Behavioral Fault Simulation and ATPG System for VHDL

Tim H. Noh
ELDC/Custom Microelectronics
Defense Electronics Supply Center
Dayton, OH 45444-5765

Chien-In Henry Chen†
Dept. of Electrical Eng'g
Wright State University
Dayton, OH 45435

Soon M. Chung
Dept. of Computer Science & Eng'g
Wright State University
Dayton, OH 45435

Abstract --- Due to the increasing level of integration achieved by Very Large Scale Integrated (VLSI) technology, traditional gate-level fault simulation becomes more complex, difficult, and costly. Behavioral fault simulation at top functional level, described in a hardware description language, offers very attractive alternatives to these problems. This paper presents a new way to simulate the behavioral fault models for the Very high speed integrated circuits Hardware Description Language (VHDL). The performance analysis shows that relatively small number of test patterns generated by the behavioral fault simulation and Automatic Test Pattern Generation (ATPG) system detects around 98 percent of all testable gate-level faults.

1. Introduction

The complexity of current VLSI and ULSI circuits has promoted the increasing use of Hardware Description Languages (HDLs), such as VHDL and Verilog, in circuit design. The use of HDLs coupled with synthesis tools provides an efficient design methodology for carrying out complex microcircuit design.

While HDLs are gaining momentum in design area, fault modeling and test generation at higher levels of abstraction have not been well developed yet. The traditional gate-level fault simulator and ATPG system display inefficiency and shortfall when the large number of gates in VLSI challenges them. On the other hand, the behavioral fault simulation for circuits at the high level functional description in a HDL is very simple, efficient, and affordable. Ten different behavioral fault models were selected and used to generate test patterns. Actual behavioral fault simulation system was implemented and the results were compared with the gate-level approach. The comparison of the results shows that even though there exists a little gap between the behavioral fault models and the physical faults, the behavioral fault simulation offers a very attractive alternative to the gate-level fault simulation, especially for complex circuits.

2. Overview of the Evaluation System

The overview of the evaluation system is shown in Figure 1. The VHDL description is an input to the both behavioral ATPG/fault simulation and the synthesis tools. The VHDL code was synthesized to an equivalent gate-level representation so that the gate-level simulation can be carried out with the test patterns generated by the behavioral ATPG/fault simulation system. The results of the behavioral and the gate-level fault simulation are compared and analyzed.

† C.-I. H. Chen was supported in part by the U. S. Air Force under contract #F33615-93-C-1226.

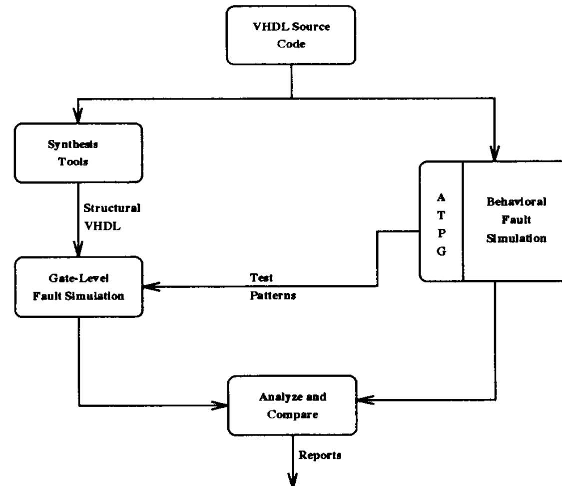

Figure 1. Overview of the evaluation system.

ATPG/Behavioral Fault Simulation System

The ATPG/behavioral fault simulation system is based on the behavioral fault simulation proposed in [1] which laid out good ground rules and guidelines. Some modifications, however, were made to accommodate ATPG and the fault grading system. The same notations introduced in [1] are used here also. In short, curly braces ({ }) indicate a single file. For example, $\{L_1,L_2\}$ means a single file composed of two items. Square brackets ([ ]) indicate a collection of related files. Thus, $[F_1,F_2,F_3]$ denotes three related but separate files, F_1, F_2, and F_3.

Figure 2 shows the ATPG system utilizing the fault simulation, and the following describes each of the blocks.

