 = 0.1745$. Similarly, the random_quality of the output probabilities in Figure 1(b) is 0.0001. As a result, we conclude that output probabilities in Figure 1(b) are more effective than those in Figure 1(a).

With this cost function in mind, our new objective is to find input probabilities which derive output probabilities that minimize the cost value. We would like to mention that it can be difficult to compute exact output probabilities given a set of input probabilities. However, there exist fast estimation techniques [9][10] which obtain the output probabilities with bounded signal relations of the circuit.

In the following, we describe two heuristics to achieve as small random_quality as possible. The algorithm in Section 3.2 attempts to find a good initial solution while the algorithm in Section 3.3 iteratively improves the previous solution.

3.2 Backward Method

Let us first consider a tree-structure circuit where all internal signals are independent. Our algorithm starts with the probability of 0.5 for each output and attempts to propagate the value from outputs to inputs. During the propagation, there are many possible ways to assign input probabilities. The assignments attempt to balance probabilities among all its direct inputs. This avoids the situation that certain nodes with extremely low or high input probabilities. We use the rules described below for probability assignments. Given the output signal probability p_o, we would like to find its input probabilities p_i. We assign $p_i=(p_o)^{1/k}$ for a k-input AND gate, $p_i=1–(1–p_o)^{1/k}$ for a k-input OR gate, and $p_i=1–p_o$ for an INV gate. The assignments can be easily verified assuming inputs are uncorrelated. For example in Figure 2, assume OUT has a probability of 0.5. Using the above rules, we assign the probabilities of node *A* and node *B* to be $a = b = 1–(1–0.5)^{1/2} = 0.71$. With the probability of a = 0.71, we can have $i_1 = i_2 = (0.71)^{1/2} = 0.46$ and similarly, we have $i_3 = i_4 = 0.84$.

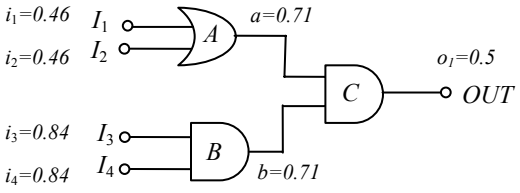


Figure 2: Backward propagation of probability on a tree-structure circuit

We now consider a general structure circuit where internal nodes may have multiple fanouts. We estimate the probability of a multi-path input by averaging the probabilities from all its fanouts. Consider the example in Figure 3 which is similar to the one in Figure 2 except that input I_2 has two fanout edges $I_2 \rightarrow A$ and $I_2 \rightarrow B$. The probability of I_2, i_2, is computed to be (0.46 + 0.84) / 2 = 0.65.

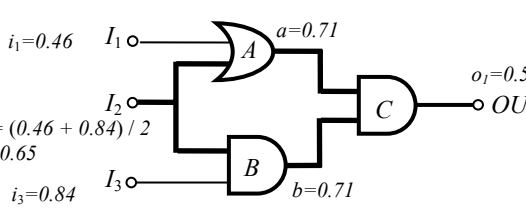


Figure 3: Backward propagation of probability on a general structure circuit

The backward_assignment method attempts to derive input probabilities so that output probabilities are 0.5. However, the method may not result in good output probabilities for some cases. For the example in Figure 4, one can find the output probability is 0.45. This mismatch mainly comes from the averaging heuristic at multi-fanout node.

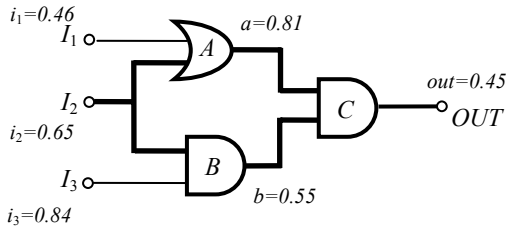


Figure 4: Disadvantage of backward method of which is caused by probability averaging

The above procedure is referred as the *backward_assignment* method and its procedure is summarized in Figure 5.

For each primary output {
Set target probability of this primary output, 0.5;
Propagate probability inversely from the primary output to primary inputs;
Record the signal probability assignments computed for the primary inputs;
}
For each primary input, we assign a probability which is the average of all signal probability assignments.

Figure 5: Pseudo code of the backward_assignment method

3.3 Refinement Method

In this section, we discuss a method called *iterative_refinement* method to iteratively refine input probabilities. First, we give a formulation which describes how a minor change in an input probability may affect its output probabilities. With this formulation, we then derive efficient algorithms to gradually modify input probabilities for better random_quality of output probabilities.

