Using VHDL Critical Path Tracing Models for Pseudo Random Test Generation

Massoud Shadfar, Armita Peymandoust, Zainalabedin Navabi
Electrical and Computer Engineering Department
Faculty of Engineering, Campus No. 2
University of Tehran
14399, Tehran IRAN
Tel: +98-21-800-8485; Fax: +98-21-646-1024
Email: navabi@ece.ut.ac.ir

Abstract

VHDL gate level models for fault simulation by the Critical Path Tracing method have been developed. These models are used by a VHDL test bench for pseudo random test generation. Modeling strategy and the test environment will be described in this paper.

1. Introduction

In this paper we will use VHDL gate models in an integrated environment for pseudo random test generation. VHDL models for fault collapsing will generate a reduced fault list. A VHDL test bench will read faults from the reduced fault list and apply random tests patterns to VHDL models for fault simulation. When a random test is found to detect faults from the fault list, the test pattern and the faults detected by this pattern are recorded. At the end of the test process, test patterns and faults that remain undetected along with achieved fault coverage will be listed in a file.

Fault simulation is done by the Critical Path Tracing method. In this method a test is applied and circuit lines with values that influence a primary output value are identified. Stuck-at-value faults on such lines are detected by the input test pattern.

The following section consists of a description of a test environment for test application and fault detection. Section 3 of this paper presents fault list generation method and its corresponding file format. Section 4 details the fault simulation method by use of Critical Path Tracing VHDL models. In Section 5, method of random generation of test vectors will be described. Section 6 will show a VHDL test bench for test application and fault detection. Conclusions including past work and present direction will be given in Section 7.

2. Test Generation Environment

Our proposed environment consists of a VHDL netlist which will be used in two separate test benches. As shown in Figure 1, the first test bench uses fault collapsing gate models for the generation of a reduced fault list. This test bench generates a list of distinguishable faults. This list will be used by a second VHDL test bench for test vector generation. This test bench uses Critical Path Tracing models for simulation of the circuit and fault detection.

Figure 1. Test Environment

As mentioned above, components of a netlist bound to fault collapsing models will be able to generate a reduced list of faults. This list contains one fault from every equivalent class of faults. An example fault list generated by such models is shown in Figure 2. This figure corresponds to the output labeled “Fault List” in Figure 1.

```
TYPE GATE_ID [ SINGLE STUCK FAULT LIST ]
...
AND2 (108) [1!:none ;2!:sa_1 ;0!:sa_1 ]
OR2 (109) [1!:none ;2!:none ;0!:sa_0 ]
NOR2 (111) [1!:none ;2!:sa_0 ;0!:sa_1 ]
INV (112) [1!:none ;2!:unused ;0!:sa_1 , sa_0 ]
...
```

Figure 2. An Example Fault List

The test environment of Figure 1 generates pseudo random test patterns and applies them to CPT models. The fault list output of the fault collapsing test bench is an input to the test generation test bench. After sufficient number of tests have been applied to the CPT models, the test
application process ceases and a list of undetected faults will be generated. A sample output of this test bench is shown in Figure 3.

<table>
<thead>
<tr>
<th>Tests Applied:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0111110001010111</td>
</tr>
<tr>
<td>011101001000010</td>
</tr>
<tr>
<td>111100010001000</td>
</tr>
<tr>
<td>1000101010000000</td>
</tr>
</tbody>
</table>

Resulting fault coverage: 88%
Faults that remain undetected are:
- AND2 (16) [1: none; 1: 2: sa_1; 1: 0: none]
- OR2 (23) [1: none; 1: 2: none; 1: 0: sa_0]
- NOR2 (45) [1: sa_1; 1: 2: none; 1: 0: none]
- NAND2 (48) [1: none; 1: 2: sa_0; 1: 0: sa_1]

Figure 3. Test generation testbench output

Also shown in Figure 1 are the FC and the CPT models that are used by their corresponding test benches. Details of the FC models were described in a paper that appeared in the VIUF Spring 1994 conference proceedings [9]. Details of the CPT models will be described here.