- The VHDL source file, $\{F_0\}$, is any VHDL model that is fault free from simulation.
- The pre-processor translates the VHDL code into a format that the fault extractor and the fault mapper can use. The output file of the preprocessor is denoted by $\{F'_0\}$.
- The fault extractor module generates fault lists from the preprocessed source code based on ten behavioral fault models to be discussed in the next section. The output of this module is a list of N faults. Notation for the list of N faults is $\{f_1, f_2, f_3,..., f_n\}$.
- The fault mapper takes $\{F'_0\}$ from the preprocessor and the fault list file $\{f_1, f_2, f_3, ..., f_n\}$ from the fault extractor, and generates N faulty files, $[F_1, F_2, F_3, ..., F_n]$. These faulty files are used during behavioral fault simulation one at a time. The fault mapper also produces the control program (Unix shell script) and necessary data files to guide the ATPG process.

• The control program controls the flow of the ATPG process. First, it selects a target fault. Second, it invokes a test generation routine for the target fault. Third, the control program will execute a test bench generator routine then invoke the VHDL simulator for behavioral fault simulation. Fourth, discard the detected fault and update the fault list, then go back to the first step. Continue until there is no more fault or only undetectable faults are left.

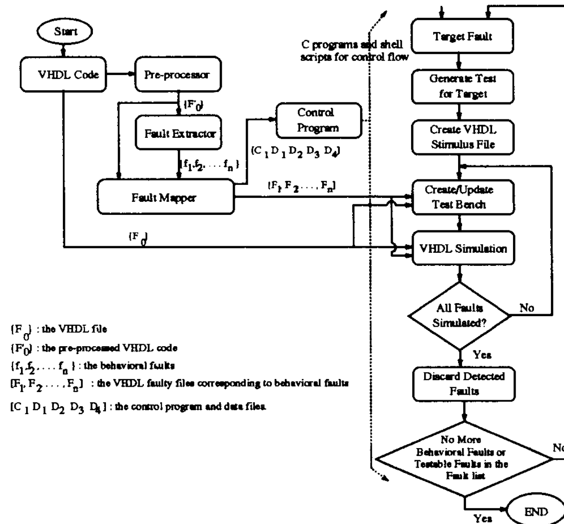

Figure 2. Behavioral ATPG/fault simulation system.

<u>The Gate-level Fault Grading and Simulation System</u>

In order to perform fault simulation at the gate-level for the circuit described in VHDL, the Synopsys Design Compiler [2] was used to synthesize the behavioral VHDL code in structural VHDL. The gate-level fault simulator uses the structural VHDL code and the test patterns generated by ATPG/Behavioral Fault Simulation block to grade faults and to generate the fault simulation report.

3. Behavioral Fault Models and Fault Mapper

Due to the potential advantages of the behavioral fault simulation, many different fault modeling techniques were introduced recently [1,3,4]. Our approach was to use the fault models whose effects were known or studied, and they were improved to handle various data types. The fault models selected for our study can be classified into ten categories: Input stuck-at fault, Output stuck-at fault, If stuck then fault, If stuck else fault, Elsif stuck then fault, Elsif stuck else fault, Assignment statement fault, Dead clause fault, Micro-operation fault, Local stuck data fault.

The behavioral fault mapper generates faulty files by using the fault list and the preprocessed VHDL code. Each faulty file contains only one behavioral level fault inserted. Now, let's discuss the fault mapping techniques for all ten fault categories, and their representation of failures.

Input stuck-at fault. The input stuck-at fault represents the failure of the primary input signal. The input signal can be stuck at 0 or 1 for the bit and bit_vector types, and false or true for the boolean type. The std_logic and std_logic_vector are treated the same as the bit and bit_vector. This fault is mapped by replacing every occurrence of the input signal in the architecture body with the corresponding stuck-at fault.

```
Example: Primary input = A;
 OUT1 <= A;    OUT2 <= A;

When A is bit/std_logic type:
  s-a-1:  OUT1 <= '1'; OUT2 <= '1';
  s-a-0:  OUT1 <= '0'; OUT2 <= '0';
When A is bit_vector/std_logic_vector with length of four:
  s-a-1:  OUT1 <= "1111"; OUT2 <= "1111";
  s-a-0:  OUT1 <= "0000"; OUT2 <= "0000";
When A is boolean type:
  s-a-false:  OUT1 <= false;  OUT2 <= false;
  s-a-true :  OUT1 <= true;  OUT2 <= true;
```