Let us consider a 2-input AND gate whose output is *OUT* and whose inputs are *X* and *Y*. Assume input probabilities are x and y respectively. The output probability is $out = x \times y$. If the probability x changes to $x+\triangle x$ where $\triangle x$ indicates a minor change of x, we can obtain the probability of *OUT*,

$$out = (x+\triangle x)\times y = x\times y+\triangle x\times y.$$

From this equation, we can find that the difference of $\triangle x$ in *X* will cause the difference of $\triangle x\times y$ in *OUT*. We call the value of $\triangle x\times y$ to be the *delta_adjustment* of *OUT*. Similarly for a 2-input OR gate, assume input probabilities are x and y. We have $out = 1-(1-x)\times(1-y) = x+y-x\times y$. If we increase probability $\triangle x$ to *X*, we can get

$$out = 1-(1-(x+\triangle x))\times(1-y)$$
$$= x+y-x\times y+(1-y)\times\triangle x.$$

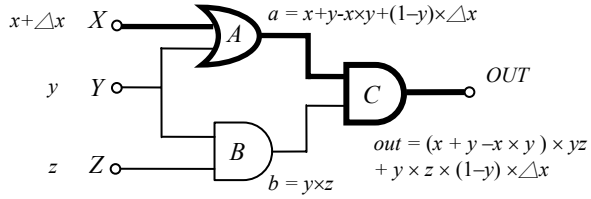
That is, the difference of $\triangle x$ in *X* will cause the difference of $(1-y)\times\triangle x$ in *OUT*.

We now extend to the case when a set of gates are in series. We illustrate our basic idea by using the example in Figure 6. Let us consider the bold path {*X, A, C*} in Figure 6(a). Suppose we would like to know how a change of probability in input *X* may affect the output probabilities. We can find $\triangle a = (1-y)\times\triangle x$ and $\triangle c = b\times\triangle a$. Since we have $b = y\times z$, the difference of c is equal to $\triangle c = y\times z\times(1-y)\times\triangle x$. In other words, if there are gates in series, the modification of output probability is equal to the multiplication of delta_adjustment of gates along the path.

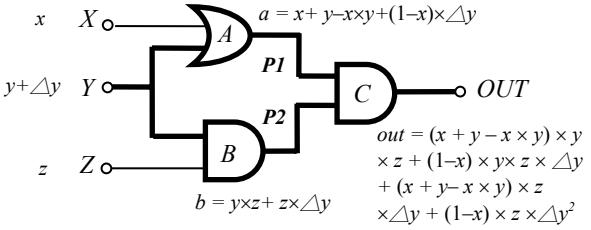
In general, there are many paths from an input to an output. We use the superposition to sum up all the delta_adjustments of paths. Let us consider the same example in Figure 6(b). There are two paths *P1*={*Y, A, C*} and *P2*={*Y, B, C*} from *Y* to *OUT*. From path *P1*, the delta_adjustment is $\triangle y\times(1-x)\times y\times z$ and from path *P2*, the delta_adjustment is $\triangle y\times(x+y-x\times y)\times z$. Using the superposition on these two paths, we obtain a first-order approximation of the delta_adjustment which is

$$\triangle out \cong \triangle y \times \{(1-x)\times y\times z + (x+y-x\times y)\times z\}.$$

In fact, the above approximation ignores the effect of higher order terms of $\triangle y$.



(a) Single path from output to input



(b) Multiple paths from output to input

Figure 6: Delta_adjustment calculation

Consider the example in Figure 7. There are two paths from *Y* to *OUT* and the delta_adjustment of path *P1* is $(1-x)\times y\times z = (1-0.46)\times 0.65\times 0.84 = 0.30$ and the delta_adjustment of path *P2* is $(x+y-x\times y)\times z = (0.46+0.65-0.46\times 0.65)\times 0.84 = 0.81\times 0.84 = 0.68$. Because *Y* has two paths, the resulting delta_adjustment of *Y* is 0.30+0.68 = 0.98. Therefore, if $\triangle y$ is small, we can re-write the equation to be $\triangle out \cong \triangle y\times 0.98$. In Figure 7, to raise the probability *out* to 0.5, we must have $\triangle out$ to be 0.05. From this equation, we know that if we want to increase $\triangle out$ by 0.05, the $\triangle y$ must be equal to 0.05 / 0.98 ~ 0.05. After the refinement, the new y becomes 0.701 (= 0.65+0.05). One can find that the new

input probabilities allow us to have the probability of *OUT* to become 0.5.

