Process Level Test Generation for VHDL Behavioral Models

by

Shekhar Kapoor

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

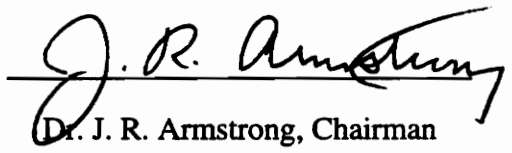
in partial fulfillment of the requirements for the degree of

Master of Science

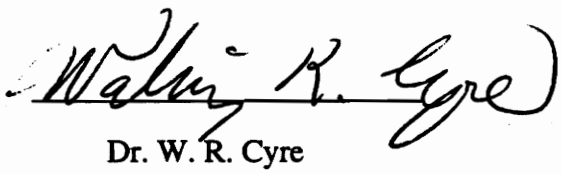
in

Electrical Engineering

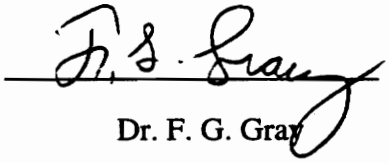
APPROVED:



Dr. J. R. Armstrong, Chairman



Dr. W. R. Cyre



Dr. F. G. Gray

March 1994

Blacksburg, Virginia

c.2

Process Level Test Generation for VHDL Behavioral Models

by

Shekhar Kapoor

Dr. James R. Armstrong, Chairman

Bradley Department of Electrical Engineering

Abstract

This thesis describes the development of the Process Test Generation (PTG) software for the testing of single-process VHDL behavioral models. The PTG software, along with Hierarchical Behavioral Test Generator (HBTG) and Modeler's Assistant, forms a part of the Automatic Test Generation System being developed at Virginia Tech. The PTG software transforms the VHDL description of a circuit, given by Modeler's Assistant, into a Control Flow Graph (CFG) that describes the control and data flow information in the behavioral model. The process test generation algorithm, called the PTG algorithm, uses the CFG to generate stimulus/response test sets that test all the functions of the VHDL model. The algorithm creates events on signals, propagates these events and uses simulation to obtain responses. Various features present in the software like the generation of the Control Flow Graph, the PTG algorithm, and the construction of paths through the CFG to propagate and justify events, are discussed. The test sets generated by PTG can be used for the hierarchical test generation by HBTG, which was developed earlier. Another program, called Test Bench Generator (TBG), is presented in this thesis. It is used to convert the test sequence generated by HBTG into a VHDL Test Bench that can be used for simulation.

Acknowledgments

I would like to express my sincere thanks to my advisor Dr. James R. Armstrong for providing constant support and guidance throughout my thesis research. It has been an honor and a pleasure to work in a challenging environment under him. I would also like to thank Dr. W. R. Cyre and Dr. F. G. Gray for serving as members of my committee.

I am indebted to many people for help with innumerable ideas that contributed to this thesis. My sincerest thanks are reserved for Dhawal Tyagi for his valuable and timely suggestions during the software development. I am grateful to Sanat Rao for his wholehearted assistance in research and other matters. Also, much gratitude to S. Shankar and David Dailey for their help and useful comments. I appreciate the motivating company of Vikram (Pizza) Shrivastava, Anand (Whasap!!) Joshi, Chang Cho, John Wicks, Sathya, and many others, who made research a very pleasant endeavor.

I would like to dedicate this work to my parents who have always encouraged me at all the times. Their unreserved love and support are the main reasons for all my achievements. I also owe it to my Swati for always instilling confidence within me. Her unbounded love and patience have meant more to me than anything else, and I love her for all that she has done for me. I specially wish to thank my sister, Sheefa, and my brother-in-law, Arun, for always giving me the right advice and help.

Finally, thanks are due to Anupam Malhotra, Sandeep Suri, Sanjiv Tandon, Sanjay Seth, Nitin Kapila, Veeresh, Varinder, Ashish Kaul, Navneet Singh, Sanjay Nagpal, Sameer, Sanju and Gagandeep, to name a few, who have provided me with some of the best moments of my life. I convey my deepest appreciation to all my friends and dear ones.

Table of Contents

List of Illustrations

Chapter 1. Introduction

1.1 Motivation

Recent advances in semiconductor technology have led to a tremendous increase in the level of integration achieved by VLSI circuit technology. The number of transistors on a chip has increased from a few hundred to over a million, e.g., the Intel Pentium microprocessor consists of 3.1 million transistors [1]. When designing such highly integrated circuits, the significance of the design verification phase cannot be over-emphasized. With the increasing complexity of integrated circuits, the test pattern generation for design verification has become extremely expensive and time-consuming. The traditional test pattern generation techniques have been rendered inadequate because of the increased complexity and the lack of information about the gate-level description. A constant effort is being made by the research community for developing new methods

of test pattern generation which are able to tackle the design verification problems arising from the technology evolution.

Design verification by simulation of the circuits using powerful computer tools is becoming widely popular. The important factors in simulation are the ability to accurately describe the functionality of a circuit and the ability to test the different operations by generating appropriate test stimuli. In order to tackle the weaknesses of the traditional test pattern generators, designers generally try to generate tests using a higher level abstraction, preferably a high-level functional or behavioral description of the circuit.

Hardware description languages (HDLs) are being increasingly used for simulation and design verification. HDLs can model concurrency in the execution of the logic circuits and can also model the timing relationship between the logic blocks. HDLs provide a convenient tool for the functional or behavioral specification and simulation of digital circuits, reducing the quantity and detail of the information which the designer must manage.

The VHSIC Hardware Description Language (VHDL) is becoming extremely popular in electronic industry as a modeling language. It is being widely used for high level test generation and design verification, and it is an IEEE standardized language (IEEE 1076 standard) [2]. It has a rich set of constructs for circuit modeling, and circuits can be described in either top-down or bottom-up fashion through varying levels of abstraction. After the VHDL behavioral model of a circuit is developed, test data can be generated

for simulation of the model. By observing the simulation output, the functionality of the model can be verified.

The general practice in the design verification process is to simulate the circuit with a very large amount of test data given by a designer or a modeler. 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. As a result, basic functions are left untested. A number of high-level test generation techniques have been proposed [4-11] for VHDL models.

A systematic test bench generation system for design verification is being developed at Virginia Tech that automatically generates the tests for a VHDL behavioral model. A modular approach to test generation is used that partitions a circuit into smaller, primitive modules with the help of a ***Process Model Graph*** (PMG). Tests for the individual modules are computed first and these are then used to hierarchically generate the tests for the entire circuit. The test sequence generated is converted into a test bench which can be used for the simulation and the design verification of the model.

The automatic test generation system was incomplete in the sense that only one major tool, called ***Hierarchical Behavioral Test Generator*** (HBTG) [3], was developed. HBTG accepts the PMG of the VHDL behavioral model as input and hierarchically generates tests for the circuit using the precomputed test sets for each module. Initially, the precomputed tests were manually written which is a very time-consuming task. This thesis presents a system that can independently generate the tests for the individual

modules. These tests can be stored in a design library which can later be used for the hierarchical test generation of a circuit.

Previously, a system was proposed for the evaluation of the test sequence that was generated by the test generation system [3]. This technique, called the *Statement Coverage*, reports the number of times each statement of the VHDL model is executed when the circuit is simulated with the help of the test data generated by the system. The Process Test Generation algorithm proposed in this thesis, ensures that every operation of the VHDL model is executed at least once and thus improves the effectiveness of the entire test generation system.

Finally, the test sequence generated by the earlier system could not be easily used for simulation as it needed to be converted into a test bench first which was also done manually. A program was developed that automatically converts the test sequence generated by HBTG into a test bench to be used for simulation. The program systematically maps the symbolic notation used earlier to actual signal values and speeds up the test bench generation process.