3. Critical Path Tracing

Critical path tracing is used for finding faults detected by a specific test vector. For this purpose, the test is applied to the inputs of the fault free circuit. This vector determines all intermediate circuit logic values. If a line value is forced to its complementary value, and this value causes the output of the circuit to toggle, then we can conclude that a fault on the line with the new complementary value can be detected at the output. Such line is said to be a critical line of the circuit with the applied test. Finding critical lines becomes more involved in circuits with re-convergent fanouts. To provide necessary background for describing our critical path tracing, an example with and without fanout will be presented.

Figure 4. Example of CPT in a fanout-free circuit

Figure 4 shows a fanout free circuit. The 10111 test at the input of this circuit causes the output value to become 1. Since changing the upper input of the OR gate changes the output value, the line at this input is critical for the applied test vector. Obviously, since the lower input line is not critical, no line that leads to this line is critical. On the other hand, lines leading to the upper input of the OR gate can still be critical. Applying the arguments used with OR gate to the other gates of the circuit, results in critical lines as highlighted in the circuit of Figure 4.

Figure 5 shows a circuit with a re-convergent fanout, with two different test vectors. The argument used in conjunction with Figure 4 for finding critical path lines can be used to mark all the critical lines of Figure 5 except the fanout stem. The decision to mark the fanout stem critical depends on the effects of the stem on the gates influenced by this line. With the 111 input vector toggling the value on the fanout stem does not cause the output to change value. On the other hand, if the input vector is 110, changing the fanout stem value to 0 causes the output to change from 1 to 0. Therefore the fanout stem in Figure 5 with the 111 input test vector is not critical, but the test input of 110 makes the fanout stem a critical line.

Figure 5. CPT with re-convergent fanout

The examples presented above illustrate that there are two key issues in finding faults by critical path tracing. Input lines of a gate whose output is critical can be marked as critical based on the type of the gate and its input and output values. On the other hand, to decide on its stem being critical more than values of fault lines is required. Lines and input values of all gates that fanout branches lead to can participate in the decision to mark a stem as critical. It is not necessary for fanout branches to be critical in order for its stem to be critical. This is another distinguishing feature of fanouts from gates which require their output to be critical if any of their inputs are to become critical.

3.1 CPT METHOD

In the original critical path tracing method [1], primary input values are propagated to all circuit lines. The circuit is traced from output to input for identifying critical lines in a manner similar to what was described in the previous section. In the process of this tracing, when a fanout is encountered, a simulation phase will determine if the effect of changing the value of a fanout stem will be marked as critical. In order to reduce the size of the part of the circuit that is simulated, a partitioning of the circuit is done to simulate only up to the point whose effect on the output is known.

From the above discussion it is clear that critical path tracing by this method requires much forward simulation and backward propagation in an iterative fashion. In addition, partitioning of the circuit into isolated parts with inputs influenced by fanout stem and outputs influencing the primary output is a time consuming process.
Above all, the case mentioned before, in which a fanout stem can be critical even if none of its branches are critical, cannot be handled by the standard CPT method. This is because, the method requires a critical path through branches to reach to a fanout.

3.2. ONE PASS CRITICAL PATH TRACING

Considering the problems associated with the original critical path tracing algorithm, we have developed a new method that eliminates the need for any partitioning or partial simulation of the circuit. In one pass CPT, simulation is only necessary at the beginning when a test vector is applied. Decision to mark a stem as critical will be based on the information that is passed from gates that are driven by the fanout branches back to the fanout node. In general one pass CPT is centered around the facts that 1) A gate with critical output locally decides on its input lines being critical, and 2) Fanout stems require information from all their branches before they can declare a stem as being critical. Therefore gates, interconnecting lines and fanout nodes must cooperate in providing necessary information to gates and fanout nodes.

3.2.1 GATE BEHAVIOR

Based on the type and the output value of a gate, a gate makes certain decisions about its input lines. A gate may declare an input line as being critical or it may declare an input as being blocked from becoming critical by another one of its inputs. The expected behavior of a gate is to identify each of its inputs as being critical or being blocked. In a two input AND gate, shown in Figure 6, if only one of the inputs of the gate has a controlling value (Figure 6-a) that input line is critical and the other is blocked, from being critical.