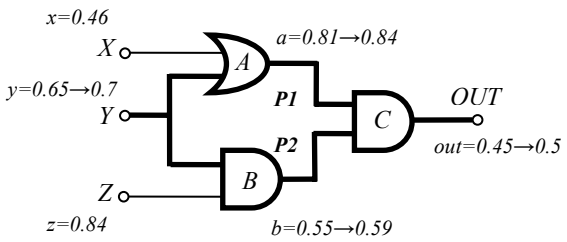


Figure 7: Probabilities after refinement

The iterative_refinement method is summarized in Figure 8.

```
While (1) {
      Calculate the original random_quality as target cost,
      Choose one of the primary outputs whose probability is farthest
        away from 0.5;
      For each input {
            Calculate the delta_adjustment;
            Determine the correction;
      }
      If all costs after doing corrections are worse than target cost
            Break;
      else
           Choose the best cost value and do the correction;
}
```
Figure 8: Pseudo code of the iterative_refinement method

The iterative_refinement method first derives the delta_adjustments between inputs and outputs. Given a target output probability to be modified, we greedily select an appropriate input whose probability adjustment can result in a better result. The process iterates until there is no improvement. In our experiments, we limit each probability adjustment to be less than 0.05.

4. Generation of Input Probabilities for Sequential Circuits

We extend our approach to sequential circuits which can be decomposed into combinational circuits and storage elements like Flip Flops (FFs). Note that given a current state, the next state function is a Boolean function of inputs. Our basic idea is to iteratively derive a set of input probabilities for the next exploration for a current state. If a new state is reached, we save it in a state queue, and derive another set of input probabilities for the next state exploration. Otherwise, a state in the state queue is brought back to the design for further state exploration. The pseudo code is shown in Figure 9.

We now describe how to obtain a set of input probabilities given a current state. Depending on the current state, it is possible that some state variables (representing some FFs) are already determined by the current state. We say that such a state variable is *input-noncontrollable* under the current state; otherwise, the state variable is *input-controllable*.

```
Current state ← initial state;
While (simulation cycles < predefined cycle limit) {
    Generate input probabilities at the current state by our method;
    Performing random verification;
    Simulation cycle incremented;
    If (new state is reached) {
       Store reached new states in the state queue;
       Current state ← new state;
       }
    else {
       Current state ← Current state;
       if (lock times > predefined limit) {
          Current state ← retrieve a state from the state queue;
          lock times = 0;
       }
       lock times incremented;
   }
}
```
Figure 9: Pseudo code for handling sequential circuits

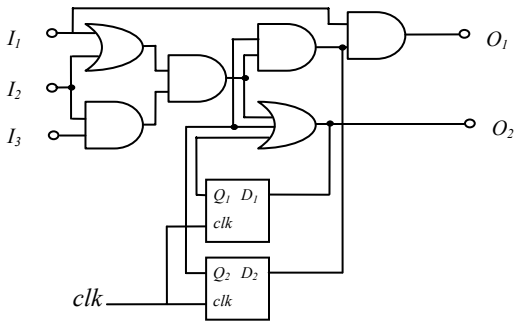


Figure 10: Example with an input-noncontrollable state variable

For example in Figure 10, assuming the current state value (Q_1 = 1, Q_2 = 1). The next state of the state variable D_1 can be immediately evaluated to 1 by the current state. The state variable D_1 is therefore input-noncontrollable under the current state (Q_1 = 1, Q_2 = 1). On the contrary, the next state value of the state variable D_2 is not determined solely by the current state. Therefore, the state variable D_2 is input-controllable under the current state, (Q_1 = 1, Q_2 = 1). Given a current state, since different input assignments cannot affect input-noncontrollable state variables, we should neglect those state variables during state exploration of the current state.

Our objective is then to find a set of input probabilities so that the probabilities of input-controllable state variables can be as close to 0.5 as possible. We then treat the circuit as a combinational circuit and obtain the input probability assignments using the methods described in Section 3. Once the input probabilities of a current state are obtained, we will use these probabilities for next state exploration.