1.2 List of Contributions

The primary objective of this thesis is to develop tools that can be interfaced to the HBTG [3] and the Modeler's Assistant [29], and provide a complete and an efficient test bench generation system that could test different functions of a VHDL model. Two tools were developed, Process Test Generator (PTG) and Test Bench Generator (TBG), that

provide a time-saving and a convenient way of test bench generation. Both the programs are running on a SUN SPARCstation2 network. The tools are versatile in the sense that process test sets for primitive models of reasonable size can be developed. The other contributions made by the thesis include the following:

- Automates the entire test generation process. There is an obvious increase in speed over the manual generation of precomputed test sets and the manual conversion of the HBTG test sequence into a test bench.

- Stimulus/response test sets are generated that test all the functions of the primitive models and thus help generating a test sequence that can test all operations of the a circuit described as an inter-connection of these primitives.

- The Control Flow Graphs are generated for the VHDL primitive models that give the complete data and control flow information for the behavioral models. Various data structures are defined that store all relevant information which can be further used for process test generation purposes.

- Shortest paths (justification and forward propagation paths) are used to propagate signal values in an efficient manner.

- Hash Tables are used that speed up the controllable node and the observable node search processes, and thus, speed up the entire test generation system.

- The test sequence generated by HBTG, consisting of a symbolic notation, is converted into a test bench by TBG with the help of user-interaction.

- User assists to the test generation process are used, wherever necessary.

- The PTG software helps in building up a Primitive Test Library, consisting of the Process Test Data files for various primitives. The storing of the process test results in a library considerably increases the speed of the automatic test bench generation system.

The PTG algorithm and the TBG algorithm consist of approximately 7500 lines of C code. The files have been appropriately documented and a programmer's manual has been developed to facilitate the test generation. A user's manual has also been developed to explain the VTIP/DLS and SPI software (a commercial tool developed by COMPASS Design Automation, Inc.) used in the process test generation.

1.3 Contents

Chapter 2, "Background and Literature Review", discusses modeling with VHDL along with gate-level and functional/behavioral test generation techniques.

Chapter 3, "System Overview", overviews and discusses the development of the automatic test bench generation system at Virginia Tech.

Chapter 4, "Graph Transformation: The Base for Process Test Generation", discusses how the VHDL behavioral description of a primitive model is transformed into a Control Flow Graph (CFG) and a complete Directed Model Graph (DMG). A Hash Table construction for a single-process VHDL behavioral model is also described in this chapter.

Chapter 5, "The Process Test Generation", describes the PTG algorithm in detail. The extraction of information from the PMG database of Modeler's Assistant, selection of controllable and observable nodes, construction of shortest propagation and justification paths and the entire process test generation algorithm, are discussed.

Chapter 6, "The Test Bench Generation", discusses how a final test sequence generated by HBTG for a circuit is converted into a test bench that can be used for simulation.

Chapter 7, "Results", presents a brief description of the VHDL behavioral models for which the process test results are given in Appendix C.

Chapter 8, "Analysis and Suggestions", considers limitations of the process test generation software, proposes possible enhancements to the system, and suggests areas for future work.

Chapter 9, "Conclusion", draws conclusions regarding the algorithms and their implementations.

Appendix A, "Programmer's Manual for PTG, TBG and their Environment," describes the various data structures used by PTG and TBG along with an explanation of the compilation procedures for these programs. It also provides detailed instructions concerning the use of the programs.

Appendix B, "The VTIP/DLS and SPI Software: PTG Related User's Manual", introduces the user to the VTIP software and describes how it is used in process test generation.

Appendix C, "Circuit Models, Control Flow Graphs and Process Test Results", shows the source listings for the VHDL behavioral models used as examples, gives the Control Flow Graphs generated for these models, and shows the process test sets produced for these models by PTG algorithm. The Test Benches created by TBG for two multi-process VHDL behavioral models are also shown.

Chapter 2. Background and Literature Review

2.1 Test Generation for Digital Systems

Testing of a system is defined as an experiment in which the system is exercised and its response is analyzed to determine whether it is behaving correctly or not [33]. *Test generation* (TG) is defined as the process of determining the test stimuli necessary to test a digital system [33]. Test generation for design verification involves a large effort that, unfortunately is still a manual activity. *Automatic Test Generation* (ATG) refers to the test generation algorithms that, given a model of a circuit, can automatically generate tests for it [33]. Test generation is generally classified into the following main categories [33]:

1. *Fault-Oriented Test Generation* in which tests are generated to detect (and possibly locate) specific faults in the circuit.

2. *Function-Oriented Test Generation* in which tests are generated which, when applied to the model, show whether the circuit performs a specific operation or not.

2.1.1 Gate Level Test Generation

The classical algorithms for test generation are presented in [12, 14, 15]. Original automatic test generation work was done using gate level structural descriptions where the circuit was explicitly given as a set of primitive interconnected logic gates, e.g., AND, OR, XOR, etc. A simple fault model based on "*stuck-at*" faults was used. The stuck-at fault model characterizes faults as producing permanent logic values on interconnection lines, e.g., a particular line is "*stuck-at-1*" (s-a-1) if the line delivers a constant 1 logic value to connected lines or components. Probably the most important development in automatic test generation was the D-algorithm proposed by Roth [12].

The D-algorithm is based on D-calculus, in which D represents a good value of 1 and a faulty value of 0, $\overline{D}$ represents a good value of 0 and a faulty value of 1, and gate functions are redefined in terms of D or $\overline{D}$ (in addition to only 0 and 1). D-calculus provides for defining the effects of a stuck-at fault, and is used by many subsequent test methods. The D-algorithm uses the structure of a circuit to propagate faulty values through the circuit, and to justify values needed on internal lines. Justification may require choices, which in turn may lead to conflicts. The D-algorithm uses backtracking when conflicts are discovered, as with all subsequent methods.

The 9-V algorithm was proposed by Cha, Donath and Ozguner in 1978 [13]. The main difference between the 9-V algorithm and the D-algorithm was the use of nine values in 9-V algorithm which reduces the amount of search done for multiple sensitization.

PODEM (Path-Oriented Decision Making) was proposed by Goel [14] in 1981. The algorithm is characterized by a direct search process. In this method the decisions consist only of primary input assignments. The algorithm uses the symbols D and $\overline{D}$ for fault sensitization. First, values are assigned to the Primary inputs (PI) to set the output of the gate under test to D or $\overline{D}$. It is then verified that these assigned values are sufficient to sensitize the fault. If the fault is not yet sensitized, additional values are assigned to other primary inputs. Once the fault has been sensitized with D or $\overline{D}$, the value is propagated towards a primary output by assigning values to the primary inputs. The direct search method used by PODEM reduces the backtracking steps, as only the primary input values have to be changed.

FAN (Fanout-Oriented Test Generation) was proposed by Fujiwara and Shimono in 1983 [16]. It also uses the stuck-at fault model and the symbols D and $\overline{D}$ are used for fault sensitization. The FAN algorithm improves upon PODEM by stopping the backtracking at internal lines rather than at the primary inputs. Thus the backtracking to justify the values can be reduced and test generation time is reduced. Further, FAN uses multiple-backtrack procedure that attempts to simultaneously satisfy a set of objectives.

2.1.2 Functional or Behavioral Level Test Generation

Test generation using the gate level approaches discussed in the previous section, becomes very time consuming and expensive as the complexity of the circuits increase. The test generation problem at the gate level is known to be NP-complete [17]. Functional test generation represents the abstraction of the digital circuit test problem to a level amenable to the testing of VLSI devices. A functional description of a circuit has various advantages over a lower-level description [18]:

- It reduces the test time required for large digital circuits
- The circuits generated can be used for more than one system with the same functionality
- Functional test generation does not require a detailed internal structure information

A few of the functional-level test generation techniques are reviewed.

2.1.2.1 Register Level Test Generation

The register level is above the gate level. At this level, functional units such as registers, multiplexers, ALUs, ROMS and RAMs are considered basic primitives. The behavior of the circuit is usually represented by the Register Transfer Language (RTL), that represent the data flow between the various primitives at the Register Level.