Output stuck-at fault. The *output stuck-at* fault represents the failure of the primary output signal. The output signal can be stuck at 0 or 1 for the bit/std_logic and bit_vector/ std_logic_vector types, and false or true for the boolean type. This fault is mapped by replacing the right hand side of all occurrences of the output signal assignment in the architecture body with the corresponding stuck-at fault.

```
Example:  Primary output = A;
  A <= COM1 XOR COM2 XOR COM3;

When A is bit/std_logic:    s-a-1: A <= '1';
When A is bit_vector/std_logic_vector with length four :
  s-a-0: A <= "0000";
```

If stuck then fault. The *if stuck then* fault represents the failure to execute the else (and elsif, if exist) portion of statement(s) for the if construct. This fault is mapped by replacing logical_expression or condition between "if" and "then" with boolean value TRUE. Let's consider the following example to illustrate this fault.

```
Example:
1#   IF (logical_expression1) THEN
2#      A <= COM1;
3#   ELSIF (logical_expression2) THEN
4#      A <= COM2;
5#   ELSE
6#      A <= COM3;
7#   END IF;
```

In the presence of an *if stuck then* fault, the logical_expression1 in line 1 will be replaced with TRUE, i.e., 1#   IF (TRUE) THEN.

If stuck else fault. The *if stuck else* fault represents the failure to execute the if then portion of statement(s) for the if construct. This fault is mapped by replacing logical_expression or condition between "if" and "then" with boolean value FALSE; i.e., 1# IF (FALSE) THEN.

Elsif stuck then fault. The *elsif stuck then* fault represents the failure to execute the following else (and elsif, if exist) portion of statement(s) for the if construct. This fault is mapped by replacing logical_expression or condition between "elsif" and "then" with the boolean value TRUE. In the presence of an *elsif*

stuck then fault the logical_expression2 in line 3 of the above example will be replaced with TRUE so that A can be assigned to the signal value of COM1 or COM2 depending on the result of the logical_expression1, but not COM3.

Elsif stuck else fault. The *elsif stuck else* fault represents the failure to execute the elsif then portion of statement(s) for the if construct. This fault is mapped by replacing logical_expression or condition between "elsif" and "then" with the boolean value FALSE; for the same example, the line 3 becomes "ELSIF (FALSE) THEN". This mapping guarantees that A is assigned either COM1 or COM3 but not COM2.

Assignment statement fault. The *assignment statement* fault represents the failure to assign a new value to a signal. In the presence of an assignment statement fault the signal to the left side of assignment operator (<=) will be assigned to one of the logic values the signal can have. The fault is mapped by replacing the expression to the right of the assignment operator with corresponding logic value, for example '0' and '1' for bit type. This fault model is extended to not only bit/std_logic, bit_vector/std_logic_vector, and boolean but also to the enumerated data type. Let's consider an example to illustrate the effect of the assignment statement fault on the enumerated type.

Example:

```
1# type traffic_light is (GREEN, YELLOW, RED);
2# SIGNAL LIGHT: traffic_light;
3# IF (condition_1) THEN
4#    LIGHT <= GREEN;
5# ELSIF (condition_2) THEN
6#    LIGHT <= YELLOW;
7# ELSE
8#    LIGHT <= RED;
9# END IF;
```

In this example, line 4 is mapped to three different ways since the type traffic_light has three values. The signal LIGHT can be assigned to GREEN, YELLOW, and RED. However, in line 4, the signal value GREEN is the original value, therefore, it should be mapped to either YELLOW or RED. The following shows the mapping results.

The LIGHT is assigned to YELLOW (stuck at YELLOW):

```
1# type traffic_light is (GREEN, YELLOW, RED);
2# SIGNAL LIGHT: traffic_light;
3# IF (condition_1) THEN
4#    LIGHT <= YELLOW;   -- Mapped to YELLOW
5# ELSIF (condition_2) THEN
6#    LIGHT <= YELLOW;
7# ELSE
8#    LIGHT <= RED;
9# END IF;
```

Dead clause fault. The *dead clause* fault represents the failure of a WHEN clause in a CASE statement to execute when selected. In the following example, there could be five dead clause faults since five alternatives exist in the CASE construct. Let's consider one of the five faults. Assume the fault is presented when OP_CODE is equal to "10". This implies that the multiplication operation will never be performed and the assignment of PC_WRITE will fail.

