An Automatic Test Bench Generation System

Shekhar Kapoor, James R. Armstrong, Sanat R. Rao

The Bradley Department of Electrical Engineering
Virginia Polytechnic Institute and State University
Blacksburg, VA 24061

Abstract

This paper presents an automatic test bench generation system for VHDL behavioral models. Modeler’s Assistant, an interactive CAD tool developed at Virginia Tech, gives the graphical representation of a VHDL behavioral model, called a Process Model Graph (PMG). The Process Test Generator (PTG) is used to generate the stimulus/response test sets for individual processes of a PMG. The Hierarchical Behavioral Test Generator (HBTG) accepts the PMG and the test sets produced by PTG as inputs, and then hierarchically constructs a test sequence for the entire model. The test sequence is converted into a test bench by the Test Bench Generator (TBG), and it is then used for simulation of the model. Experimental results show that the test benches generated exercise the models thoroughly.

1. Introduction

Design verification using a behavioral description of a circuit is becoming extremely popular in the electronics industry. Hardware Description Languages (HDLs) provide a convenient method by which the functionality of a digital circuit can be represented by its behavioral model. VHDL is widely used to model the behavior of a circuit accurately in either top-down or bottom-up fashion at many levels of abstraction [2]. After the VHDL behavioral model of a circuit is developed, test data is generated for simulation of the model and the simulation results are observed to verify the functionality of the model. The general practice is to simulate a model with a very large amount of test data given by a designer. The generation of this test data is a very time-consuming and a labor-intensive task [3]. Also, the test patterns constructed by a modeler may not test all the operations of a circuit. A number of hierarchical techniques have been proposed [4-8] that use high-level primitives for test generation. In some of these approaches [5, 7], test information for high-level modules is stored in the form of stimulus/response sets which are then used to generate tests for the entire model.

In this paper, we present an automatic test bench generation system that exploits the hierarchy information in a VHDL behavioral model and automatically produces a test bench that relieves the modeler of the time-consuming task of test data development. The test sequence generated by the system exercises the functionality of the circuit thoroughly and makes the design verification process considerably faster.

A Process Model Graph (PMG) is used to represent graphically the division of functionality within a VHDL behavioral model. The PMG gives an interconnection of different processes that partition the model into inter-related but distinct functional modules. The VHDL source for the model is given by the Modeler’s Assistant. The test data file for each process is generated separately by the Process Test Generator (PTG) program. The PTG takes the VHDL source of the process as input and constructs a Control Flow Graph (CFG) that gives the flow of information in the VHDL description. It then activates different sensitive ports of the model, sensitizes different paths along the graph, and generates stimulus/response test sets for each process. The Hierarchical Behavioral Test Generator (HBTG) extracts the graphical interconnection and other design information from the PMG database, and the functionality information from the test data file for each process. It then hierarchically generates a test sequence for the whole entity based upon the process test data files and the PMG inter-connection information. The final test sequence generated by HBTG is converted into a VHDL test bench by the Test Bench Generator (TBG) program. The resulting test bench is used for simulating the model. A method for evaluating the tests generated by the proposed system is also presented. It is observed that the tests generated exercise the models fully [3].
2. System Overview

Figure 1 shows a block diagram of the complete Test Bench Generation System under development at Virginia Tech. The base of the entire system is the interactive graphical CAD tool, called the Modeler's Assistant [9]. The Modeler's Assistant develops a model using a Process Model Graph (PMG). Figure 2 shows a PMG generated by Modeler's Assistant for an example circuit. The PMG is a directed graph; the nodes represent the VHDL processes and the edges correspond to signals between the processes. A process is represented by a bigger circle while the ports\(^1\) of a process are represented by smaller circles along the circumference of the bigger one. A "shaded" port represents a sensitive port\(^2\).