Lin and Su [18] proposed the S-algorithm, a symbolic execution approach for functional testing based upon an RTL description. An RTL fault model based on a well defined RTL is described. The S-algorithm generates tests by injecting RT level faults (jump faults,

condition faults, data transfer faults, register decoding faults and operator decoding faults) and then simulating symbolically both the good and faulty models. The symbolic inequalities obtained are then solved; input values are selected such that the constraints are satisfied and the results for good and bad executions differ. This set of input values, if they are found, constitute a test. They report that the generated test sets give good functional level fault coverages.

Kawai et al. [19] proposed a heuristic ATPG system using the FDL Functional Description Language. FDL descriptions of circuits incorporate some detail of possible circuit implementations; each FDL statement corresponds to a primitive circuit element, but may be fairly complex. Using FDL circuit descriptions, a test generation algorithm, the P-algorithm, was developed to generate tests for combinational circuits or circuits with a scan path, where all internal registers form a shift register that can be loaded or read. The FDL circuit description is translated into a graph form; each node in the graph representing an FDL operator and the branches between nodes representing data or control lines between the nodes. An output node is picked up and a path is sensitized from the output to an input by propagating a "P" value. Test patterns are then generated which justify the sensitized path.

2.1.2.2 Test Generation with Binary Decision Diagrams

The functional behavior of a circuit can also be represented as a binary decision diagram [20]. A binary decision diagram is a graph model describing the logical operations of digital functions of a circuit. The traversal through the graph is determined by examining

the value at each node. At each node the left or right branch is followed depending on the value (0 or 1) of the corresponding input variable. An experiment consists of finding a complete path from the input to the output. The complete path from the input to the output constitutes a test. A complete test is generated when all the paths have been traversed. Various test generation schemes using binary decision diagrams have been proposed [20, 21, 22]. In all these approaches fault models are defined in two ways: stuck-at faults and the functional faults. However, the complexity of these techniques increases for bigger models and hence this method is no longer suitable for test generation.

2.1.2.3 Graph Methods

The graph methods model the circuit under test as directed graphs and use graph theory principles to simplify the test generation problem.

Robach and Saucier [23] proposed an approach in which each instruction of a microprocessor is represented by an *abstract execution graph*. In such a bipartite graph, each node represents a memory element or a micro-operation and each arc represents the data flow between the nodes. The instructions of a microprocessor are grouped into classes and subclasses. Functional testing can be done by either verifying correct execution of instructions from the smallest subclass to the largest subclass (start-small test) or verifying correct execution from the largest to the smallest one (start-big test).

Thatte and Abraham [24] modeled a microprocessor as a directed graph (S-graph) where nodes are registers and edges represent data flows. A fault model is developed which is

independent of the implementation details. A disadvantage of this technique is that the S-graph does not contain the control flow information.

These graph approaches are well suited to microprocessor testing but they lack the generality necessary for application to all types of digital systems.

2.1.2.4 Hierarchical Test Generation

In many cases, the circuit to be tested is composed of submodules that are not directly decomposable into logic gates or the gate-level descriptions are not available. As a result, *hierarchical* techniques based on high-level primitives have been proposed. These techniques exploit the hierarchy inherent in the circuit descriptions. The circuit is described as an interconnection of high level primitives. Various high-level approaches to test generation are discussed in this section.

Kunda, Narain, Abraham and Rathi [17] suggest a method to speed up the test generation process by making use of the high-level primitives. The circuit to be tested is modeled as a data-flow graph; the nodes of the graph are the high level primitives. Dependency-directed backtracking is used to reduce the number of backtrackings required.

Murray and Hayes [25] propose a hierarchical technique for test generation which does not assume any specific fault model. A modular approach to test generation is suggested that divides the circuit into smaller high-level functional modules. Stimulus/response test sets are first generated for the correctly functioning modules and are stored as primitive

tests in the design library. Two kinds of tests are used: the FTPs which are used to sensitize the faults in the module, and the PTPs which are used to perform signal propagation and justification. The test for the whole model is then hierarchically constructed using these precomputed tests. Experimental results show that this approach gives a great improvement over the traditional gate-level test generation.

Kruger [26] proposes a similar technique for hierarchical test generation. The test generator is part of a computer hardware design system, called MIMOLA. Predefined stimulus/response packages are defined for primitive models as in [25] and are stored in the design library. These test packages are then used for hierarchical test generation at the next level of hierarchy.

Sarfert, Markgraf, Schulz and Trischler [27] propose another scheme for hierarchical test generation for combinational circuits. High-level primitives supported include decoders, encoders, etc. The test generation for faults inside a high-level primitive is performed by expanding the primitive dynamically to its gate-level representation.

2.1.2.5 Test generation using Hardware Description Languages

Most recently, circuits have begun to be described at the chip level [4], with hardware description languages. Describing the model in a standardized hardware description language (HDL) saves time as well as provides compatibility between different approaches. Hardware description languages have proven to be a powerful tool for modeling, simulation, design verification and logic synthesis. The digital circuits of

varying sizes and complexities can be represented from transistor level to system level and from behavioral description to structural description by an HDL. HDLs in general are characterized as *procedural* and *non-procedural* [4]. In non-procedural HDLs, each statement in a given design is evaluated in parallel (i.e., concurrently). In the more abstract procedural HDLs, each statement is evaluated in sequence, with execution of a given statement occurring in the time period after the preceding statement is evaluated. In general, non-procedural HDLs are closer to a hardware implementation than procedural HDLs. Behavioral HDL models are excellent sources of information for test generation at the functional level. Several approaches to test generation using HDLs are discussed below.

Levendel and Menon [4] introduced an algorithm based on an extension of the D-algorithm for generating test vectors from the hardware description language model of digital circuits. The fault model consisted of *stuck-at-1* and *stuck-at-0* on the input and output lines, functional faults and control faults. The D-algorithm approach was used to generate D-equations and D-propagation cubes for each functional module and control constructs in the HDL description. However, this proves to be very complicated for relatively complex functional modules, and hence, test generation efficiency is reduced using this algorithm.

Gupta and Armstrong [8] proposed a functional modeling scheme based on the hardware description language GSP (General Simulation Program). The HDL description is viewed as a sequence of micro-operations and control structures. A fault model based on model perturbation is used. Two types of faults are defined - micro-operation faults and control

faults. Micro-operation faults perturb individual micro-operations, while control faults perturb the control points that switch between micro-operation sequences.

Barclay and Armstrong [9] have incorporated the above fault model into a test generation system based on the hardware description language VHDL. Four types of fault are defined based on the fault model discussed above: Stuck-Then/Stuck-Else faults on IF statements, Deadclause faults on each clause of the CASE statement, ASSCNTL fault on each assignment statement and MICROOP faults on each micro-operation in the model.

A Behavioral Test Generator [6] has also been developed at Virginia Tech that has further enhanced Barclay's system while using the same high-level fault model. It has resulted in an average speed improvement over the earlier system.

Cho and Armstrong [7] investigate the relationship between the VHDL semantics of event-driven simulation and the test-generation algorithm. Methods are presented to generate tests without being influenced by the VHDL semantics.

F. E. Norrod [10] proposed the E-algorithm in Feb. 1988. It generated tests from circuits described in VHDL. The VHDL description is transformed into a graphical representation. The representation is as a directed graph with the nodes representing micro-operations and arcs representing data flow. The fault model is based on the data and control logic paths in the circuits. The test generation process is in 3 steps: *fault sensitization*, *propagation* and *justification*. A special method called E-propagation is used to propagate the fault effect through control constructs [10].