![Figure 6. AND gate behavior](image)

If both inputs of a two input gate have controlling values, Figure 6-b, then each input is considered as blocked by the other input. Therefore, both inputs will be referred to as being blocked. In this case if both inputs change simultaneously, the output value will toggle. As shown in Figure 6-c, if both inputs of a gate have non-controlling values then both inputs are critical. Since changing each input will change the output value. The significance of a gate in one pass CPT is only due to its critical and blocked input lines.

![Figure 7. Notations](image)

3.2.2 GATE INTERCONNECTION

An interconnection of several gates for critical path tracing can be done by use of a graph using the notation presented in section 3.1. Such a graph will be referred to as critical path graph (CPG). Figure 8 shows a simple AND-OR circuit and its corresponding CPG.

![Figure 8. Critical path graph](image)

The graphs show lines leading to nodes. In this and other graphs shown in this paper, two lines lead to a node, indicating use of two input gates. Lines leading to a node are either Critical or they are Blocked. We will therefore define Kind of line as being CL (critical line) or BL (Blocked line).

![Figure 9. Path kinds](image)

As is Figure 7, solid lines (normal or bold) represent critical lines and dotted lines show blocked lines. Furthermore we define a path as connection of several line segments. A critical path consists of continuous critical lines.
that lead to a primary output. In Figure 8 bold lines signify CLs that are on a critical path. A blocked path (BP) can be blocked by multiple number of gates. Therefore an integer appended to the kind of a path signifies the number of gates that block the path from being critical (BPn, n=1, 2, ...). Path kinds for the example of Figure 8 are shown in Figure 9.

A fanout node will be represented by a filled circle in critical path graphs. Therefore a line can either connect gates or a gate and a fanout. The following paragraphs formalize the definitions of this section.

**CPG**: Critical Path Graph is a graph formed by replacing logic gates with circles and fanouts with filled circles. In such a graph interconnections are identified by lines according to the following definitions.

**Node**: A Node represents a gate or a fanout in a CPG.

**Line**: Lines represent interconnections between gate inputs and outputs as well as fanout edges and stems.

**CL or BL**: Critical or Blocked lines are those identified by a graph node as being critical or being blocked.

**Critical path**: A path is critical if all its line segments are critical and it leads to a primary output.

**Blocked path**: A path is blocked if it contains at least one blocked line. Blocked path n (BPn) is a BP with n blocked lines.

### 3.2.3 FANOUT BEHAVIOR

In addition to gates determining the kind of their input lines, the other decision made in one pass critical path tracing is the decision to mark a fault stem as being critical to not critical. This decision is made by a fanout node and will be based on the types of paths (CP and BPn) associated with fanout branches. The following definition is needed for study of fanout behavior.

**Loop**: A reconvergent fanout not containing another reconvergent fanout is called a loop. Obviously loop has a convergence and a divergence point.

A loop in a CPG can either be removed or replaced by a critical line, according to the following rules.

**Rule a**: A loop containing no BLs at the convergence point and at least one continuous path of CLs between convergence and divergence points can be replaced by a CL starting from divergence point (fanout node) and ending in the divergence point (gate).

As an application of this rule consider the circuit of Figure 10 in which the stem at the divergence point directly influences the output of the gate at the convergence point. This direct relationship can more easily be represented by a straight line from stem to the output. For critical values such a loop and a line have equivalent behavior (Figure 10).

**Figure 10. Rule a replaces two converging critical paths**

**Rule b**: A loop with only one blocked line at the reconvergence point and at least one continuous path of CLs between convergence and divergence points is removed.

**Figure 11. Rule b removes a converging CP and BP**

Justification for this rule can be explained in the circuit of Figure 11. Since the AND gate at the output is blocking the propagation of values from gate b to the output of the circuit, and since values at the output of gates a and b both depend on the fanout stem x, therefore changing value of x causes both a and b output to change. This will reverse values at the input of gate c. Instead of output of b being blocked by a, now output of a is being blocked by b. Therefore, values from x do not propagate to the z output. This is represented by a disjoint CPG as shown in Figure 11.

**Rule c**: A loop with both lines blocked at the convergence point and no other blocked lines is represented by a straight line between divergence and convergence points.