Table 1: Experimental results of combinational circuits

circuit	|PI|	|PO|	literals	uniform		our	
				vectors	outputs	vectors	outputs
apex6	135	99	854	435521	254666	434177	340286
apex7	49	37	274	906785	458857	884129	603929
b9	41	21	140	1946305	8619	1714433	10475
C880	60	26	473	656609	173124	483105	311300
dalu	75	16	1159	389025	35106	385889	37497
i1	25	16	51	4076129	2236	4032033	2418
k2	45	45	1092	408225	283	402721	300
pair	173	137	1964	170113	138608	169441	167615
term1	34	10	258	1302369	649	1299457	666
x1	51	35	357	968289	520482	961153	609529
x3	135	99	890	362817	213357	355754	292073
x4	94	71	412	659809	462432	658337	599116
total				12281996	2268419	11780620	2975201
ratio				1	1	0.95	1.31

Table 2: Experimental results of sequential circuits

circuit	|PI|	|latch|	literals	uniform		our	
				vectors	states	vectors	states
s344	9	15	269	15890495	1489	14072425	2625
s349	9	15	273	15380933	1488	13720774	2625
s382	3	21	306	12865699	432	7822741	8865
s400	3	21	320	12264698	448	7453164	8865
s444	3	21	352	12460643	446	7765588	8865
s526	3	21	445	8035827	423	5958576	8868
s641	35	19	539	6003471	1226	4419548	1544
s713	35	19	591	5181231	1207	2093128	1544
s1196	14	18	1009	2124653	2614	1602308	2614
s1238	14	18	1041	1923065	2613	1483571	2615
total				92120715	12386	66391823	49030
ratio				1	1	0.72	3.95

5. Experimental Results

We conduct experiments over a set of benchmark circuits, a public domain TV80 microprocessor core and an industrial AES design. Table 1 shows the results for combinational benchmarks and Table 2 for sequential circuits. In the experiments, we record the number of reached output combinations for a combinational circuit by simulating 100 seconds and record the number of visited states for a sequential circuit by simulating 10,000 seconds. The first four columns show the name, number of inputs, number of outputs, and number of literals, respectively. Column five and six show the number of input vectors, and number of reached output combinations by uniform random approach (uniform). Column seven and eight show the number of input vectors, and number of reached output combinations by our approach (our). For example, after 100 seconds, 603,929 output combinations of circuit apex7 are reached in our approach while uniform random approach only reaches 458,857 output combinations.

Table 2 summaries the experimental results on sequential benchmarks. The run time of random verifications are set to 10,000 seconds. Take circuit s344 as an example. The results are shown in Figure 11. After running 10,000 seconds our approach can reach 2,625 states while uniform random approach can reach 1,489 states. The experimental results show on average, we obtain 31% more output combinations and 295% more sequential states than uniform random approach. The number of input vectors applied by our approach is less than that of uniform random approach because of the computation overhead to find good probabilities.

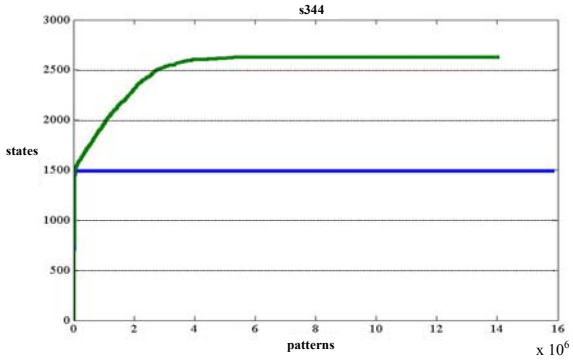


Figure 11: State coverage comparison of s344

We also apply our algorithms to the microprocessor TV80 core [16] from the OPENCORES.ORG. TV80 is an 8-bit microprocessor compatible with 8080/Z80 instruction set. In the experiments, we restrict the CPU time to 100,000 seconds. The results are shown in Figure 12. Our approach generates 952,186 vectors to cover 254,267 states but uniform random approach generates 1,857,071 vectors to cover only 57,084 states. We use half vectors of uniform random to reach five times of state coverage.

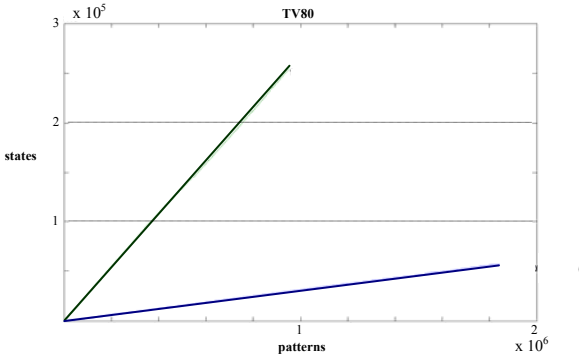


Figure 12: State coverage comparison of TV80

We also perform an experiment on an industrial AES (Advanced Encryption Standard) encryption processor which is a 4-stage pipeline design. Our algorithm chooses the last stage FFs as the target FFs and lets all the FFs in the previous stages become transparent. Then, we can use the same set of equations to derive "good" input probabilities. In this way, we are able to reach many states in a pipeline design. We limit the CPU time to 100,000 seconds. Our approach generates 299,810 vectors to cover 231,315 states and the uniform random approach generates 450,121 vectors to cover only 227,771 states. In fact, if we focus on the target FFs, our approach covers 231,315 states but the uniform random approach covers only 120,073 states.