In the example circuit of Figure 2, the entire function of the circuit is divided into four processes: INV, an inverter. MUX, a 2 to 1 multiplexer. AND, an and gate, and REG, a register. The VHDL code specifying the functionality of each process is selected from a process primitive library or is input textually by the user. Given the PMG and the functionality of each process, the Modeler's Assistant generates the VHDL behavioral model for the circuit. Figure 3 shows the VHDL code output by Modeler's Assistant for model represented by the PMG in Figure 2. The information about the geometry of the PMG [3], i.e., the information about the various graphical constructs like processes, ports, signals and variables, and the interconnection between them, is stored in the PMG database. This information is used by HBTG (to be discussed later) for test generation.

The functionality information required for test generation is given to HBTG in the form of symbolic test data files for each process. These test data files are generated by PTG. The VHDL source for each process in the PMG is analyzed with the help of VTIP/DLS and SPI software (CLSI/COMPASS) [10]. The software analyzes the model and stores the design information in an intermediate form in a DESIGN LIBRARY SYSTEM (DLS) [10]. The PTG extracts the design information from DLS and generates a Control Flow Graph (CFG) describing the data and control flow information in a process. The CFG is then used to generate test sets for each process. The stimulus/response test data files produced by PTG for each process are put in the Primitive Test Library to be used for hierarchical behavioral test generation by HBTG. The test sequence produced by HBTG is then used to simulate the model. The following sections describe the PTG algorithm, the HBTG algorithm and the TBG. The test sequence evaluation system is described at the end.

---

use WORK_VHDLCAD.all, WORK_USER_TYPES.all;

---

entity EXAMPLE is

  port (L: in BIT;
     D_OUT: out BIT_VECTOR(7 downto 0);
     LD: in BIT;
     CLK: in BIT;
     B: in BIT;
     A: in BIT;
     IN2: in BIT_VECTOR(7 downto 0);
     IN1: in BIT_VECTOR(7 downto 0));

end EXAMPLE;

---

1 A port is defined as an input or an output of a process node of a PMG.
2 A sensitive port is a port such that a change in the value on it triggers the process and the statements within the process block are executed.
architecture BEHAVIORAL of EXAMPLE is
signal SEL: BIT;
signal CLR: BIT;
signal D_IN: BIT_VECTOR(7 downto 0);

begin

-- Process Name: INV
INV_4: process (L)
begin
  SEL <= L;
end process INV_4;

-- Process Name: REG
REG_8: process (CLK,CLR)
begin
  if CLR = '1' then
    D_OUT <= "00000000";
  elsif CLK'EVENT and CLK = '0' then
    if LD = '1' then
      D_OUT <= D_IN;
    end if;
  end if;
end process REG_8;

-- Process Name: AND
AND_15: process (A,R)
begin
  CLR <= A and R;
end process AND_15;

-- Process Name: MUX
MUX_20: process (SEL)
begin
  if SEL = '0' then
    D_IN <= IN1;
  else
    D_IN <= IN2;
  end if;
end process MUX_20;
end BEHAVIORAL;

3. Process Level Test Generation

Several research efforts have concentrated on test generation using data flow graph and control flow graph techniques [6, 11, 14]. In this section, a high-level process test generation algorithm, called Process Test Generator (PTG), is described which derives a Control Flow Graph (CFG) from the VHDL behavioral description of a circuit and uses it for process test generation. The VHDL description of the process for which the stimulus/response test sets are to be generated, is analyzed with the help of the VTIP VHDL analyzer (CLSI/COMPASS). The analyzed design information is stored in the DESIGN LIBRARY SYSTEM (DLS) of the VTIP software in an intermediate form. The PTG extracts the useful design information from the DLS and constructs a CFG for the VHDL behavioral model.

3.1 Control Flow Graph (CFG) Generation

Definition: A Control Flow Graph (CFG) is a directed graph \( G(N, E) \) consisting of a set of nodes \( N \) and a set of directed edges \( E \) such that \( E_{ij} \in E \) denotes a directed edge from the head node \( N_i \in N \) to the tail node \( N_j \in N \) [12].