The FunTestIC test generation algorithm (FUNctional TEST pattern generation for Integrated Circuits) proposed by Hummer, Veit and Topfer [5] derives functional tests for modules and systems described in VHDL. The circuit model is represented by a Control Flow Graph, a Data Flow Graph and a Sequence Graph. The Control Flow Graph gives a flow chart representation of the circuit model. The Data Flow Graph represents the possible data flows between the variables of the models. The Sequence Graph provides information about the sequence in which the data has to be passed in the Control Flow Graph. Input/Output paths are derived from the description and activated, and the circuit is exercised with specific data.

2.2 Modeling with VHDL

One of the most widely used hardware description languages is the VHSIC Hardware Description Language (VHDL) [30]. VHDL was standardized by the IEEE in 1987 as its 1076 standard [2]. Work with VHDL has successfully been done in the areas of behavioral modeling, structural modeling, abstract level modeling and logic synthesis. The growing popularity of VHDL is because of its powerful set of constructs for modeling.

2.2.1 Structural and Behavioral Level Modeling

The VHDL models are generally defined in two kinds of abstraction hierarchy as follows:

Structural domain: A domain in which the model is represented as an interconnection of lower-level primitives.

Behavioral domain: A domain in which the model is described by defining its input/output response.

The structural and behavioral descriptions of a D flip-flop circuit, as shown in Figure 1, are given in Figure 2 and Figure 3 respectively. The circuit consists of a D flip-flop (DFF) , two AND gates, an inverter and an OR gate.

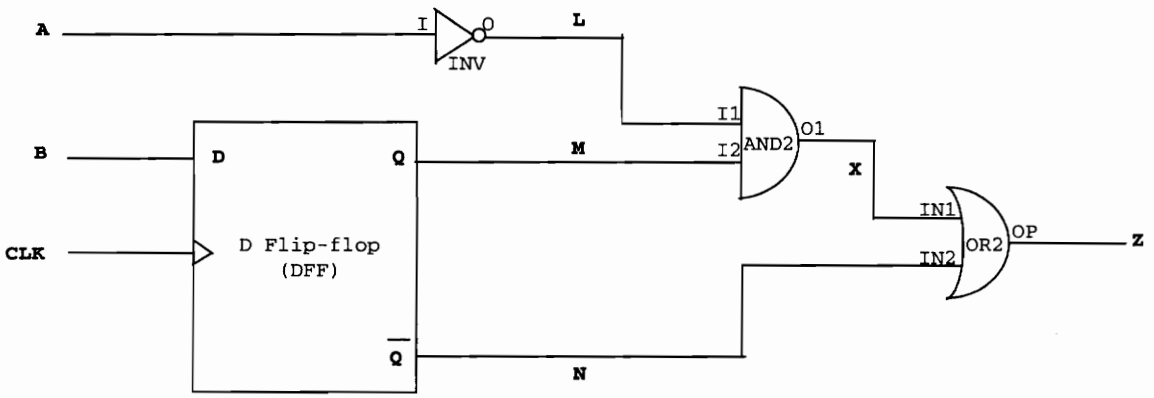


Figure 1. A D-FLIP-FLOP (DFF) Circuit

```
entity EXAMPLE1 is
  port (A, B, CLK: in BIT; Z: out BIT);
end EXAMPLE1;

architecture STRUCTURAL of EXAMPLE1 is

  component DFF_1
    port (D, CLK: in BIT; Q, Q_BAR: out BIT);
  end component;
  component INV_1
    port (I: in BIT; O: out BIT);
  end component;
  component AND2_1
    port (I1, I2: in BIT; O1: out BIT);
  end component;
  component OR2_1
    port (IN1, IN2: in BIT; OP: out BIT);
  end component;

  for all: DFF_1 use entity DFF(BEHAVIOR);
  for all: INV_1 use entity INV(BEHAVIOR);
  for all: AND2_1 use entity AND2(BEHAVIOR);
  for all: OR2_1 use entity OR2(BEHAVIOR);

  signal L, M, N, X: BIT;

  begin

    C1: DFF_1
      port map (B, CLK, M, N);
    C2: INV_1
      port map (A, L);
    C3: AND2_1
      port map (L, M, X);
    C4: OR2_1
      port map (X, N, Z);

  end STRUCTURAL;
```

Figure 2. VHDL Structural Model of the DFF Circuit

In the case of the structural description, the model is described as an interconnection of the primitives: D flip-flop, AND gates, OR gate and an inverter INV. On the other hand, the behavioral description describes the operation of the circuit in terms of its inputs and outputs without resorting to lower-level representations. The behavioral model can either be described by a single process or by a multi-process abstraction as shown in Figure 3. The advantage of using the multi-process behavioral descriptions is that it explicitly represents the division of functionality within the circuit model.

```
entity EXAMPLE1 is
  port (A, B, CLK: in BIT; Z: out BIT);
end EXAMPLE1;

architecture BEHAVIOR of EXAMPLE1 is

signal L, M, N, X: BIT;
begin

  INV: process (A)
  begin
     L <= not A;
  end process INV;

  OR2: process (X, N)
  begin
     Z <= X or N;
  end process OR2;

  AND2: process (L, M)
  begin
     X <= L and M;
  end process AND2;

  DFF: process (CLK)
  begin
     if CLK = '1' then
        M <= B;
```

```
    N <= not B;
  end if;
end process DFF;

end BEHAVIOR;
```

Figure 3. VHDL Behavioral Model of the DFF Circuit

2.2.2 The Process Model Graph

An architectural body is used to define the behavior in the VHDL description of a device [30]. The architecture body is a set of concurrently running processes that give the division of functionality within the VHDL model. The ***Process Model Graph*** gives a pictorial representation of a behavioral architectural body in the form of concurrently running VHDL process blocks. Figure 4 shows an example process model graph (PMG).

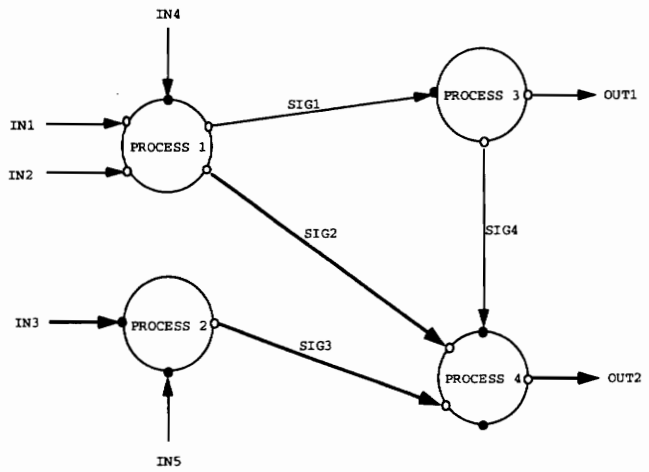


Figure 4. An Example of a PMG

A 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]. Thus, the PMG gives the partitioning of the VHDL model in the behavioral domain. Figure 5 shows the PMG for the circuit shown in Figure 1.

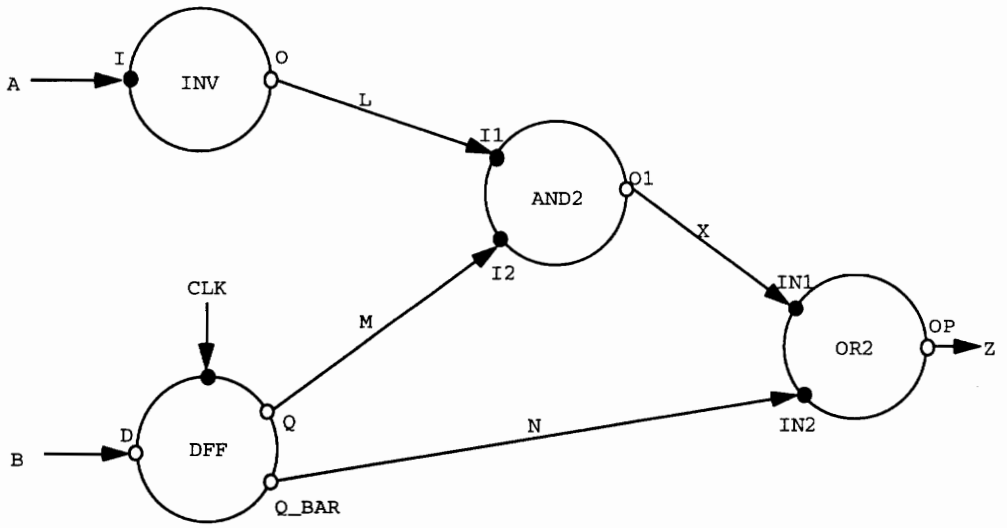


Figure 5. PMG for the DFF Circuit

[1] A *port* is defined as an input or an output of a process node of a PMG.
[2] A *sensitive port* is defined as a port such that a change in the value on it triggers the process and the statements within the process block are executed.