The example of Figure 12 is used for justifying this rule. Since the two inputs of gate c are both 0, it can be said that the two are blocking each other and therefore both can be considered as blocked lines. Changing the value at
the stem from 0 to 1 causes both inputs of gate c to change. This represents a direct dependency between the fanout stem x and output of gate c. This direct relationship can be modeled by a CL line from the fanout stem to the output as shown in Figure 12.

Figure 12. Rule c replaces two converging BPs with a CP

Rule d: A loop with at least a blocked line on each path between divergence and convergence points, except if it is covered by Rule c, must be removed.

The above rules are discussed in conjunction with simple circuits with only one loop between a divergence point and a convergence point. In a more complex structure several interlinked loops can exist. These rules can be applied repeatedly from inner loops to complex structures with divergence and convergence points until all loops are removed or replaced. Determining critical lines in a fanout free circuit can be easily done by the methods described in [1]. Using the above rules we can decide whether a stem is critical or not in only one processing pass.

Figure 13. Applying Rules a, b, c, and d

An example illustrates application of the above rules to a fanout for finding faults detected by an input vector. Circuit shown in Figure 13-a has reconvergent fanouts with three edges at the divergence point and two convergence points. For input ABCDE=01000 all line values are fast calculated. Based on these values and gate characteristics critical and blocked lines will be known, as shown in Figure 13-b. Whether stem B is critical or not will be decided by the application of Rules a, b, and c.

Initially a loop can be observed between the fanout node and gates a, b, and d. Applying Rule c to this loop reduces it to a straight line, and therefore converting the graph of Figure 13-b to that of Figure 14-a. This graph also contains a loop, for the reduction of which Rule a can be applied. The resulting graph is shown in Figure 14-b.

The reduced graph illustrates that the stem is directly connected to the output. We will therefore conclude that stem is critical. Therefore ABCDE=01000 detects six faults in circuit of Figure 13-a including an se-0 at stem B.

Figure 14. Continuing example of Figure 13

3.3. VHDL IMPLEMENTATION

The above rules and their application to logic circuits can be easily implemented in a concurrent programming environment. The above discussion facilitates the presentation of VHDL gate models for performing critical path tracing.

After the initial simulation, and after several delta times, all node values will be known. Based on its output values each gate produces CP or BP on its inputs. Primary outputs, being always on a critical path, initiate critical value determination of the output gates. Therefore gate models may observe a CP transaction on their outputs. Such a gate will decide CP or BP on its inputs. This information will eventually be collected and observed at a fanout node. In order to keep track of paths leading to a fanout, each gate collectively send its identification along with the kind information (Critical or Blocked) on its input line signals.

Faults detected are reported by the individual gates and PIs, and POs. Fault detected at a fanout stem will be reported by gates or PIs immediately connected to it. In the CPT implementation for pseudo random test generation, reporting is done by crossing out detected faults in a shared variable structure that lists all circuit faults that are to be detected. The VHDL implementation of critical path tracing consists of types, resolution function, gate models and utility programs. In this section each of these parts will be described. The report shared variable will be addressed in a later section. Before this description, we will make
3.3.1 INFORMATION FIELDS

Interconnecting lines between gates carry information in the forward and backward directions. In the forward direction only logic values are transmitted. In the backward direction information regarding critical, blocked, and the index n number of blocked paths (CP, BnP) are transmitted. The logical values will be referred to as logic, and the information needed for critical path algorithm will be referred to as path_info.

3.3.2 CRITICAL LEVELS

Path_info of a node whose forward logical value is not known (U) is none. The code value for this state is -3. At the beginning of the simulation all line path_info is none. After an input vector is applied to the primary inputs, logic values propagate to the outputs changing logic values to 0 or '1', while keeping path_info values to none. The process of determining path_info begins with outputs identifying themselves as critical (CP). CP is a value that is assigned to the path_info and has a code value of 0. Path_info propagate backwards from output towards the inputs of the circuit. On their way, all lines will receive different values of path_info. If an input of a gate is blocking critical path of another input, its path_info becomes Bp which has a code value of 1. A Bp blocked by another input becomes Bp+1, increasing its code value by 1.

The level of a path_info is determined by its code value. Therefore, CP with level 0 has the lowest level and a BpS with level 5 has the highest level. Two adjacent levels are those whose codes are consecutive numbers of 0, or above (i.e., none is not included).

