A Rule-Based Software Testing Method for VHDL Models

Anneliese Andrews, Andrew O'Fallon
School of Electrical Engineering
and Computer Science
Washington State University
Pullman, WA 99164
{aandrews,aofallon}@eecs.wsu.edu

Tom Chen
Department of Electrical
and Computer Engineering
Colorado State University
Fort Collins, CO 80523
chen@engr.colostate.edu

Abstract

The verification of behavioral models is an important step before transferring a hardware design to the layout. A popular approach is to use a variety of code coverage measures to evaluate how much of the design has been simulated. Common coverage measures include branch coverage and bit toggle coverage. This paper presents a test pattern generation approach based on controlflow and dataflow analysis. Heuristic rules use the results of the static analysis to generate new patterns. We applied this technique to a sizable VHDL design and compared resulting coverage to other methods. The results showed that this new method achieved higher branch coverage.

1. Introduction

With rapid advances in VLSI systems, the complexity of behavioral models describing these systems has increased dramatically, resulting in increased effort in behavioral model verification. For a given VLSI system, a variety of verification techniques can be applied to verify the correctness of its behavioral model. The most common techniques treat the behavioral model as a black box and apply a large amount of random and pseudo-random patterns (test cases) to generate model responses that can be manually or semi-manually verified against its specification. Such simulation-based approaches tend to be very time consuming. Recent black-box based verification approaches [5] use simulation coverage metrics to guide test pattern generation [4]. Although other random methods [6, 8] as well as anti-random methods [7, 11] were proposed to improve the effectiveness of random patterns, they are, in general, not efficient when coverage saturates. In addition, there is no guarantee that new test cases will exercise the parts of the model that can potentially exhibit errors in the behavioral model. Therefore, increasingly larger sets of randomly generated test cases are applied to a complex behavioral model over the entire product development cycle to mine any potential errors in the model.

Recent advances in formal verification and assertion-based verification techniques provide a more systematic approach to the problem of behavioral model verification. The use of formal methods for behavioral verification of large designs has been attempted in the past [1]. However, existing formal verification techniques can only handle limited model size and cannot be used for model verification in general. Formal verification techniques are often found useful when used in conjunction with simulation based techniques [9]. The assertion-based verification techniques provide a better monitoring mechanism, but do not address the issue of test pattern generation.

Other methods for generating test cases include the use of constraint programming techniques [10], and fault in jection techniques [3]. This paper aims to provide a more systematic approach to address the issue of test pattern generation. Although behavioral models contain special constructs for modeling aspects of hardware and timing, they, nevertheless, also possess a great deal of software structures, and hence, can be considered as a software program. The method discussed in this paper is based on white box testing, in which the internal structures of the behavioral model are analyzed in terms of their control flow and data flow. New test cases are generated based on heuristic rules. The goal is to supplement the existing black-box based verification approaches by generating new test cases that are more effective in exercising parts of the behavioral model that have not been exercised by the test cases generated using the black-box based approaches. Thus, combining the existing black-box techniques with the proposed white-box technique, the overall number of test cases needed for verifying a given behavioral model can be reduced. This reduces the overall complexity of behavioral model verification.

[2] proposes a rule-based software test data generator to test Ada programs. In the approach, some heuristic rules and symbol information specific to the subject module under test are used by a rule interpreter to generate test cases. The described test data generator is used in conjunction with a software testing environment for a high level programming language. Heuristic rules are specific to the types of statements in the language. To adopt this approach for VHDL requires reformulating the heuristic rules. Section 2 proposes the rule-based approach for VHDL models. Section 3 describes the results of apply-

ing the proposed rules to some VHDL programs. Section 4 draws conclusions and suggests further work.

2. Rule-Based Approach

Fig. 1 shows a diagram of the rule-based test pattern generation system. There are three phases in this model. In phase I, a control flow graph is built for a given VHDL source. It models control flow within an entity and between entities, including concurrency. In phase II, data flow analysis (DFA) is used to generate reaching definition information. The reaching inputs $RI(n)$ are input signals that influence the value of a condition or assignment in statement(s) of node n of the Control Flow Graph (CFG). $RI(n)$ is generated similar to ALGORITHM 3.4. in [12]. In phase III, based on the CFG and reaching definition information, test data generation rules are used by a rule interpreter to generate new test cases. A simulator executes these test cases. The simulator records coverage statistics for the test inputs. The rule interpreter analyzes the execution results and generates additional test cases. This cycle continues until the amount of requested coverage is achieved or until a user specified number of test cases have been generated and executed. The coverage metric in the rule-based approach is branch coverage.