Chapter 3. System Overview

This chapter gives an overview of the CAD environment and Automatic Test Bench Generation System for VHDL Behavioral Models, being developed at Virginia Tech. The two main tools, Modeler's Assistant [29] and the Hierarchical Behavioral Test Generator (HBTG) [3], are discussed and two other tools: Process Test Generator (PTG) and Test Bench Generator (TBG) are introduced. PTG and TBG are the foci of this thesis and will be explained in detail in the following chapters. The system block diagram of the entire test generation system, that exists at present, is shown in Figure 6.

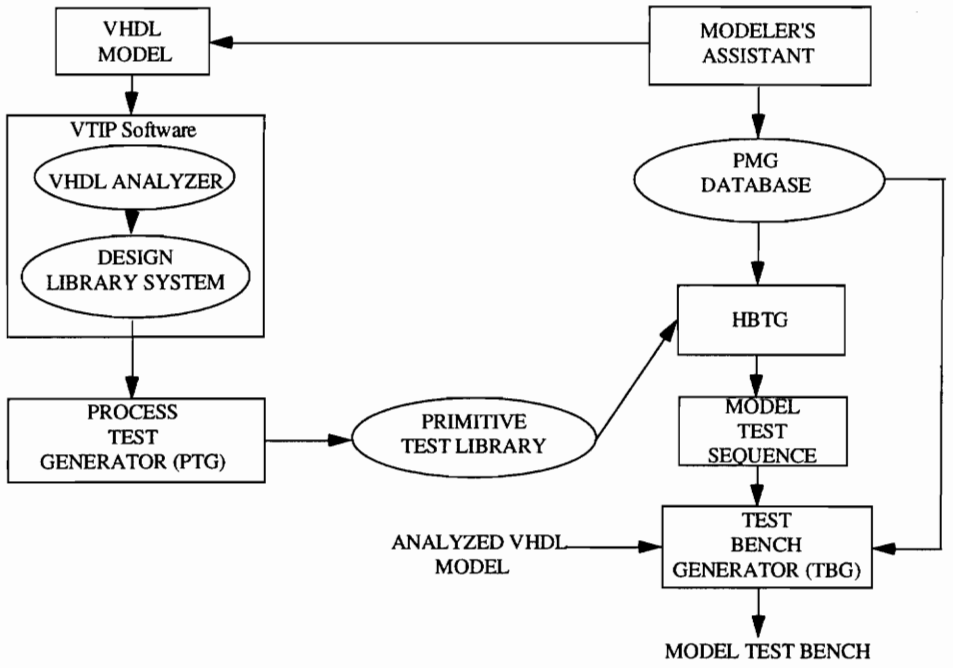


Figure 6. System block diagram of the Automatic Test Generation System

Modeler's Assistant, a CAD tool developed at Virginia Tech, interactively creates the graphical representation of a VHDL behavioral model, called a Process Model Graph (PMG), which is used as the base for test generation. It also generates a VHDL behavioral model for the circuit. A program called PTG, Process Test Generator, is used to generate the tests for individual processes of the PMG. The VHDL model of a process given by Modeler's Assistant, is analyzed with the help of VTIP VHDL analyzer and the design information is stored in the VTIP Design Library System (DLS) [31]. The PTG extracts the stored design information form DLS and generates a Control Flow Graph (CFG) describing the data and control flow information in the process. The

stimulus/response test sets are then generated by executing different paths in the graph. Another algorithm called HBTG, Hierarchical Behavioral Test Generator, accepts the PMG information 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 TBG, Test Bench Generator program, and it is then used for simulation of the model. Experimental results show that the test generation system gives test benches that exercise the models thoroughly.

The previous work done on the test generation system did not include the Process Test Generator (PTG) and the Test Bench Generator (TBG) tools. Both the tasks of developing a process test data file and the conversion of the final test sequence generated by HBTG into a test bench for simulation purposes, were done manually. The two previously existing systems, the Modeler's Assistant and the Hierarchical Behavioral Test Generator, are discussed in the following sections.

3.1 The Modeler's Assistant

The Modeler's Assistant [29] is a graphical CAD tool for the development of VHDL behavioral models. The graphical representation used is the ***Process Model Graph*** (PMG), as described in the previous chapter. It is generated interactively by the user. The VHDL code specifying the functionality of each process in the PMG 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. It thus provides a mixed textual/graphical design capture to reduce the effort required in the development of a VHDL behavioral model.

Figure 8 shows an example Process Model Graph created interactively by Modeler's Assistant. As stated in the last chapter, the larger circles represent VHDL processes within the architecture body of the entity and the smaller circles along the periphery represent the ports of the process to which they belong. A sensitive port is represented by a shaded port. An unshaded small square inside a process indicates a variable, while a shaded small square represents a constant. One or more processes together form a unit, called a PMG. Figure 7 shows the VHDL "shell" program produced by Modeler's Assistant for the PMG shown in Figure 8. The Modeler's Assistant assumes that there exist two predefined packages, VHDLCAD.vhd and USER_TYPES.vhd, consisting of various functions and type declarations.

```
use WORK.VHDLCAD.all, WORK.USER_TYPES.all;
-- ************************************************************
entity example_pmg is
  port (D_IN1: inout BIT_VECTOR(0 to 7);
    DATA_IN1: in BIT_VECTOR(0 to 7);
    I2: in BIT;
    I1: in BIT;
    OUT1: out BIT_VECTOR(0 to 7);
    B: in BIT;
    A: in BIT);
end example_pmg;
-- ************************************************************

architecture BEHAVIORAL of example_pmg is
  signal SIG2: BIT_VECTOR(0 to 7);
  signal SIG4: BIT;
  signal SIG1: BIT_VECTOR(0 to 7);
  signal SIG3: BIT;
begin
```

```
-- ------------------------------------------------------
-- Process Name: PROC_5
-- ------------------------------------------------------
PROC_5_4: process (SIG4)
   variable VAR2: BIT;
   constant CON2: BIT;
begin
        <User Defined Function>
end process PROC_5_4;

-- ------------------------------------------------------
-- Process Name: PROC_4
-- ------------------------------------------------------
PROC_4_11: process (SIG4)
begin
        <User Defined Function>
end process PROC_4_11;

-- ------------------------------------------------------
-- Process Name: PROC_3
-- ------------------------------------------------------
PROC_3_16: process (I2,I1)
   variable VAR1: BIT;
   constant CON1: BIT;
begin
        <User Defined Function>
end process PROC_3_16;

-- ------------------------------------------------------
-- Process Name: PROC_2
-- ------------------------------------------------------
PROC_2_23: process (SIG2,SIG1,SIG3)
begin
        <User Defined Function>
end process PROC_2_23;

-- ------------------------------------------------------
-- Process Name: PROC_1
-- ------------------------------------------------------
PROC_1_29: process (B,A)
begin
        <User Defined Function>
 end process PROC_1_29;
end BEHAVIORAL;
```