If a line is neither critical nor it is blocking a critical path, it is UNCP and has a code value of negative 2. A similar line, but one that also contains fanout stem information has a code of -1 and will be referred to as UNCP.I.

3.3.3 LINE TYPE

A line type for containing logical and path information is shown in Figure 15. This is a record named fsim with logic and path_info elements.

<table>
<thead>
<tr>
<th>TYPE</th>
<th>fsim IS RECORD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>logic : imp; path_info : bkt; END RECORD;</td>
</tr>
</tbody>
</table>

Figure 15. Node record

The logic element of an fsim record takes values 0', 1', and 'U'. Initially all line logic values are 'U'. Assigning a value 0' or 1' to any of several connecting lines will change the value of all such lines accordingly. The imp resolved type uses the implication resolution function and it is responsible for this behavior.

The path_info element is an array of 4 rows and 20 columns of integers. Path information on each line is included in this array. In each row, the first integer specifies a code value of a path. Therefore a line that is part of a critical path has a 0 value in position (0,0) of the path_info array. The rest of the integers in a row of this array contain gate identifications for all gates that a critical path has gone through. The other three rows contain similar information which will be described in the following sections.

<table>
<thead>
<tr>
<th>ENTITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARCHITECTURE</td>
</tr>
<tr>
<td>PROCESS</td>
</tr>
<tr>
<td>Forward Mode:</td>
</tr>
<tr>
<td>Assign Logic Values</td>
</tr>
<tr>
<td>Backward Mode:</td>
</tr>
<tr>
<td>Assign Path Information</td>
</tr>
<tr>
<td>Mark Detected Faults</td>
</tr>
<tr>
<td>Reinitialize</td>
</tr>
<tr>
<td>GO TO Forward Mode</td>
</tr>
</tbody>
</table>

END

Figure 16. General CPT model structure

The path_info element of the fsim record is resolved by the backtracing resolution function (bkt). For non fanout nodes, the role of this function is to pass path information from an input of a gate to output of another. For fanout nodes this function will make decisions regarding passing of path information to fanout stems. The details of this function will be described in a later section.

3.3.4 GATE MODELS

A gate model has inputs and outputs of type fsim and INOUT mode. A model is responsible for forward propagation of logic values, and backward (output to input) propagation of path information. A model is also responsible for reporting its inputs and outputs on which faults are being detected. Each gate has a unique identification number passed to it via a generic clause. A single process in the statement part of a gate architecture handles logical and path information generation. In the following discussion we will use a two input NAND gate for demonstrating the behavior of critical path tracing models. A general outline of a gate model is shown in Figure 16.

3.3.4.1 LOGIC OPERATION

The code for generation of logic value on the output of a gate is shown in Figure 17. After none 'U' values
are observed on all gate inputs, a gate uses logic field of its inputs and assigns proper value to the logic field of its output. At this time the gate goes in the backward mode and will only be sensitive to events on the path info field of its output.

\[
\begin{align*}
\text{WAIT UNTIL } & i1.\text{logic } = 'U' \text{ AND } i2.\text{logic } = 'U'; \\
& \text{in1 := i1.logic; in2 := i2.logic;} \\
& \text{out1 := in1 NAND in2;} \\
& \text{o.logic <= out1;}
\end{align*}
\]

Figure 17. Partial code of a gate behavior

When fault values for specific test vector have been crossed out from the shared variable fault list, gates will again go into forward mode and again become sensitive to events on their inputs. The process of re-initialization which operates after a gate has marked off its detected faults is shown in Figure 18.

\[
\begin{align*}
\text{-- initialization} \\
& i1 := \text{none}_U; \\
& i2 := \text{none}_U; \\
& o := \text{none}_U; \\
\text{-- wait until all lines in circuit initialized.} \\
& \text{WAIT FOR 10 PS;}
\end{align*}
\]

Figure 18. Partial code of a gate behavior

### 3.3.4.2 Generation of Path Information

A gate waits until an event on its output changes the path info field of the output to a value other than none. This happens when a code value greater than -1 is placed on the path info of an output. The code value of a path info greater than -1 is an indication that the output is either on a critical (CP), or on a blocked path (BP). In this case, based on input logic values, path information for the path info fields of the inputs will be determined and assigned to the fields of the individual inputs. The code section responsible for this behavior is shown in Figure 19.