Definition: A Node \( N_i \) is defined as an element in a Control Flow Graph which represents a control statement or an assignment statement within a VHDL process.

Definition: A Directed Edge \( E_{ij} \) is defined as a pair of nodes \( (N_i, N_j) \) such that the node \( N_i \) precedes the node \( N_j \) during the execution of the CFG.

In this work we use the CFG to represent the flow of data and control information in the VHDL description of a process. The direction of data and control flow information is given by the directed edges defined by the fields of the data structures representing the nodes. The directed edges are of two types: a true edge and a false edge, corresponding to a true condition or a false condition.

Some properties and important definitions related to a CFG are:

- Between any two nodes there is at most one directed edge.
- A path between two nodes \( N_i \) and \( N_j \) is a sequence of nodes \( N_{i_1}, N_{i_2}, N_{i_3}, \ldots, N_{j} \) such that any two successive nodes are connected by a directed edge. The path length of a path between two nodes \( N_i \) and \( N_j \) is the number of nodes in the path including the nodes \( N_i \) and \( N_j \). The shortest path \( S_{P_{ij}} \) between

Figure 3. VHDL code of Example Circuit
nodes $N_i$ and $N_j$ is defined as the one with the minimum path length between the two nodes.

- There is a **CFG node** for every VHDL statement inside a process within the architecture body of a VHDL behavioral model. The VHDL statement can either be a control statement or it can be an assignment statement and accordingly the corresponding CFG node is either defined as a **Control Node** or an **Assignment Node** respectively.

Besides the control and the data flow information, the PTG algorithm requires other information, e.g., the declarations of ports, internal signals and variables, and the related information like the type and range of a signal or a variable. Accordingly, **Declaration Nodes** and **Signal/Variable Nodes** are defined which complement the information stored in CFG nodes and thus give a complete **Directed Model Graph (DMG)** representing the information to be used for process test generation.

The VHDL model of the process REG in our example circuit, as shown in Figure 2, is analyzed with the help of VTIP VHDL analyzer. The output listing with the source lines numbered, as shown in Figure 4, is produced when the model is analyzed with a "-list" qualifier option [10]. A complete DMG produced by PTG for the VHDL model of the process REG is as shown in Figure 5. The nodes MOD_TOP, ENTITY_TOP, ARCH_TOP and PROCESS_TOP are the declaration nodes forming the loci of information for the model, entity, architecture and process declarations respectively. The nodes shown as PORT SIGNALS and SENSITIVITY LIST SIGNALS are the signal nodes holding the information about the type, range, mode and the sensitivity of process port signals. The CFG nodes are shown by the shaded circles. The CFG nodes 0, 2 and 3 are control nodes corresponding to the lines 9, 11 and 12 respectively in Figure 4. The CFG nodes 1 and 4 are assignment nodes corresponding to lines 10 and 13 respectively.

```vhdl
2   9 | if CLR = '1' then
2   10 | D_OUT <= "00000000";
2   11 | elsif CLK EVENT and CLK = '0' then
2   12 | if LD = '1' then
2   13 | D_OUT <= D_IN;
2   14 | end if;
2   15 | end if;
2   16 | end process REG_1;
1   17 | end BEHAVIOR;
```

---

**Figure 4. VHDL source for process REG**

**Figure 5. A Complete DMG for process REG**

### 3.2 Format of the process test data file

We have already stated that HBTG accepts the PMG of a unit and the process test data files as inputs, and then hierarchically constructs a test sequence for the entire circuit. The test sets for input to HBTG are generated by PTG. Various symbols are used to represent signal values or transitions in signal values while writing the process test sets. The symbolic notation used is described in Table 1, as shown on the next page [3].