Example:

```
1# CASE OP_CODE IS
2#   WHEN "00" =>
3#     INSTRUCTION <= ADD;
4#   WHEN "01" =>
5#     INSTRUCTION <= SUB;
6#     REG_SEL <= '0';
7#   WHEN "10" =>
8#     INSTRUCTION <= MUL;
9#     PC_WRITE <= '1';
10#  WHEN "11" =>
11#    INSTRUCTION <= DIV;
12#  WHEN OTHERS =>
13#    INSTRUCTION <= NO_OP;
14# END CASE;
```

The mapping of the fault when OP_CODE is "10" is done by replacing the expression to the right of the assignment operator with the signal name to the left of the operator in the WHEN clause. However, if the signal to the left of the operator is of type "OUT" then the fault is mapped by commenting the assignment statement in the WHEN clause.

```
....
7#   WHEN "10" =>
8# -- INSTRUCTION <= MUL; ** instruction: "OUT"
9#    PC_WRITE <= PC_WRITE;   ** pc_write: signal
....
```

Micro-operation fault. The *micro-operation* fault represents a failure of a micro-operation to perform its intended function. The operator can be classified into four categories; logical operators, relational operators, unary operators, and arithmetic operators. An operator may fail to any other operator in its category. This fault is mapped by replacing the operator considered with its counter operator which must be defined. For example, logical duality can be utilized to select the counter operator. Here are some examples.

Example:

```
COUT <=(A AND B) OR (B AND CIN);
-- FAULTY: 1-st AND failed to OR
COUT <=(A OR B) OR (B AND CIN);
```

Local stuck data fault. The *local stuck data* fault represents a failure for a signal object to have a proper value within a local expression. More than one expression within a device model may use the signal. The fault is mapped such that the signal in only one expression will be replaced with one of logic values that the signal can have. The signal in other expressions will retain proper logic value. The *local stuck data* fault is extended to handle bit/std_logic, bit_vector/ std_logic_vector, boolean, and enumerated data types.

Example:

```
1# SIGNAL C0: STD_LOGIC;
2# SUM1 <= A1 XOR B1 XOR C0;
-- the local stuck to '1'
2# SUM1 <= A1 XOR B1 XOR '1';
```

4. Behavioral ATPG/Fault Simulation

The ATPG system utilizing the behavioral fault simulation is outlined in Figure 2. Since the fault extractor and the fault mapper were described in previous sections, mainly the control flow of ATPG system is described in this section.

Control Program in ATPG System

The purpose of the control program is to guide the behavioral fault simulation with the ATPG system. The control program is generated along with an additional shell script, a data file, and three frame files by the fault mapper. The shell script is used to compile/analyze the fault-free VHDL model and all faulty files. The data file contains the names of all faulty files, one on each line. This data file is used by the control program to update the test bench and its configuration file. After update of the test bench and its configuration, the name of the faulty file is removed from the data file. When the data file becomes empty the fault simulation exits. Three frame files are used to create VHDL stimulus file and to update test bench and its configuration file.

Behavioral Test Pattern Generator. The control program invokes a test generator to devise the test pattern(s) for the target fault. The linear feedback shift register (LFSR) algorithm was utilized to generate the test patterns in this research. However, any test generation algorithm can be used for this purpose.

Creating VHDL Stimulus. The control program invokes the VHDL stimulus generator. The VHDL stimulus file is generated by using the test patterns generated from previous step, stimulus frame file generated by the fault mapper, and fault list file. The stimulus file is compiled.

Creating/Updating Test Bench. Once the VHDL stimulus file is generated then the control program invokes the test bench generator routine. The purpose of the test bench is to determine whether or not the set of test patterns (T) generated for the target fault could be a test for the other fault. The stimulus file (component in VHDL) generated from the test patterns drives the fault-free model and the faulty model, and the responses of these two models are compared. If they are different then the same set of test patterns are also tests for the fault modeled in the faulty file. This procedure is repeated until all faulty files are completely simulated. The simulation of respective good (fault-free) model/faulty model pairs are controlled through the configuration file of the test bench. The following shows the frame of test bench in VHDL. Italicized terms denote the codes varying simulation to simulation.