2.1. Test Generation Process

Before a detailed discussion of the test generation process, we define more formally the terms test pattern, test case, and branch point. We also define what it means for a test case to reach a branch point.

Test Pattern. An input pattern $t = (i_1, i_2, \cdots, i_k)$, each i_j $(1 \le j \le k)$ is a input bit $i_j \in \{0, 1\}$.

Test Case. A test case $T = \{t_1, t_2, \cdots, t_n\}$, each t_i $(1 \le i \le n)$ corresponds to a VHDL input pattern. For example, $T = \{t_1, t_2\} = \{(0, 1), (0, 0)\}$ is a test case.

Branch Point. Let N be the number of nodes in the CFG. For $n \in N$, n is a branch point if there is more than one outgoing arc.

T **reaches** b**.** Given a branch point b and a test case $T = \{t_1, t_2, \cdots, t_m\}$, T reaches b if b is not covered by $t_1 \cdots t_{m-1}$, but covered by t_m.

The test generation process has the following steps:

1. Generate an initial test case $T_0 = \{t_{01}, t_{02}, \cdots, t_{0n}\}$ $(n \ge 1)$ by some technique (e.g. random, functional, etc.). Set $i = 0$.

2. Simulate the VHDL model with T_0 and collect branch coverage statistics, including branch coverage for each t_{0j} $(1 \le j \le n)$ and summary branch coverage.

3. While ($i < max$ and $coverage < 100\%$), for a branch point b_k which is not covered yet:

(a) Determine a subsequence $S_m\{t_{i1}, t_{i2}, \cdots, t_{im}\}$ of $T_i = \{t_{i1}, t_{i2}, \cdots, t_{in}\}$ $(m < n)$ such that S_m reaches b_k. Let $S_{m-1} = \{t_{i1}, t_{i2}, \cdots, t_{im-1}\}$

(b) Using test generation rules, generate additional input patterns $S_{m'} = \{t'_{im}, \cdots, t_{im'}\}$, $(m' > m)$

(c) The new test sequence concatenates patterns in S_{m-1} with patterns in $S_{m'}$: $T_{i+1} = S_{m-1} \otimes S_{m'}$

(d) Simulate T_{i+1}

(e) Increment i

We assume that there is an initial test sequence of input patterns (step 1) that has been simulated and for which we have step-by-step coverage information (step 2). As long as the coverage is not complete, and as long as a user-defined number of generation steps has not been exceeded, we do the following: first we determine a branch point we want to cover. Then we determine the subsequence that reaches the branch point (step 3a). We replace the test input pattern of this sequence with a sequence of inputs that according to the heuristic rules, is more likely to execute the uncovered branch. The simulator executes this new test sequence and records coverage information.

2.2. Rules

A test data generation rule consists of two parts: the precondition and the test generation effect. The precondition of a rule describes situations in which the rule could be applied. The test generation effect describes how to generate new test sequences. There are three types of rules: 1) rules with preconditions that apply to any statement (Rule 0). 2) rules for specific types of statements (Rules 1-4). 3) rules that do not have preconditions and apply to all statements (Rules 5-6).

In the following, b is a uncovered branch point which we want to cover. b' refers to a branch point that directly dominates b[12]: $b' >_d b$. t_m is the last test pattern in T that reaches b'.

Rule 0:

Precondition: $|RI(b)|$ is less than a given number max

Test generation effect:

Generate all combinations of values for $RI(b)$. Rule 1 to Rule 6 will not be executed.

Rule 0 is applicable when the length of $RI(b)$ is small. For example, if $|RI(b)|$ is 2, only four new test patterns $(00, 01, 10, 11)$ will be generated that contain all the possible values. We do not have to consider the statement type and possible relationships among $RI(b)$. Thus it makes generation easy and fast. However this rule is not applicable when length is big, for instance if length is 10, 1024 test cases will be produced.

For IF/ELSIF and WHILE statements, new patterns are generated depending on whether or not the condition contains constants.

1. b contains constants:

Rule 1:

Precondition: the statement represented by b is IF, ELSIF or WHILE and b contains constants

Test generation effect:

For EACH constant c contained in b, do:

For EVERY $v \in RI(b)$ that $|v| = |c|$, set the value of v $= c$, $> c$ and $< c$ in t_m respectively in two ways:

(a) set each v one at a time.