Figure 6 shows the process test data file for the process MUX in our example circuit. In the figure, test sets are preceded by the number of time frames required for each of the test sets [3].
Table 1. The Symbolic Notation for Signals

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'0'</td>
<td>A constant '0' in a single-bit bus.</td>
</tr>
<tr>
<td>'1'</td>
<td>A constant '1' in a single-bit bus</td>
</tr>
<tr>
<td>'R'</td>
<td>A transition from '0' to '1'</td>
</tr>
<tr>
<td>'F'</td>
<td>A transition from '1' to '0'</td>
</tr>
<tr>
<td>'X'</td>
<td>An unknown value</td>
</tr>
<tr>
<td>'Z'</td>
<td>A high-impedance state</td>
</tr>
<tr>
<td>'D_i'</td>
<td>Any chosen value in either a single-bit bus or a multi-bit bus, e.g., D1, D2, D3, etc.</td>
</tr>
<tr>
<td>'P'</td>
<td>A value which is the same as the last value of the signal in the previous time-frame</td>
</tr>
<tr>
<td>'C_i'</td>
<td>A value used to represent control signals, like the output of a counter. The values can be C1, C2, C3, C4, etc. These values are hierarchically defined, i.e., C_{i+1} always follows C_i in a sequence</td>
</tr>
<tr>
<td>'CX'</td>
<td>The next value in the hierarchical sequence of control signals</td>
</tr>
</tbody>
</table>

The salient features of a process test data file are [3]:

- For each sensitive port of the process, there must be a test that generates an event on that port.
- In a particular time frame, only one port should have an event. All the other ports should be assigned constant stable values.
- For each sensitive input port, it is also required to have a two time-frame test set that generates both an event on the port and a stable constant value on it. This is used during the justification phase of HDTG to justify any signal values assigned to a signal in the process during forward propagation.
- In addition, certain special symbols like 'P', 'C_i' and 'CX' are used, as defined in Table 1, which are required for dealing with sequential circuits [3].

Figure 6. Process Test Data File for process MUX

In Figure 6, when the signal SEL rises, the output OP gets the value 'D2' from the input IN1. When there is a fall 'F' on SEL, OP gets assigned the value 'D3' from the input signal IN2. The two time-frame test sets give the tests that generate both an event on the sensitive signals and the constant stable values on them.

3.3 Process Test Generation Algorithm

In the case of the PTG algorithm, no explicit fault model is assumed. The PTG tries to generate test sets that fully exercise the functionality of a process. The main criterion used by the PTG algorithm is: The test sets generated should be such that every assignment node of the CFG is passed (or executed) at least once.

We know that a process inside the architecture body of a VHDL behavioral model is executed only when there is an event on one of its sensitive input ports [1]. The PTG algorithm uses this fact to create stimulus/response test sets. The algorithm selects each sensitive signal one by one and searches for the first CFG node, that is associated with the sensitive signal, with the help of a hash table. The hash table is implemented as an array of fixed size and is used to transform a signal name into a table index directly. All the CFG nodes associated with a signal are linked together in a link-list and are also linked to the table at the index corresponding to the signal name, as given by the hash function. The hash table organization eliminates most unnecessary comparisons and speeds up the test generation algorithm. The CFG node selected to create an event on the associated sensitive signal is called a **Controlable Node** (CN). The algorithm checks to see if the node has already been chosen as a CN. If it has already been chosen, it selects another node which is associated with the sensitive signal and which has not been chosen as yet, as the CN. A suitable event is then created on the sensitive signal. The stimulus given to the CN has to be propagated forward to an output where the response can be observed. The PTG searches for a CFG node associated with an output port, with the help of the hash table. If a path exists from the CN to this node, the node is selected and is called an **Observable Node** (ON). The PTG then finds the shortest path SP from CN to ON, called the **forward propagation path** (FPP). The shortest path algorithm used is a modification of Dijkstra's shortest path algorithm [13]. The PTG also finds a shortest path from the first CFG node (or the start node SN, i.e., node number 0) to the CN. This path is called the

3A CFG node, corresponding to a VHDL statement, is said to be associated with the signals or variables that occur in the statement.
justification path (JP) and is used to justify the event created on the sensitive signal.