\[
\begin{align*}
\text{ELSIF } & \text{temp_o(1,1) > -1 THEN} \\
\text{IF } & \text{in1 } = 'U' \text{ AND in2 } = 'I' \text{ THEN} \\
& \text{temp_i1 := added_node(temp_o,id);} \\
& \text{temp_i2 := increment_depth(temp_o,id);} \\
\text{ELSIF } & \text{in1 } = 'I' \text{ AND in2 } = 'U' \text{ THEN} \\
& \text{temp_i1 := increment_depth(temp_o,id);} \\
& \text{temp_i2 := added_node(temp_o,id);} \\
\text{ELSIF } & \text{in1 } = 'U' \text{ AND in2 } = 'U' \text{ THEN} \\
& \text{temp_i1 := increment_depth(temp_o,id);} \\
& \text{temp_i2 := temp_i1;}
\end{align*}
\]

Figure 19. Generating path information

This code is reached when an event occurs on the path info field of the output. At this point, the path info includes all the path information accumulated in the backward direction into the output of the gate. A path info on the output may look like the record shown in Figure 20. This illustrates a critical path that has been marked critical by gates 1, 2, and 5.

\[
\begin{array}{|c|c|c|c|}
\hline
\text{CP} & 1 & 2 & 5 \\
\hline
\text{none} & & & \\
\hline
\text{none} & & & \\
\hline
\text{none} & & & \\
\hline
\end{array}
\]

Figure 20. A path information example

The path info assigned to the individual inputs is the output path info modified based on input logic values. Assume the record of Figure 20 is the path info at the output of a NAND gate. In this gate, if in1 logic value is 'U' and in2 is 'I', in1 is on critical path, and in2 is on a blocked path.

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{CP} & 1 & 2 & 5 & 6 \\
\hline
\text{none} & & & & \\
\hline
\text{none} & & & & \\
\hline
\text{none} & & & & \\
\hline
\end{array} \quad \quad \begin{array}{|c|c|c|c|}
\hline
\text{BPI} & 6 & 1 & 2 & 5 & 6 \\
\hline
\text{none} & & & & & \\
\hline
\text{none} & & & & & \\
\hline
\end{array}
\]

Figure 21. Path info for in1, in2 being 0,1

The code of Figure 19 shows that for in1, the added_node function is called to form path info. This function appends the id number of the present gate to the path info and keeps the code value of the output the same. The new path info is assigned to the path info of in1. For in2 which is a BP (blocked path) the increment depth is called. This function copies the first row of path info into the second row, increments the code value (level) of the first row, and appends the id of the present gate to both rows of the path info. Incrementing CP (code 0) makes it BPI (code 1) and incrementing BPI makes it BPI+1. Figure 21 shows path info for in1 and in2.

\[
\begin{align*}
\text{IF } & \text{temp_o(1,1) } = -1 \text{ THEN} \\
\text{IF } & \text{in1 } = 'I' \text{ AND in2 } = 'U' \text{ THEN} \\
& \text{temp_i1 := uncp; temp_i2 := temp_o;} \\
\text{ELSIF } & \text{in1 } = 'U' \text{ AND in2 } = 'I' \text{ THEN} \\
& \text{temp_i1 := temp_o; temp_i2 := uncp;} \\
\text{ELSIF } & \text{in1 } = 'U' \text{ AND in2 } = 'U' \text{ THEN} \\
& \text{temp_i1 := uncp; temp_i2 := uncp;} \\
\text{ELSE} & \\
& \text{temp_i1 := temp_o; temp_i2 := temp_o;} \\
\end{align*}
\]

Figure 22. Partial code deciding on a non critical fanout

Assignment of values to path info of inputs when they are 'I' or 'U' is done similarly. For in1, in2 being '00'
increment_depth procedure is called for both inputs since each input is being blocked by the other input. For in1, in2 being ‘11’ added_node function, which appends the current gate id to the output path_info while preserving the code value of the output, is called for both inputs.