(b) set all vs at once.

In this rule, for a given constant contained in b, we first

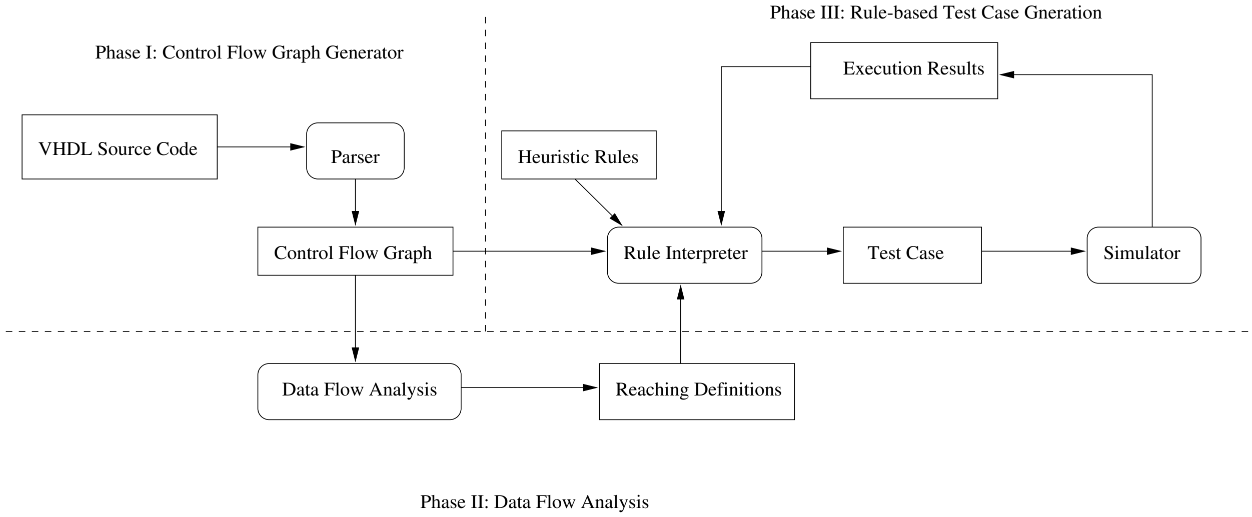

Figure 1. Rule-Based Model

determine which reaching input $v \in RI(b)$ has the same dimensionality as the constant. Then we set these reaching inputs to the value of the constant, a value greater than the constant and a value smaller than the constant to form t'_m. Since there can be more than one such v, this should be done in two ways: (a) setting one v at a time, and (b) setting all vs at once, either all equal to c, all greater than c or all smaller than c. For example, in Fig. 2 (a), both a and d have the same dimensionality as constant c = 0101. Given t_m = 00011011, we first generate test patterns by changing the value of a without changing d. This produces 01011011 (set a equal to c), 01101011 ($a > c$), 01001011 ($a < c$) (Fig. 2 (b)). Then we change the value of d without changing a. This produces 00010101, 00010110 and 00010100. Finally, we change a and d together which produces 01010101, 01100110 and 01000100. For the other constant "1010", the same steps are executed.

2. b may or may not contain constants:

Rule 2:

Precondition: the statement represented by b is IF, ELSIF or WHILE and it may or may not contain constants

Test generation effect:

(a) For each $v \in RI(b)$, set v to the NOT value of v in t_m to form t'_m in two ways:
(i) set each v one at a time.
(ii) set all vs at once.
(b) Let $v_1, v_2, \cdots, v_n$ be the reaching inputs such that $v_i \in RI(b), 1 \leq i \leq n$
(i) generate combinations of reaching inputs of 2, 3 and 4 inputs v_i at a time. The combination has the following format:
- all pairs: $\{v_i, v_j\}, 1 \leq i, j \leq n$ and $i \neq j$.
- all triplets: $\{v_i, v_j, v_q\}, 1 \leq i, j, q \leq n$ and $i \neq j \neq q$.
- all quadruplets: $\{v_i, v_j, v_q, v_l\}, 1 \leq i, j, q, l \leq n$ and $i \neq j \neq q \neq l$.
(ii) set all v in each combination to the value 0 one at a time while other v in the same combination to the value 1.