Once the CN, the ON, the JP and the FPP have been found, the PTG algorithm determines the other unknown input signal values along these paths that result in the sensitization of the paths, justification of the event on the sensitive signal and the propagation of the event to the observable node. The user is then prompted to assign any undetermined input data values. All the nodes along the JP and the FPP are marked as passed. Also, CFG nodes which are executed because of true conditions on certain input signals that are determined along the JP and the FPP, are also marked as passed.

Once all the input signal values are known, these values are put in a test bench. The test bench and the VHDL behavioral model of the process are analyzed and simulation is performed. The simulation results are parsed to find out the values assigned to the output ports which give the responses corresponding to the stimulus given to the sensitive signal earlier. This completes the activation of the selected paths. The whole process is continued till all the input sensitive ports of the process have been activated.

As already mentioned, the main criterion for process test generation is that every assignment node of the CFG is executed at least once. After all sensitive ports have been activated, the PTG algorithm checks to see if all the signal or variable assignment statements have been executed. If there is an assignment statement not yet executed, the algorithm selects the CFG node corresponding to this statement as an observable node (ON) and constructs a new path from the start node (SN) to this node. An event is created on the first sensitive signal found along this path. The other input signal values are also determined along this path; all the input signal values are put in a test bench and the model is simulated again. The simulation output file is parsed and the responses are stored. This continues till all assignment statements are executed at least once.

The stimulus/response test sets generated by PTG consist of the actual values given to the input signals and the actual values obtained for the output signals after simulation. But the input to HBTTG is in the form of a test data file for each process consisting of the stimulus/response test sets that use a symbolic notation for data values, as shown in Table 1. In order to map the actual data values to specific symbols, the PTG prompts the user to assign various symbols to the actual data values from the set of values given in Table 1. After the mapping is done, the PTG algorithm writes the symbolic test sets in the process test data file and the process test generation is completed.

3.4 PTG Example

In this section, we discuss the generation of process test sets for the process REG in our example circuit of Figure 2. The source listing of the VHDL behavioral model for the process REG is shown in Figure 4. When the signal CLR is '1', the output D_OUT of the register gets cleared. When CLR is '0', LD is '1', and there is an event FALL (transition in value from '1' to '0') on CLK, the data value at the input D_IN of the process REG is assigned to the output D_OUT. The CFG for the process REG is redrawn in Figure 7. The process test data file generated by PTG for the process REG is shown in Figure 8.

Figure 7. The CFG for process REG

The PTG algorithm picks up the first sensitive signal from the sensitivity list of the process REG. The first sensitive signal selected is CLK. The CFG node number 2 is selected as the controllable node (CN) and an event fall is created on the signal CLK. In order to justify the assigning of event 'F' on CLK, a justification path (JP) is build up. The CFG node 2 will be passed only if the condition on CFG node 0 is false. The PTG thus constructs a JP comprising of nodes 0 and 2, with the help of shortest path algorithm. The assigning of values to the input signals along JP to justify the execution of node 2 results in assigning a value '0' to CLR. In order to observe a response at an output corresponding to the event on sensitive input CLK, the event has to be propagated forward. The PTG picks up node 4 as an observable node (ON) and constructs a forward propagation path (FPP) from the controllable node 2 to the observable node 4 comprising of node numbers 2, 3 and 4. The propagation of the event to the ON results in assigning a value '1' to the signal LD. The only input signal with an unknown value left is the signal D_IN. The user is prompted to assign a real bit vector value and also a corresponding symbolic value to the signal. As has already been discussed, the symbolic signal values (i.e., D1, D2, etc.) are required
because the stimulus/response test file input to HBTG uses a symbolic notation for data values. Let the symbolic value assigned to D_IN by the user be 'D2'. All the input signal values are now known. The actual signal values assigned to the input signals are put in a test bench, the test bench is analyzed and the model is simulated to obtain the output signal values. In this case, the output signal D_OUT gets the same value as the input D_IN and so this is also assigned the symbolic value 'D2'. The stimulus/response pair generated is as shown below:

<table>
<thead>
<tr>
<th>D_OUT</th>
<th>CLR</th>
<th>LD</th>
<th>D_IN</th>
<th>CLK</th>
</tr>
</thead>
<tbody>
<tr>
<td>D2</td>
<td>0</td>
<td>1</td>
<td>D2</td>
<td>F</td>
</tr>
</tbody>
</table>

This stimulus/response test clearly tests an operation of the process REG. It implies that when CLR is '0' and LD is '1', then an event 'F' (fall) on signal CLK will result in the value 'D2' at input D_IN getting assigned to output D_OUT. This completes the propagation of event on signal CLK. The symbolic values are written to the test file in the order in which the signals are stored in a link list in the PMG database of the Modeler's Assistant. This order is called the "portorder". The portorder for the process REG is D_OUT - CLR - LD - D_IN - CLK.

The PTG algorithm similarly picks up the other sensitive signal CLR, creates an event on it and finds the response at an observable node. The test set for the stimulus on CLR is also inserted into the test data file and the sensitization of the sensitive input signals is completed. The two time frame test sets are generated from the first two single time-frame test sets by assigning constant stable values to all the signals in the second time frame, the first time frame remaining the same. All the test sets, both the single time frame and the double time frame test sets, are preceded by the number of time frames needed for the set.

The test data file for process REG is shown in Figure 8. Similarly, the process test data files for the other processes in the example circuit of Figure 2 are also generated by PTG algorithm.

<table>
<thead>
<tr>
<th>portorder: D_OUT CLR LD D_IN CLK</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>D2 0 1 D2 F</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>D2 0 1 D2 1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>D2 0 1 D2 F</td>
</tr>
<tr>
<td>D2 0 1 D2 0</td>
</tr>
</tbody>
</table>

Figure 8. Process Test Data File for process REG

4. Hierarchical Behavioral Test Generation

Once the process test sets for all the individual modules have been generated by the PTG software and stored in the design library, another program called Hierarchical Behavioral Test Generator (HBTG), is invoked to construct a test sequence for the whole entity. The final tests are generated hierarchically from these test sets with the interconnection information between the processes coming from the PMG data base of the Modeler's Assistant.

The HBTG algorithm consists of two parts. The first part is the construction of sensitive paths through the PMG of the circuit, and the second part is the test generation process.

Definition: A Sensitive Path is a directed path that begins at a sensitive primary input port and ends at a primary output port consisting of as many sensitive ports along the path as possible [3].

The number of sensitive paths constructed are equal to the number of primary sensitive input ports of the model. In our example circuit of Figure 2, four sensitive paths are constructed as shown below:

- path 0: L->M->SEL->OP->D_IN->D_OUT
- path 1: CLK->D_OUT
- path 2: B->O->CLR->D_OUT
- path 3: A->O->CLR->D_OUT

After the sensitive path construction, the test generation process starts which generates tests for the entire model using the process test sets provided by the PTG software for each process in the model. Each sensitive path is selected one by one and is sensitized. A path is sensitized by activating the first port along the path. This activation is then propagated to the last port of the path which is a primary output of the circuit, thereby activating other intermediate ports along the path. After the forward propagation of the event, the justification process of the HBTG algorithm starts which justifies the assigning of any signal values to the internal signals of the PMG during propagation, towards its primary inputs. Finally, the implication process calculates any new or unknown values at an output port of a process when the values on its inputs are specified [3].
The HBTG program selects appropriate test sets from the process test data files stored in the design library and instantiates them during the test generation process [3]. It selects different test sets for different operations and generates the final test sequence as described in the example in the following section.

### 4.1 HBTG Example

We again consider the example circuit shown in Figure 2 and discuss how the final test sequence is constructed for this circuit by HBTG.