We would like to mention that these experiments are done on the signal on bit level without considering relations between input signals. The same algorithm can be further extended to word level by restricting the same distribution of all bits of the same word.

6. Conclusions

In this paper, we proposed algorithms to analyze the circuit structure to guide the probability assignment of inputs for random verification with the aim at higher coverage. The experimental results have shown that on average our approach can obtain 31% more output combinations and 295% more states than those from the uniform random approach for combinational circuits and sequential circuits, respectively.

References

[1] Ken Albin, “Nuts and Bolts of Core and SoC Verification”, in *Proc. of Design Automation Conference,* pages 249-252, June 2001.

[2] A. K. Chandra and V. S. Iyengar, “Constraint Solving for Test Case Generation: A Technique for High Level Design Verification,” in *Proc. Int'l Conference on Computer Design*, pages 245-248, 1992.

[3] A. Chandra, V. Iyengar, D. Jameson, R. Jawalekar, I. Nair, B. Rosen, M. Mullen, J. Yoon, R. Armoni, D. Geist, and Y. Wolfsthal, “AVPGEN-A Test Generator for Architecture Verification,” in *IEEE Transactions on VLSI Systems*, vol. 3, issue 2, pages 188-200, June 1995.

[4] M. Kantrowitz, and L.M. Noack, “Functional Verification of a Multiple-issue, Pipelined, Superscalar Alpha Processor – the Alpha 21164 CPU Chip,” in *Digital Technical Journal*, vol. 7, no.1, Fall 1995.

[5] D. Moundanos, J. A. Abraham, and Y. V. Hoskote, "Abstraction Techniques for Validation Coverage Analysis and Test Generation," in *IEEE Trans. on Computers*, vol. 47, no.1, pages 2-14, 1997.

[6] K. Shimizu, and D. L. Dill, “Deriving a Simulation Input Generator and a Coverage Metric from a Formal Specification,” in *Proc. of Design Automation Conference*, June 2002.

[7] S. Tasiran, F. Fallah, D. G. Chinnery, S. J. Weber, and K. Keutzer, “A Functional Validation Technique: Biased-Random Simulation Guided by Observability-Based Coverage,” in *Proc. of Int'l Conference on Computer Design*, pages 82-88, 2001.

[8] S. Tasiran, F. Fallah, D. G. Chinnery, S. J. Weber, and K. Keutzer, “Coverage-Directed Generation of Biased Random Inputs for Functional Validation of Sequential Circuits,” *International Workshop on Logic and Synthesis*, California, June 2001.

[9] H.-J. Wunderlich, “PROTEST: A Tool for Probabilistic Testability Analyses,” in *Proc. of Design Automation Conference*, pages 204-211, June 1985.

[10] H.-J. Wunderlich, “On Computing Optimized Input Probabilities for Random Tests,” in *Proc. of Design Automation Conference*, pages 392-398, June 1987.

[11] L. Yossi, “Verification of the PalmDSPCore Using Pseudo Random Techniques,” http://www. veri-sure.com/ papers.html.

[12] J. Yuan, K. Shultz, C. Pixley, H. Miller, and A. Aziz, “Modeling Design Constraints and Biasing in Simulation using BDDs,” in *Proc. of Int'l Conference on Computer-Aided Design*, pages 584-589, Nov. 1999.

[13] J. Yuan, K. Albin, A. Aziz, and C. Pixley, “Constraint Synthesis for Environment Modeling in Functional Verification,” in *Proc. of Design Automation Conference,* pages 296-299, June 2003.

[14] Synopsys, Inc., “Constrained-Random Test Generation and Functional Coverage with Vera,” http://www.synopsys.com/ products/vera/vera.html.

[15] Verisity Design, Inc., “Specman: Spec-based Approach to Automate Functional Verification,” http://www. verisity.com.

[16] TV80, OPENCORES.ORG, http://www.opencores.org/ projects.cgi/web/tv80/overview.