Figure 7. VHDL "shell" produced by Modeler's Assistant for the PMG

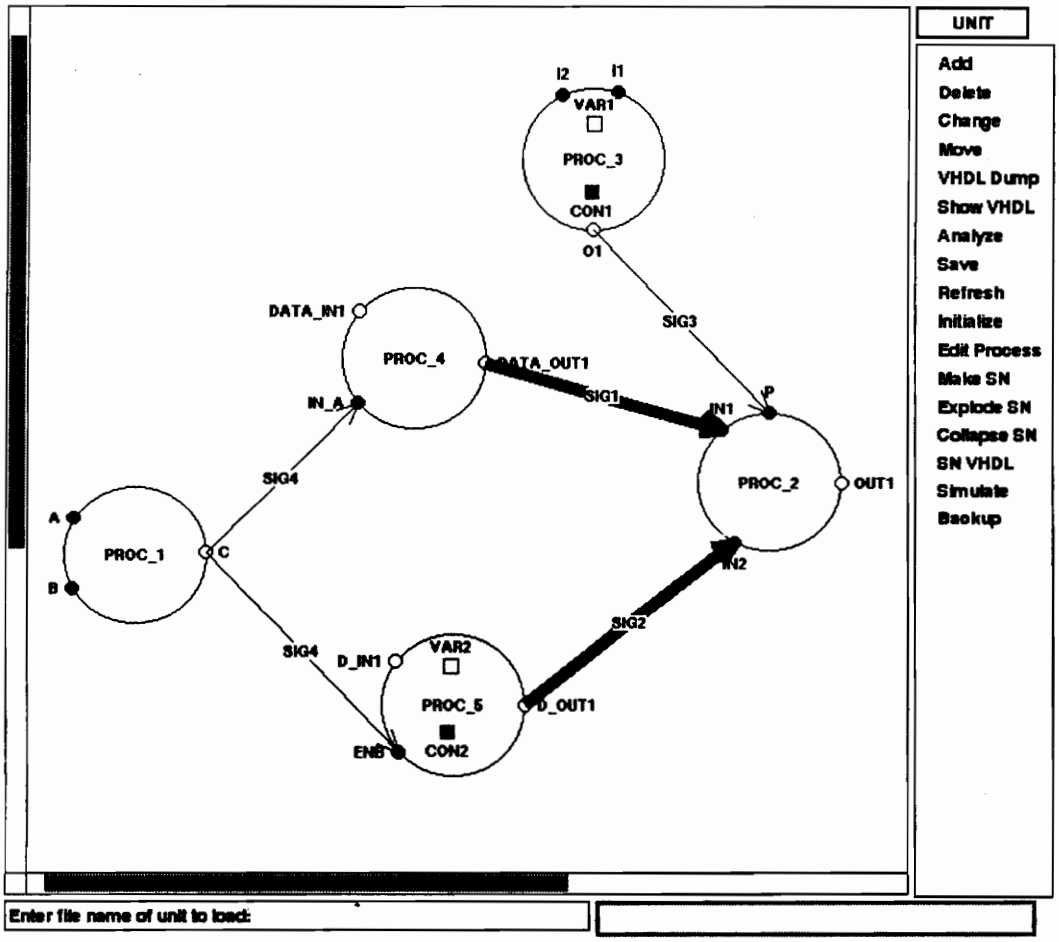


Figure 8. A PMG created by the Modeler's Assistant

Figure 9 shows the system block diagram of the Modeler's Assistant. The important components of the system are two databases containing the information required to generate all the graphics as well as the information required to produce the corresponding VHDL code. The process model graph database contains the details of the geometry of the process model graph in terms of the co-ordinates of the various graphical constructs like the processes, process ports, signals, variables and constants. It also contains details like the types of the signals, constants, variables and generics, the mode of the signals, information on whether the signal is in the sensitivity list of the process or not, and also the names of the elements of the process model graph. The PMG editor permits the user to manipulate the information in this database. The user is also provided with the ability to change the geometry of the PMG or to modify its characteristics, e.g., data types of signals or variables. The process functionality database contains the text of the functionality of all the processes currently in the process model graph. The HBTG algorithm and the PTG algorithm use these databases to extract useful information required for test generation.

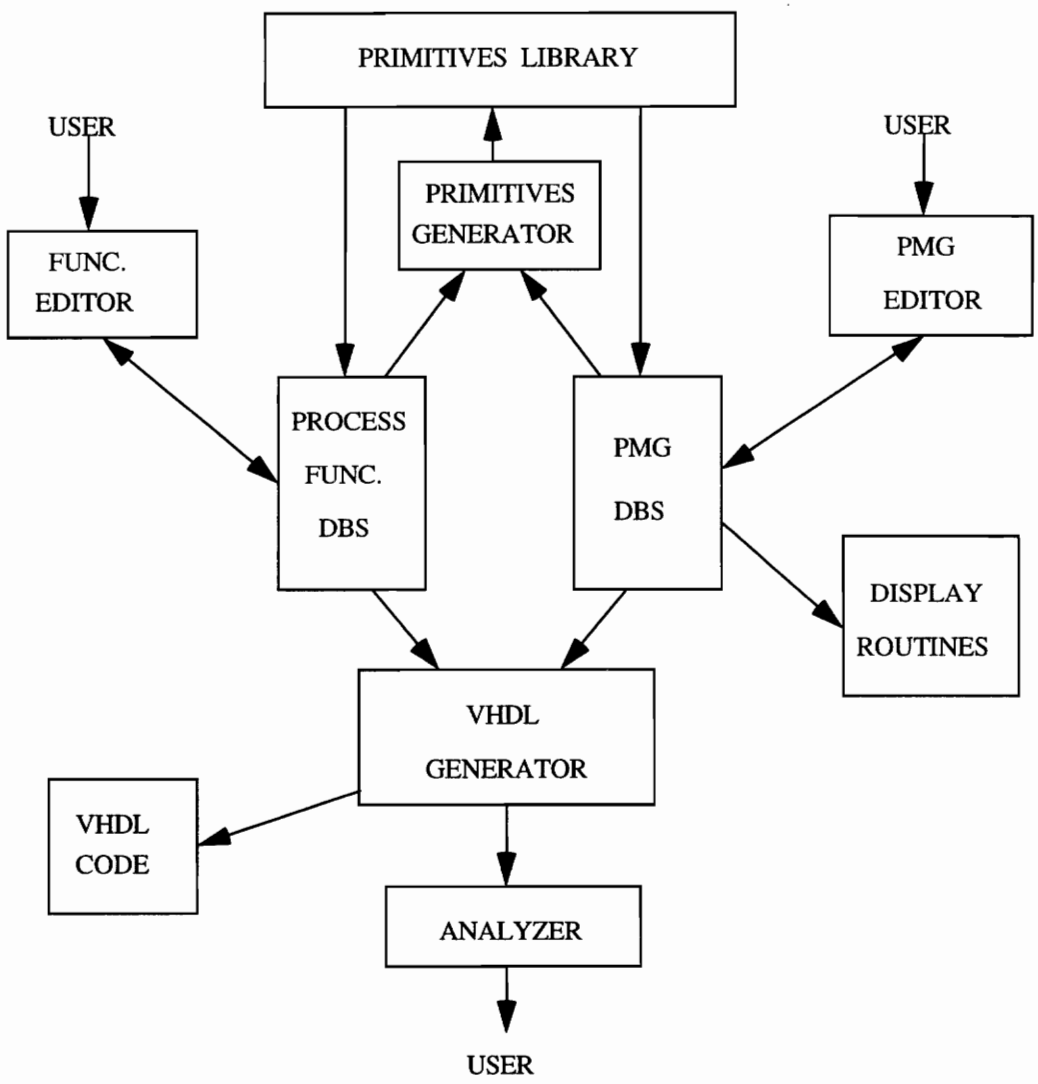


Figure 9. System block diagram of the Modeler's Assistant

3.2 Hierarchical Behavioral Test Generation

The Hierarchical Behavioral Test Generator (HBTG) [3] is a program developed at Virginia Tech for the automatic test generation for VHDL behavioral models. The algorithm performs functional testing and does not assume any specific fault model. The main task of the algorithm is to test as many functions of the model as possible. The PMG of the VHDL model to be tested is created using the Modeler's Assistant described in the previous section. For each process in the PMG, tests are manually computed and stored in the design library. These tests provide sufficient information about the functionality of the processes in the PMG. Using these tests, the HBTG hierarchically constructs a test sequence for the entire model.