In the test sequence generated by HBTG, the signal values are along the horizontal axis while the vertical axis shows the time-frames in the order in which they are inserted during test generation. All the signal values are "unknown" at the start of the test generation process.

<table>
<thead>
<tr>
<th>Frame</th>
<th>M</th>
<th>L</th>
<th>D_OUT</th>
<th>LD</th>
<th>CLK</th>
<th>O</th>
<th>B</th>
<th>A</th>
<th>OP</th>
<th>IN2</th>
<th>IN1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

The test generation starts with the sensitization of the first sensitive path, i.e., the sensitive path 0. An event is created on the first port of the path, i.e., port L. From the process test data file for module INV, a test with an event 'F' (fall) on L is selected, which is:

\[ \text{M} \quad \text{L} \quad \text{R} \quad \text{F} \]

The test selected assigns an event 'R' (rise) to signal M after a delta delay\(^4\). This event now has to be propagated to the primary output port on the path 0, i.e., port D_OUT. In order to do so, a test with the same event on the next port SEL on the path has to be selected. Thus, the following test set is selected from the process test data file of the process MUX:

<table>
<thead>
<tr>
<th>OP</th>
<th>IN2</th>
<th>IN1</th>
<th>SEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>D2</td>
<td>D3</td>
<td>D2</td>
<td>R</td>
</tr>
</tbody>
</table>

The above test assigns a signal value 'D2' to the port OP after a delta delay. This is further propagated forward by picking the following single time-frame test set from the process test data file of the process REG:

<table>
<thead>
<tr>
<th>D_OUT</th>
<th>CLR</th>
<th>LD</th>
<th>D_IN</th>
<th>CLK</th>
</tr>
</thead>
<tbody>
<tr>
<td>D2</td>
<td>0</td>
<td>1</td>
<td>D2</td>
<td>F</td>
</tr>
</tbody>
</table>

\(^4\)In a time frame, it is assumed that signal values are assigned to the output ports a delta delay after the event on an input port has occurred.

The primary output port D_OUT is thus assigned a value 'D2' and the forward propagation is completed. The justification process now begins to justify the signal value '0' assigned to the internal signal CLR during forward propagation. The justification process selects the following two time-frame test set from the test data file of the process AND:

<table>
<thead>
<tr>
<th>O</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>1</td>
<td>F</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

A two time-frame test is needed to guarantee that a constant value is assigned to signal CLR (output port O of process AND) in frame 1 in the table shown below. Hence, the complete test sequence for activation of path 0 is:

<table>
<thead>
<tr>
<th>Frame</th>
<th>M</th>
<th>L</th>
<th>D_OUT</th>
<th>LD</th>
<th>CLK</th>
<th>O</th>
<th>B</th>
<th>A</th>
<th>OP</th>
<th>IN2</th>
<th>IN1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>R</td>
<td>F</td>
<td>D2</td>
<td>1</td>
<td>F</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>D2</td>
<td>D3</td>
<td>D2</td>
</tr>
</tbody>
</table>

As can be seen, the justification process inserts a time-frame (frame 2) between the frames 0 and 1. The frame numbers indicate the order of selection. A delay of a few ns is assumed between the time-frames 2 and 1. The event 'F' on A in time-frame 2 stabilizes to '0' in time frame 1 a few ns before the event 'F' on I, takes place. The paths 1, 2 and 3 are activated in the same way. The final test sequence generated by HBTG for the example circuit is as shown below:

**Table 2. Final Test Sequence for the Example Circuit**

<table>
<thead>
<tr>
<th>Frame</th>
<th>M</th>
<th>L</th>
<th>D_OUT</th>
<th>LD</th>
<th>CLK</th>
<th>O</th>
<th>B</th>
<th>A</th>
<th>OP</th>
<th>D2</th>
<th>IN1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>2</td>
<td>R</td>
<td>F</td>
<td>D2</td>
<td>1</td>
<td>F</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>D2</td>
<td>D3</td>
<td>D2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>D2</td>
<td>1</td>
<td>F</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>D2</td>
<td>D3</td>
<td>D3</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>D2</td>
<td>0</td>
<td>1</td>
<td>R</td>
<td>R</td>
<td>1</td>
<td>D2</td>
<td>D3</td>
<td>D2</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>D2</td>
<td>0</td>
<td>1</td>
<td>R</td>
<td>R</td>
<td>1</td>
<td>D2</td>
<td>D3</td>
<td>D3</td>
</tr>
</tbody>
</table>