For the same example shown above, with Rule 2 (a), 11101011 is generated by replacing a with $\overline{a}$ in t_m without changing d (Fig. 2 (c)). 00010100 is generated by setting d without changing a. 11100100 is generated by changing both a and d. With Rule 2 (b), a combination of length 2 $\{a, b\}$ is generated. If $|RI(b)| > 2$, different combinations will be generated which produce more test patterns.

The main idea of Rule 1 and Rule 2 is to set those reaching inputs based on the constants contained in the branch point b. Since we do not know the exact relationships between the reaching inputs and the constants, we set the reaching inputs with values equal to, less than and greater than the constants repectively. Some reaching inputs cannot be set to the constants because their length does not match that of any constant, or because the branch condition does not contain a constant. In this case, we replace them with their logical NOT value in t_m. Further, we do not know the relationships between reaching inputs. To consider relationships, we change values for combinations of two, three, and four input signals. Combinations of more than four result in too many new patterns and test cases.

Rules 3 and 4 apply to CASE statements.

Rule 3:

Precondition: the statement represented by b is WHEN and the constant used in b is c.

Test generation effect:

1. for each $v \in RI(b)$ such that $|v| = |c|$, set $v = c$ to form t'_m in two ways:
(a) set each v one at a time.
(b) set all vs at once.
2. for each reaching input $v \in RI(b)$ that $|v| > |c|$ set every sequential $|c|$ bits of v to the value c one at a time.

The idea of this rule is to set the reaching inputs with the constant used in the WHEN statement. The examples are shown in Fig. 2 (d) and (e). Given t_m = 101011, in step (1), we determine $|sel| = |01|$, so the value of sel is changed to 01 without changing en. Then 011011 is generated. In step (2), $|en| = 4 > |01|$, so we set the 1st and 2nd bit of en to 01 and then set the 2nd and 3rd it to 01 and so on. Thus we generate 100111, 101011 and 101001.

```
Input vector V: a[3:0], b[3:0]

bk:
      if (a>"0101" and d<"1010") then

tm: (00011011)
```

(a) Sample VHDL Code for IF Statement

```
Input vector V: sel[1:0], en[3:0]

bk:
     case sel is
     ......
     when "01" =>......
     ......

tm: (101011)
```

(d) Sample VHDL Code for CASE Statement

```
  a    b
01011011
01101011
01001011
00010101
00010110
00010100
01010101
01100110  <--
01000100
10101011
10111011
10011011
00011010
00011011
00011001
10101010
10011001
10001000
```

(b) Generated Test Case using Rule 1 for Example (a)

```
     a    b
(a) 11101011
    00010100
    11100100  <--
(b) 00000000
```

(c) Generated Test Case using Rule 2 for Example (a)

```
    sel en
(1) 011011  <--
(2) 100111
    101011
    101001
```

(e) Generated Test Case using Rule 3 for Example (d)

```
V: a1[2:0], a2[1:0], a3[3:0]
RI: a1, a2

111111111
000000000
000111111
111001111
111000000
000110000
```

(f) Example using Rule 7

Figure 2. Examples Using Rules

.

Rule 4:
Precondition: the statement represented by b WHEN OTHERS.
Test generation effect:
1. determine all constants c that do NOT occur in any $b' \in DOM(b)$ (dominators of b [12]).
2. for every such c , perform Rule 4.

The idea of Rule 4 is to set the reaching inputs to constants that do not appear in WHEN statements. Those WHEN statements must be the dominators of b.

Rule 5:
Precondition: None.
Test generation effect:
1. generate two special test patterns: t_1 with all '1's, t_2 with all '0's.
2. for each $v \in RI(b)$,set $v = 0$ in t_1 one signal at a time.
3. for each $v \in RI(b)$,set $v = 1$ in t_2 one signal at a time.

This rule generates new test cases based on two special test cases: all '1's and all '0's. Then we set every reaching input to '0' in the all '1' test case and to '1' in the all '0' test case. An example is shown in Fig. 2 (f). (a_1, a_2, a_3) is the input vector, but for a given branch point b, assume $RI(b) = \{a_1, a_2\}$. First the rule requires the two special test patterns $t_1 = 111111111$ and $t_2 = 000000000$. Then we set a_1, a_2 to '0' one at a time in t_1 and 000111111 and 111001111 are generated. Similarly, we set a_1, a_2 to '1' one at a time in t_2. This generates 111000000 and 000110000.

Rule 6:
Test generation effect:
For each newly generated test pattern t, generate another new test pattern $t' = \overline{t}$.