3.2.1 Precomputed Test Data File Format

We have stated that the 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 were written manually and were stored in the design library. Figure 10 shows the PMG for an 8-bit latch circuit. The PMG has three processes: LTCH, ENB and OUTPUT. The VHDL source for the PMG is shown in Figure 11.

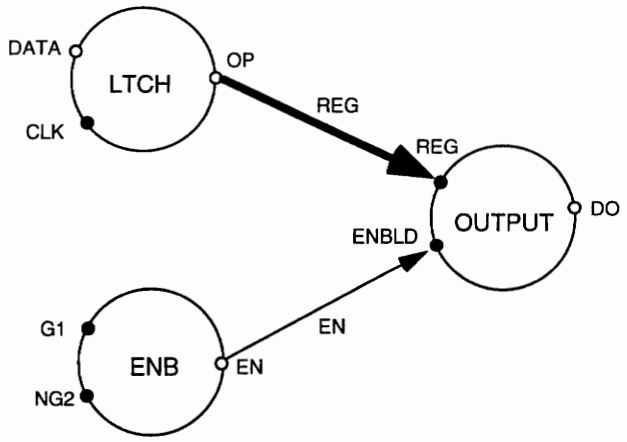


Figure 10. PMG for an 8-bit latch circuit

```
entity LATCH8 is
 port (DO: out MVL4_VECTOR(0 to 7); NG2: in BIT;
       G1: in BIT; CLK: in BIT; DATA: in BIT_VECTOR(0 to 7));
end LATCH8;
architecture BEHAVIORAL of LATCH8 is
 signal EN: BIT;
 signal REG: BIT_VECTOR(0 to 7);
begin

 process (EN,REG)
 begin
    if (EN = '1') then
            DO <= BV_to_MVL(REG);
    else
            DO <= "ZZZZZZZZ";
    end if;
 end process;

 process (NG2,G1)
 begin
```

```
    EN <= G1 AND NOT NG2;
  end process;

  process (CLK)
  begin
    if (CLK = '0') then
          REG <=  DATA;
    else
          REG <= REG;
    end if;
  end process;
end BEHAVIORAL;
```

Figure 11. VHDL source for the 8-bit latch circuit

Various symbols are used to represent signal values or transitions in signal values while writing the process test sets. These are described below [3]:

The Symbolic Notation used for Signals

'0'	*A constant '0' in a single-bit bus*
'1'	*A constant '1' in a single-bit bus*
'R'	*A transition from '0' to '1'*
'F'	*A transition from '1' to '0'*
'X'	*An unknown value*
'Z'	*A high-impedance state*
'D_i'	*Any chosen value in either a single-bit bus or a multi-bit bus, e.g., D1, D2, D3, etc.*
'P'	*A value which is the same as the last value of the signal in the previous time-frame*
'NP'	*A value which is "not" of the last value of the signal in the previous time frame*
'C_i'	*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 sequentially defined, i.e., C_{i+1} always follows C_i in a sequence*
'CP'	*The previous value in the sequence of control signals*
'CN'	*The next value in the sequence of control signals*

Figure 12 shows the process test data file for the process LTCH in the PMG of Figure 10. In the figure, test sets are preceded by the number of time frames required for each of the test sets [3]:

```
                    portorder: OP CLK DATA
Initialization frame: D2
                    1
                    P R D1
                    1
                    D2 F D2
                    1
                    D1 F D1
                    2
                    D1 F D1
                    D1 0 D1
                    2
                    D2 F D2
                    D2 0 D2
```

Figure 12. Process Test Data File for the process LTCH [3]

The process data files have the following salient features [3]:

(1) The first line gives the ***portorder***, i.e., the order in which the ports are stored in a linked list in the PMG database of Modeler's Assistant. The second line is the *initialization frame*. This is of significance in the case of sequential primitives where the output depends on its previous value. A value specified in the initialization frame corresponding to a particular output signal, is used as the initial value for the signal in

the hierarchical test generation process, e.g., the symbol 'D2' specified for output OP in the Figure 12, is used as the initial value for the signal during test generation.

(2) For each sensitive port of the process, there must be a test that generates an event on that port.

(3) In a particular time frame, only one port should have an event. All the other ports should be assigned constant stable values.

(4) 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 HBTG to justify any signal values assigned to a signal in the process during forward propagation.

(5) If a test is included where the output depends on the previous value, the test needs to have a symbolic value 'P' for the output port.

(6) In addition, symbols like 'C_i' and 'CX' are also used, as defined above, which are required for dealing with sequential circuits [3].

In the example test data file in Figure 12, when CLK rises, the output OP stays at its last value denoted by the symbol 'P'. When there is a fall 'F' on CLK, OP gets the value 'D2' on the input signal DATA. The initialization frame shows a value 'D2' corresponding to the output signal OP, so that HBTG will initialize OP to 'D2' before the actual test generation process starts.

3.2.2 Construction of Sensitive Paths

The HBTG algorithm consists of two parts. The first part is the construction of sensitive paths through the PMG of the circuit, and it is discussed in this section. The second part is the hierarchical test generation process which is discussed in the next section.

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 HBTG picks up a primary sensitive input port not yet selected as the first port of a new sensitive path. If the process to which the sensitive port belongs has only one output port, that port is selected as the next port along the path, otherwise that output port is selected which is on the least number of sensitive paths till then. If the output port selected has fanout greater than one, then the least activated of all the destination ports of all the fanout signals is chosen as the next port along the path. This is continued till a primary output port is reached and the sensitive path construction is completed. The number of sensitive paths constructed are equal to the number of primary sensitive input ports of the model. In the PMG of Figure 10, three sensitive paths are constructed as follows:

path 0 : NG2 => EN => ENBLD => DO;
path 1 : G1 => EN => ENBLD => DO;
path 2 : CLK => OP => REG => DO;

The sensitive path construction is a critical phase in the entire hierarchical test generation process. The activations of ports, the propagation of these activations and the justification processes all are performed along the sensitive paths.

3.2.3 HBTG algorithm

After the sensitive path construction, the test generation process starts which generates tests for the entire model using the precomputed process test sets 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. The selection of suitable test sets is done in accordance with the following rules [3]:

- For the activation of a sensitive port on the path selected, HBTG selects a test with an event on that port. The test should also have an event on the next port on the path as

well. A test with a constant value at the output is selected only by default, i.e., if a test with an event at the output port is not present.

- For the propagation of an event on the path, a test with the same event on the port is needed. Again, a test that has an event on the next port of the path is always selected over one that does not.
- In order to propagate a constant signal value on a sensitive input port or a value on a non-sensitive input port, HBTG selects a test with an event on another sensitive input port of the same process.
- HBTG may select a test whose output depends on the value of the signal in the previous time-frame. In such a case, the value on the output port is calculated based on the previous value in the test generation process.
- For the justification of a signal value, a test that can generate the same value on the signal is required.
- For the implication process, HBTG chooses a test with values on all the input ports of the process equal to the current values of the ports.