There are six time-frames in the sequence. Frames 2 and 1 are used for the activation of path 0, frame 3 is used for the activation of path 1, frame 4 is used for the activation of path 2 and frame 5 is responsible for the activation of path 3. During simulation the test sequence is applied in lexical order [3].
5. Test Bench Generation

The final test sequence produced by HBTG has to be converted into a test bench before it can be used to perform simulation. In order to do this, a program called Test Bench Generator (TBG) was developed that automates the generation of a test bench from the HBTG test sequence.

The TBG parses the HBTG test sequence and the VHDL source file and also extracts information from the PMG database of the Modeler's Assistant. The user is prompted to assign specific values to the symbolic signal values D_i's from the set of values assigned earlier during process test generation, and a test bench is automatically generated for the model. In the test bench generation we have assumed each time-frame to be equal to 4 ns duration. It is also assumed that there is a delay of 2 ns between any two consecutive time-frames.

If there is an event 'R' on a signal in a time-frame, TBG assigns a value '0' to the signal in the 1st ns of its 4 ns time period and it assigns a value '1' to it in the 4th ns of the same time interval. Similarly, if there is a event 'F' on a signal, it is assigned a value '1' in the 1st ns and a value '0' in the 4th ns of the 4 ns period to represent a transition from high to low and thus to denote an event fall on the signal. If the test sequence shows a constant value for a signal, TBG assigns the same value to it in both the 1st ns and the 4th ns of the time-frame period. The test-bench file generated by TBG is then used to simulate the model using the Synopsys VHDL System Simulator [15].

6. Test Generation System Evaluation

We have used the coverage property of the Synopsys VHDL System Simulator [15] to determine the effectiveness of an HBTG test sequence. The coverage property is a feature that gives the number of times each VHDL statement is executed when a model is simulated.

The test bench given by TBG for the example in this paper is used to simulate the model. A portion of the coverage results obtained are shown in Figure 9. As can be seen, all the statements are executed several times when the model is simulated using the test bench derived from the test sequence generated by HBTG.

<table>
<thead>
<tr>
<th>Line CountText</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
</tr>
</tbody>
</table>

Figure 9. Coverage Results for the Example Circuit Model

In [3] a representative set of models were tested and simulated. The coverage results obtained show that every statement of the VHDL behavioral models was executed at least once and most of the statements were executed several times.
7. Conclusions and Future Work

A systematic test bench generation system has been presented in this paper that can generate a test bench for VHDL behavioral models. The Process Model Graph, a graphical representation of a VHDL behavioral model, forms the base of the entire system. Test sets for the individual processes are generated by the PTG algorithm and are stored in the design library. The HBTG algorithm uses the graphical information stored in the PMG database and the process test sets to hierarchically construct a test sequence for the whole entity. The test sequence generated is finally converted by the TBG program into a test bench to be used for simulation.

The results obtained from the statement coverage measure during simulation of the models with the test benches given by the proposed system confirm that the test bench generation system exercises the functionality of the models fully.

Our future work will attempt hierarchical test generation for larger circuits at a level higher than the process level. The Modeler's Assistant uses a term "supernode" to define a PMG collapsed into a node [9]. A test sequence generated by the system for a PMG at one level of hierarchy will become the primitive test sets for larger circuits at a higher level consisting of an interconnection of various nodes and supernodes. Using techniques similar to those in this paper, the test bench generation system will be able to generate a test bench for a much larger VLSI system which is broken down into a hierarchy of nodes and supernodes.

References