This rule can help to satisfy the signal toggling criterion that every input bit can make a transition from '0' to '1' and then from '1' to '0'.

3. Case Study

We used one large VHDL model to examine the effectiveness and efficiency of the proposed heuristic rules. This model represents an image processing chip. It is referred to as the *sys7* model. It contains 3785 lines of VHDL code with 591 branches. *sys7* is highly sequential in nature, making it more difficult to achieve high coverage. Table 1 summarizes the characteristics of *sys7*.

Table 1. Characteristics of the sys7 Model

sys7 model	
Lines Of Code	3785
Branches	591
Control Bits	7
Data Bits	62
Process Blocks	62
Hierarchy	5

Because applying the rules requires prior execution history, we assume first a small functional test and a series of random input patterns. This should cover the easily reachable branches of the model for which there is no need to employ a sophisticated coverage tool. The first question we wanted to answer was whether applying only DFA-based rules (DFA) after this initial set of tests is better than using only random patterns (Random). Table 2 shows the coverage using DFA after the initial tests. Table 3 shows the coverage using Random with the same number of test patterns used in Table 2. In the tables, #P is no. of patterns, #CC is no. of clock cycles, Brcov is branch coverage, totP is total no. of patterns, and totCC is total no. of clock cycles. We observe that using only DFA provides lower coverage than continuing with random patterns. Observe, however, that the first application of DFA based patterns shows higher coverage than the corresponding random patterns. The DFA based patterns plateau faster.

This gives rise to the second question: Should one have interspersed Random and DFA-based patterns to avoid a coverage plateau?

Table 2. Test Coverage Using DFA

Phase	#P	#CC	Brcov	totP	totCC
Functional	283	283	88.66%		
Rand cc=7	30	210	90.69%	30	210
DFA cc=7	824	5768	92.89%	854	5978
DFA cc=4	766	3064	93.06%	1620	9042
DFA cc=2	6282	12564	93.40%	7902	21606
DFA cc=1	5204	5204	93.40 %	13106	26810

Table 3. Test Coverage Using Random

Phase	#P	#CC	Brcov	totP	totCC
Functional	283	283	88.66%		
Rand cc=7	854	5978	91.37 %	854	5978
Rand cc=4	766	3064	92.55 %	1620	9042
Rand cc=2	6282	12564	94.25%	7902	21606
Rand cc=1	5204	5204	94.25%	13106	26810

Table 4 shows the coverage using both DFA and Random methods. Table 5 shows the coverage using only Random inputs with the same number of test patterns per clock cycle as in Table 4. Comparing these two tables, we observe that we get higher coverage when we use DFA early in combination with Random patterns. This coverage is higher from the first application of the DFA-based patterns and maintains this higher coverage. However, the last round of DFA-based tests does not increase coverage further, similar to the results of the first strategy. Thus, it is important to determine when to switch generation strategies.

To investigate the utility of applying DFA only once, when coverage is already very high, we applied more than twice as many random patterns, followed by a single round of single clock cycle DFA tests. Table 6 shows the coverage results. There is no increase in the coverage since the DFA rule-based method is used too late. While coverage is high, neither the random nor the DFA

Table 4. Test Coverage using Combination

Phase	#P	#CC	Brcov	totP	totCC
Functional	283	283	88.66%		
Rand cc=7	30	210	90.69%	30	210
DFA cc=7	824	5678	92.89%	854	5978
DFA cc=4	766	3064	93.06%	1620	9042
Rand cc=4	800	3200	94.25%	2420	12242
Rand cc=2	4000	8000	95.94%	6420	20242
Rand cc=1	4200	4200	96.79%	10620	24442
DFA cc=1	2048	2048	96.79%	12668	26400

Table 5. Test Coverage Using Random

Phase	#P	#CC	Brcov	totP	totCC
Functional	283	283	88.66%		
Rand cc=7	854	5978	91.37%	854	5978
Rand cc=4	1566	6264	92.55%	2420	12242
Rand cc=2	4000	8000	94.25%	6420	20242
Rand cc=1	4200	4200	95.09%	10620	24442

approach was able to achieve 100% coverage. This is because this design has a relatively long pipeline and thus contains portions of the design that is simply not reachable with the test set-up (the maximum clock cycle used here is 7). We thus recommend that knowledge about the "depth" of the design be used to determine some of the parameters used with test generation and execution. In order to conduct a fair comparison with random patterns, we did not use such DFA based knowledge to form a rule related to clock cycle length.