Path information propagates from gate to gate as described in the previous discussion. When a path_info reaches a fanout, the resolution function at the fanout decides the code value at the stem. A CP path_info with code value 0, becomes a CP on the stem. Since the notation of BPn only applies to gates with an input blocking another, and this notation does not appropriately apply to the fanouts. Therefore, we will use UNCP for a fanout stem that is not CP. The BPn information reaching a fanout may still be needed by other fanouts for deciding their critical values. For this reason an UNCP value at a stem will carry such information along. An UNCP with such information will be referred to as UNCP_I. If an UNCP_I is blocked, it looses its gate information and becomes UNCP. The partial code of Figure 22 shows the behavior of a gate model when UNCP or UNCP_I path information reach its output.

3.3.5 FANOUT RESOLUTION

Although, all interconnecting lines are resolved, resolution of value is only significant at the fanout nodes, and that is only important when processing path information in the backward direction. The resolution function for this purpose is called backtracing which will be discussed here.

Drivers of the backtracing resolution function are path_info that reach a fanout from various gates. The resolution function makes a copy of all its drivers in a temporary buffer and starts processing and making modifications to this buffer using rules a, b, c of Section 3.2. Initially the resolution function searches for a path record which has the highest code value of all the other drivers. The highest code value corresponds to a path with most number of blockages. This path being blocked the most, represents part of an innermost loop. The part of the resolution function searching for the deepest BPn is shown in Figure 23.

FUNCTION backtracing (paths : int_4by20_vector)
RETURN int_4by20 IS
VARIABLE
    temp1, temp2 : int_4by20_vector(paths\RANGE);
VARIABLE depth, util1 : INTEGER := 0;
    ...
BEGIN
    ...
FOR i IN paths\RANGE LOOP --Finding deepest path
    IF depth < temp1(i)(1,1) THEN
        depth := temp1(i)(1,1);
    END IF;
END LOOP;

Figure 23. Resolution function, finding deepest path

In the discussion that follows, CP and BP definitions of Section 3.2 will be used. The following points from the definitions need to be elaborated upon. The level of a CP is always 0. A BPn is a blocked path with n number of blockages, this refers to a path through several gates with a gates blocking it from being critical. The number n is the level of a BP and this number is at least 1. The deepest BPn is one with the highest level (n). Decreasing the level of a BPn results in BPn-1. Decreasing the level of a BP1 results in a CP because the level of CP is 0.

The partial code shown in Figure 24 becomes active when at least a BPn is found on one of the edges of the fanout stem. In this case, starting from the path information with the deepest BPn, the temporary buffer of all drivers is searched for another path information with an equal BPn. If such a driver is found then the replace_check function determines if the two path_info have a common node at the convergence point. If so, using rules a or c these path_info will be replaced by BPn or BPn-1 in the temporary buffer. If rule a is used, the two paths will be replaced by a path with same level, while application of rule c causes the replacement path level to become n-1.

Figure 24. Partial code for the execution of rules a and c

IF depth > 0 THEN ------- construct and reject
    FOR i IN depth DOWNTO 1 LOOP
        -- path with i depth will be constructed.
        FOR j IN paths\RANGE LOOP
            IF temp1(j)(1,1) = i THEN
                FOR i IN j + 1 TO (paths\LENGTH - 1) LOOP
                    IF temp1(i)(1,1) = i THEN
                        replace_check (temp1(i), temp1(l), constructed, connection);
                        IF connection THEN
                            temp1(l) := constructed;
                            temp1(l) := none;
                        END IF;
                        END IF;
                    END LOOP;
                    END IF;
                END LOOP;
                END LOOP;
            END IF;
        END LOOP;
    END LOOP;
END IF;

FOR j IN paths\RANGE LOOP
    -- path with i-1 depth is rejected
    IF temp1(j)(1,1) = i THEN -------by pass i depth.
        FOR i IN paths\RANGE LOOP
            IF temp1(i)(1,1) = (i - 1) THEN
                remove_check (temp1(j), temp1(l), constructed, connection);
                IF connection THEN
                    temp1(l) := constructed;
                END IF;
            END IF;
        END LOOP;
    END IF;
END LOOP;
END IF;
END IF;
END LOOP;
END LOOP;
END IF;

4.8
After application of rules a and c, as above, the resolution function searches the temporary buffer for remaining BPl path info that have not been replaced by BPl-1. In this case, if a BPl-1 is found that has a common node at the convergence point with the BPl under consideration, then the two path info will be removed from the temporary buffer. This removal process is done by performing rules b and d, which are checked by the remove_1_check function.