Frame for test bench:

```
use std.textio.all;
ENTITY_name_TB IS
END entity_name_TB;

ARCHITECTURE STRUCTURE OF entity_name_TB IS

-- component declaration for good and faulty model
component MUT
PORT(input_ports :IN types; output_ports :OUT types);
end component;

-- component declaration for stimulus
component MUT_ST
PORT(input_ports:OUT type);
end component;

-- signals used to drive good model and faulty model inputs
signal A1_IN,A2_IN .......;
-- Output signals from good model
signal S1_OUT,S2_OUT.......;
-- Output signals from faulty model
signal S1_FOUT,S2_FOUT,S3_FOUT,S4_FOUT,C4_FOUT:BIT;

BEGIN
-- component instantiation of good model
GOOD: MUT port map(A1_IN,A2_IN, .....,S1_OUT,S2_OUT,.......);
-- component instantiation of faulty model
FAULTY:MUT port map(A1_IN,A2_IN, ...,S1_FOUT,S2_FOUT,...);
-- component instantiation of stimulus model
STIMULUS: MUT_ST port map(A1_IN,A2_IN, .........);
-- Compare responses of good model and faulty model
COMPARATOR: PROCESS
FILE tp_file: TEXT is OUT "/home/TEMP.TP";
variable outline: LINE;
variable st_tmp:string(1 to 20);
BEGIN
  Wait on A1_IN,...,S1_OUT,S2_OUT,...,S1_FOUT,S2_FOUT,...;
  st_tmp := ":faulty file name";
  if (S1_OUT /= S1_FOUT) then
    write(outline,A1_IN); write(outline,A2_IN);
     ...
    write(outline,S1_OUT); write(outline,'x');
     ...
    write(outline,st_tmp); writeline(tp_file,outline);
    ASSERT FALSE report "TEST GENERATED" severity failure;
  elsif (S2_OUT /= S2_FOUT) then
   .....
  elsif (.......) then
   ....
  else   -- This else clause is needed for sequential circuit only,
   .....  -- (Capture input sequences since multiple input patterns
  end if; --    are required to detect the faults in the sequential circuit.)
END COMPARATOR PROCESS;
END STRUCTURE;
```

Discard Detected Faults. The control program will discard the detected faults by removing the name of the faulty files from the data file (*entity_name*nn). If the data file is not empty and the remaining faults are not redundant faults then go back to the behavioral test pattern generator to generate test patterns for a new target fault.

5. Gate-Level Fault Grading and Simulation

In general, it is difficult to correlate behavioral fault models and physical circuit failures due to the high level abstraction of behavioral faults. However, the effectiveness of the behavioral ATPG system in detecting physical defects must be evaluated and analyzed. One way to measure the quality of the behavioral ATPG system is to perform a fault simulation on the equivalent gate-level representation with the test patterns generated by the behavioral ATPG system. When starting with a behavioral description, it is a two-step process: the synthesis process and the gate-level fault simulation.

The synthesis tool translates a RTL design description into a gate-level representation, and optimizes it with respect to a set of design goals/constraints for a given target technology library. The design compiler of Synopsys Inc.[2] was chosen as the synthesis tool. All example circuits were synthesized with respect to two separate design goals: one for the fastest circuit in speed, and the other for the smallest circuit in area. The technology library used for the synthesis was a strip down version of LSI Logic's 10K library. The synthesis results were saved in the form of a

structural VHDL description.

Once the structural VHDL description is available from the synthesis process, it is converted to ISCAS benchmark formats [5, 6]. This internal format and the test patterns from the behavioral ATPG system are used for one of the gate-level fault simulation. The gate-level single stuck-at fault model was used for the fault simulation.

Since it is not guaranteed that the synthesized circuits are fully testable, the testability of the circuits must be estimated. In order to measure the testability, a large number of random test patterns were generated and the gate-level fault simulation was carried out.

6. Results and Analysis

The behavioral ATPG/fault simulation system was implemented by using Unix C-shell programming language and C language on SUN Sparc10 workstations. A couple of commercially available tools, Vantage Spreadsheet [7] and Synopsys VHDL System Simulator [8], were utilized to support the behavioral ATPG/fault simulation system. Nine circuits were used to evaluate the performance of the system. The results of the fault simulation are shown in Table I. For the gate-level fault simulation, the fault coverage with the test patterns generated by the behavioral ATPG/fault simulation system is provided under "BT" column and the fault coverage with the random test patterns is provided under "GT" column.