Table 6. Test Coverage Using DFA Last

Phase	#P	#CC	Brcov	totP	totCC
Functional	283	283	88.66%		
Rand cc=7	5000	35000	93.57%	5000	35000
Rand cc=4	1000	4000	93.91%	6000	39000
Rand cc=2	5000	10000	95.26%	11000	49000
Rand cc=1	5000	5000	96.11%	16000	54000
DFA cc=1	182	182	96.11%	16182	54182

Another important issue was the efficiency of the DFA method. We interpret this as reaching the same coverage with fewer test patterns. To this end we experimented with half a dozen smaller designs. The DFA-based approach was able to reach the same coverage as the random patterns, needing only a fraction of the number of patterns of the randomly generated patterns. This fraction ranged from 0.4% to 44% of the randomly generated patterns.

4. Conclusion

This paper proposes a new test generation method based on control flow and data flow analysis and VHDL specific heuristic rules for test pattern generation. The results show that a judicious combination of random and rule-based test generation will yield higher coverage. This holds even in situations when increasing code coverage has become difficult, because all easily reached code has been covered already. One must take care, though, to recognize when a method has reached a plateau and when one should switch to another method. We plan to investigate further other types of knowledge that could be used to determine more sophisticated rules. For example, the rules currently do not say anything about the relationships between entities and concurrent structure, nor do they use knowledge of the ”depth” of the structure of the design. In addition, this paper is only interested in coverage. In the future we plan to investigate test generation heuristics that can explore timing characteristics. Morever, further tests should be applied using a variety of test methods other than the random pattern generation and the rule-based method.

References

[1] D.P.Appenzeller, A. Kuehlmann, “Formal Verification of a PowerPC(Microprocessor”, *Tech. Report RC (19971)*, IBM T.J. Watson Research Center, Yorktown Heights, N.Y., March 1995.

[2] W.H.Deason, D.B.Brown, K.Chang, J.H.Cross II, “A Rule-Based Software Test Data Generator”, *IEEE Transactions on Knowledge and Data Engineering*, vol.3, no.1, March 1991.

[3] T. A. DeLong, B. W. Johnson, J. A. Profeta III, ”A Fault Injection Technique for VHDL Behavioral-level Models”, *IEEE Design and Test of Computers*, vol. 13, no. 4, Winter 1996, pp. 24–33.

[4] A. Fine and A. Ziv, ”Coverage Directed Test Generation for Functional Verification using Bayesian Networks”, *Proc. 40th Design Automation Conf.*, Anaheim, CA, June 2003.

[5] A. Gluska, ”Coverage-Oriented Verification of Banias”, *Proc. 40th Design Automation Conf.*, Anaheim, CA, June 2003.

[6] R.Kapur, S.Patil, T.J.Snethen, T.W.Williams. ”A Weighted Random Pattern Test Generation System”, IEEE Transaction On Computer-Aided Design Of Integrated Circuits And Systems, Vol.15, NO.8, Augustr 1996.

[7] Y.K.Malaiya. ”Antirandom Testing: Getting the Most out of Black-Box Testing”, Procs. ISSRE ’95, Toulouse, Oct. 1995.

[8] L.Nachman, K.Saluja, A.Upadhyaya, R.Reuse. ”A Novel Approach to Random Pattern Testing of Sequential Circuits”, IEEE Transactions on Computers, vol.47, NO.1, January 1998.

[9] S. Tasiran, et. al. ”Using a Formal Specification and a Model Checker to Monitor and Direct Simulation Verifying the Multiprocessing Hardware of the Alpha 21364 Microprocessor”, *Proc. 40th Design Automation Conf.*, Anaheim, CA, June 2003.

[10] R.Vemuri, R.Kalyanaraman, ”Generation of Design Verification Tests from Behavioral VHDL Programs Using Path Enumeration and Constraint Programming”, *IEEE Trans. VLSI Systems*, Vol. 3, No. 2, June 1995, pp. 201-214

[11] A.von Mayrhauser, A. Bai, “Fast Antirandom (FAR) Test Generation”, *3rd IEEE high Assurance Systems Eng. Symp.*, IEEE Computer Society PRess, Los Alamitos, Calif., 1998.

[12] Hans Zima, Barbara Chapman, ”Supercompilers fpr Parallel and Vector Computers”, ACM Press.