The above processes, in which rules a, c and b, d are applied, will be repeated for all path info from highest to the lowest blocked level (BPl). When this is done, if the resulting path info in the temporary buffer has level 0 (CP), then the fanout stem is critical, otherwise the fanout stem will be declared as being un-critical (UNCP_I).

4. Random Test Generation

As shown in Figure 25 the test generation testbench consists of a test controller and the circuit netlist. As described in the CPT section, the primary outputs have a special role in the activation of the CPT gate models and are instantiated separate from the circuit netlist.

![Figure 25. Block diagram of testbench VHDL code](image)

The test controller reads data generated by the fault collapsing test bench (see Figure 1), generates pseudo random test data and applies this data to the circuit under test. As noted in the CPT section, the circuit CPT models mark off the detected faults from a shared variable that contains a list of faults reduces by the fault collapsing process. The general outline of the test controller is shown in Figure 26.

![Figure 26. Tasks performed by the test controller](image)

The test controller is implemented as a VHDL architecture consisting of several processes that utilize several utility packages. We will briefly describe tasks performed by this architecture. Initially the test controller reads an ASCII file of fault collapsing data and loads the fault information in a shared variable linked list. The shared variable is called fault_table and is declared as shown in Figure 27. Faults on inputs and output of a gate are loaded into a variable of g_fault type.

In addition to the fault information, the total number of faults is also obtained from fault collapsing file and read into a shared variable. This value will be used for calculation of fault coverage.

Another process statement in the test controller is responsible for the generation of pseudo random test vectors. This process uses a polynomial and a seed as input for the PRPG shift register. With each clock a new test is generated.

```vhdl
TYPE g_faults;
TYPE g_faults_ptr IS ACCESS g_faults;
TYPE g_faults IS RECORD
  g_type : STRING (1 TO 6);
  id : POSITIVE;
  in_1 : faults; -- inverters use in1
  in_2 : faults;
  out_1 : faults;
  link : g_faults_ptr; --for linked list;
END RECORD;

SHARED VARIABLE fault_table :
  g_faults_ptr := NULL;
```

![Figure 27. Gate fault pointer and fault table linked list](image)

The third process shown in Figure 26 handles test application to the CPT models of the circuit under test. This process applies none_U (see Figure 18) values to the inputs of the circuit in order to initialize the circuit and prepare it for the a new test vector. This is followed by application of a test vector from the output of the PRPG process statement to the logic fields of the circuit inputs. This test is also recorded so that it can later be reported along with the information regarding the undetected circuit faults. At this point the test controller becomes idle waiting for the CPT models to propagate test data forward and critical information to be propagated backward into the test controller.

As the critical information move in the backward direction towards the test controller outputs, the CPT gate models mark off detected faults in the shared variable fault linked list (Figure 27). A gate marks off a detected fault on its in_1, in_2, or out_1 by setting the appropriate field of its corresponding g_fault record to value n.

When the test controller receives the information on its outputs from the CPT gate models, its lines starts behaving as primary inputs and crosses out faults that are detected on these lines.
Before repeating the process of initializing the CPT models and applying a new test vector, the test controller checks for a target fault coverage. If this is satisfied, the test process terminates. At this time the list of undetected faults is extracted from the fault linked list and reported to an output file. An example of this file is shown in Figure 2. Other information such as the number of accomplished fault coverage are also reported to this output file.

If the test process is not to terminate, the PRPG clock is activated for it to generate another random test data. Reaching a given fault coverage or exhausting all random test data are two reasons for the test process to terminate.

5. Conclusions

We have shown VHDL models for fault collapsing and the use of such models for pseudo random test generation. The emphasis here was on the CPT method and its implementation. More work is being done on the pseudo random test generation and will be reported in a later paper.

This work is part of an on-going research on VHDL modeling strategies for test related applications and environments utilizing such models. In this work gate models for tasks such as test generation, fault collapsing, and fault simulation are being developed.

REFERENCES:


4.10