The current implementation of the behavioral ATPG/fault simulation system performed exceptionally well in generating test patterns for the behavioral faults. System performance can be greatly enhanced if a more efficient behavioral test generation algorithm, instead of LFSR algorithm, is used for the target faults.

The results of the fault simulation shows that relatively small number of the test patterns generated by the behavioral ATPG/fault simulation system detected around 98 percent of all testable gate-level faults. In particular, if synthesized circuits are testable, mainly combinational circuits, then the overall gate-level fault coverage achieved by using the test patterns of the behavioral ATPG/fault simulation system was around 98 percent. However, the synthesized sequential circuits displayed poor testability. This poor testability is an inherent characteristic of the sequential circuits and is partially caused by the logic optimization technique of the synthesis tool. Often, the logic optimization goal and testability goal conflict each other. Even though the overall gate-level fault coverage is low for the sequential circuits, it is important to notice that the test patterns of the behavioral ATPG/fault simulation system detected nearly all testable faults.

Table I: Fault Simulation Results.

Test Circuits		Behavioral Fault Simulation	Gate-level Fault Simulation			
			Fastest		Smallest	
			BT	GT	BT	GT
4-bit full Adder	Coverage (%)	100	95.41	95.41	94.63	94.64
	no. of patterns	50	50	1024	50	1024
Mux_2_1 16-bit input	Coverage (%)	100	100	100	100	100
	no. of patterns	24	24	262144	24	262144
Full adder Subtractor 8-bit	Coverage (%)	100	96.32	98.68	94.45	95.41
	no. of patterns	150	150	1024	150	1024
Parallel Multiplier 4-bit	Coverage (%)	96.67	96.68	97.34	96.45	97.52
	no. of patterns	125	125	1024	125	1024
Moore (FSM)	Coverage (%)	100	100	100	100	100
	no. of patterns	10	10	21	10	20
16 bit Register	Coverage (%)	100	100	100	100	100
	no. of patterns	6	6	10	6	10
Decade Counter/ Divider	Coverage (%)	99.10	84.82	84.82	34.83	34.83
	no. of patterns	51	51	262144	51	262144
Vending Machine Controller	Coverage (%)	98.20	52.50	61.25	17.78	18.67
	no. of patterns	55	55	262144	55	262144
Controller for RISC R-2000	Coverage (%)	81.10	39.50	44.67	40.91	40.91
	no. of patterns	101	101	262144	101	262144

7. Conclusions

A complete behavioral fault simulation and ATPG system for circuits modeled in VHDL has been presented in this paper. Ten different behavioral fault models were selected and used to generate test patterns through fault simulation. The results are very encouraging but not perfect: the behavioral fault simulation detected about 98 percent of the testable gate-level faults. Nevertheless, gate-level fault simulation is not an effective solution for complex microcircuits, and the results of this research show that behavioral fault simulation will remain as a highly attractive alternative for the future generation of VLSI/ULSI circuits.

References

[1] P. C. Ward and J. R. Armstrong,"Behavioral Fault Simulation in VHDL", Proc. of 27th ACM/IEEE Design Automation Conference, pp. 587-593, 1990.

[2] Synopsys Design Compiler Reference manual, Synopsys, Inc., 1994.

[3] U.H. Levendel and P.R. Menon, "Test Generation Algorithms for Computer Hardware Description Languages", IEEE Transactions on Computers, Vol. C-31 (7), pp.577-588, 1982.

[4] C.-I. H. Chen and S. Perumal, "Analysis of the Gap between Behavioral and Gate-Level Fault Simulation", Proc. of 6th Annual IEEE International ASIC Conference and Exhibit, September, pp.144-147, 1993.

[5] F. Brglez, P. Pownall, and R. Hum, "Accelerated ATPG and Fault Grading via Testability Analysis", Proc. of IEEE International Symposium on Circuits and Systems, pp. 695-698, 1985.

[6] F. Brglez, D. Bryan, and K. Kozminski, "Combinational Profiles of Sequential Benchmark Circuits", Proc. of IEEE International Symposium on Circuits and Systems, pp. 1929-1934, 1989.

[7] Vantage SpreadSheet User Guide, Vantage Analysis Systems, Inc., 1993.

[8] Synopsys VHDL System Simulator reference manual, Synopsys, Inc., 1994.