3.2.4 HBTG Example

We again consider the example of the 8-bit latch circuit shown earlier. The HBTG creates a test sequence for this circuit based on the pre-computed stimulus/response test sets developed manually for each of the processes in the PMG of its VHDL behavioral model. The final test sequence generated is as shown:

Test Sequence for 8-bit Latch [3]

frame	DO	EN	NG2	G1	OP	CLK	DATA
0	X	X	X	X	D2	F	D2
1	X	X	X	X	D2	0	D2
3	X	X	X	X	D1	F	D1
2	D1	R	F	1	D1	0	D1
4	Z	F	0	F	D1	0	D1
6	D1	R	1	R	D1	0	D1
5	D1	1	1	1	D1	R	D1

There are seven time-frames in the sequence. Frame 0 and 1 are the initial frames, where all the signal values are "unknown" or have been assigned values during initialization. Frames 3 and 2 are responsible for the activation of path 0. Frame 4 represents activation of path 1. Frames 6 and 5 are responsible for activation of path 2. During simulation the test sequence is applied in lexical order. The frame numbers indicate the order in which they are inserted in the final test sequence by HBTG.

3.3 Test Evaluation System

In the case of Hierarchical Behavioral Test Generation, an *effective* test sequence is defined as one that is short, checks various operations of the model and generates an event on every sensitive port of the PMG at least once [3]. The property of *Statement coverage* is used to evaluate the test sequence quality. Statement Coverage refers to the condition in which every statement in the model is executed at least once during simulation of the model.

The test sequence generated by HBTG is converted into a test bench before the test evaluation can be done. The test bench, which is manually generated, is used to simulate the model with the help of the Synopsys VHDL System Simulator [32]. To determine the effectiveness of the generated test sequence, the *coverage* property of the simulator is utilized. This property provides an environment to evaluate the number of times each statement in the VHDL model is executed during simulation. For example, the test sequence for the 8-bit latch model is converted into a test bench and the model is then simulated. A portion of the coverage results obtained are shown in Figure 13.

```
Line      Count              Text
-------------------------------------------------------
 46
 47              -- ---------------------------------------------------
 48               -- Process Name: LTCH
 49              -- ---------------------------------------------------
 50
 51               LTCH_14: process (CLK)
 52               begin
 53      6           if (CLK = '0') then
 54      3                   REG <=  DATA;
 55      3           else
 56      3                   REG <= REG;
 57      6           end if;
 58
 59
 60               end process LTCH_14;
 61
 62
 63              end BEHAVIORAL;
```

Figure 13: A portion of the Coverage Results for the 8-bit latch circuit

Chapter 4. Graph Transformation: The Base for Process Test Generation

The previous test generation system was inefficient in the sense that the generation of the process test sets for input to the HBTG, was done manually. The process test sets form a crucial part of the entire hierarchical test generation process. The final test sequence generated by HBTG depends on how the test sets are written. The manual generation of process test sets is not only time-consuming and tedious but may also be incomplete. The major criterion used by HBTG to evaluate the effectiveness of the entire system is that every statement of the VHDL model must be exercised at least once. Thus, a high-level process test generation algorithm, called Process Test Generator (PTG), has been developed that can automatically generate the process test sets that not only saves time but also provides stimulus/response tests for all the operations of the VHDL process.

Several research efforts have concentrated on test generation using data flow and control flow graph techniques [5, 10, 19]. In this chapter, the Process Test Generator (PTG), is

introduced 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 VTIP VHDL analyzer (CLSI/COMPASS) [31]. 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.

4.1 The Generation of VHDL Model for a PMG Process

The main purpose of the test generation system is to hierarchically generate the tests for a circuit that is represented by a Process Model Graph. A Process Model Graph is used because it gives the division of functionality within a VHDL behavioral model. Given a PMG and the functionality of each process, the Modeler's Assistant [29] generates the VHDL behavioral model for the entire circuit. However, the PTG algorithm (to be discussed in next chapter) requires the complete VHDL description of each process of the PMG. A complete description includes the entity declaration including the port declaration, architecture body, signal and variable declarations, and the process block inside the architecture body specifying the functionality of the process. This information is needed by PTG to generate the process test sets as explained in the next chapter.

The complete VHDL model for each PMG process can be obtained in two ways. The first method is to generate another PMG with just one process for which the complete description is required. Using the "*VHDL Dump*" option from the Modeler's Assistant's

main menu, the VHDL source code for the process can be obtained. Another method is to use a special program developed to generate the complete description. Using the "*Edit Process*" menu option a process can be selected from the PMG and the VHDL specifying the process block can be dumped. The "***mvhd.c***" program (*MakeVHDL*) is then invoked that automatically extracts the useful information from the PMG database and uses the dumped process block code to create a complete VHDL description for the process. The program makes the whole process efficient by providing an alternative to making a separate PMG for each process to obtain the complete VHDL description. The program "*mvhd.c*" uses the macros defined in the file "***macrosm.h***", the Modeler's Assistant structures defined in the file "***vhdlm.h***", the functions declared in the file "***pmg_info.c***" and the dumped process VHDL to generate the complete VHDL. The above programs and the files are explained in the programmer's manual in appendix A.

4.2 Interface to VTIP Software

The complete VHDL model for the PMG process, for which the process test sets are to be generated, is analyzed with the help of VHDL Tool Integration Platform (VTIP) Software (COMPASS Design Automation, Inc.) [31] before the process test generation can start. The PTG is interfaced to VTIP software because it is used to create a Control Flow Graph (CFG) for a VHDL behavioral model. The VTIP VHDL analyzer [31] analyzes the model and stores the analyzed design information in an intermediate form in VTIP's DESIGN LIBRARY SYSTEM (DLS) [31] which can later be used for CFG generation. The design information is stored in the form of DLS Design Libraries and Design Library Units. The Design Libraries are implemented as system directories while the Design Library Units are

implemented as files in those directories. The libraries and library units store design data information in a *view domain* with a hierarchy of *region nodes* forming the backbone of the domain. The DLS has two components [31]: Abstract component, called ***DLS Data Definition***, which specifies data elements, objects, structures and the abstract operations on them, and the Concrete component, called ***Software Procedural Interface*** (SPI) [31], which defines routines to extract the design information. The VTIP/DLS software is discussed in little more detail in appendix B.

4.3 Control Flow Graph: Definition and Properties

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 $N_j \in N$ [33].

Definition: A ***Node*** N_i is defined as an element in a Control Flow Graph (CFG) 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 thesis a CFG is used 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 Nj is a sequence of nodes N_i, N_a, N_b, ..., 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*** SP_{ij} between 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 (e.g., case statement condition, if-then-else clause condition, etc.) or it can be an assignment statement (e.g., signal or variable assignment) and accordingly the corresponding CFG node is either defined as a ***Control Node*** or an ***Assignment Node*** respectively. The false edge of an assignment node is always undefined while the true edge points to the next CFG node to be executed. The true and the false edges of a control node point to the next CFG node depending on whether the corresponding VHDL condition is true or false respectively. All the case clauses in a VHDL case statement are represented by a separate control node that is true or false depending on the case selector signal value. If the selector signal has the same value as that of the control node, the

assignment nodes following the clause are executed, else the execution falls through to the control node representing the next case clause.

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, range, mode 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. A declaration node holds declaration information like entity declaration, architecture declaration and VHDL process declaration while a signal/variable node contains the necessary information related to a signal or a variable, e.g., its type range, mode, sensitivity, value, etc. A generalized Directed Model Graph (DMG) link list structure generated by the process test generation software for a VHDL behavioral model is as shown in Figure 14. The Control Flow Graph (CFG) is a ***sub-graph*** of the DMG and just represents the data and control flow for the VHDL source within the process block of the architecture body of an entity. The link list structure of the generalized DMG is explained in the next section with the help of an example. All the different kinds of nodes of a DMG are implemented as distinct data structures in C/UNIX, and are described in detail in appendix A.

4.4 Graph Generation

A typical single-process VHDL behavioral model is shown in Figure 15. The bold italics are not a part of the normal VHDL syntax but are just included for clarity. The process

test generation software described in this thesis can generate a control flow graph for VHDL constructs like IF-THEN-ELSE, CASE, SIGNAL ASSIGNMENT, VARIABLE ASSIGNMENT and FOR LOOP statements. The FOR LOOP and IF-THEN-ELSE nesting can be handled to a reasonable complexity and depth.

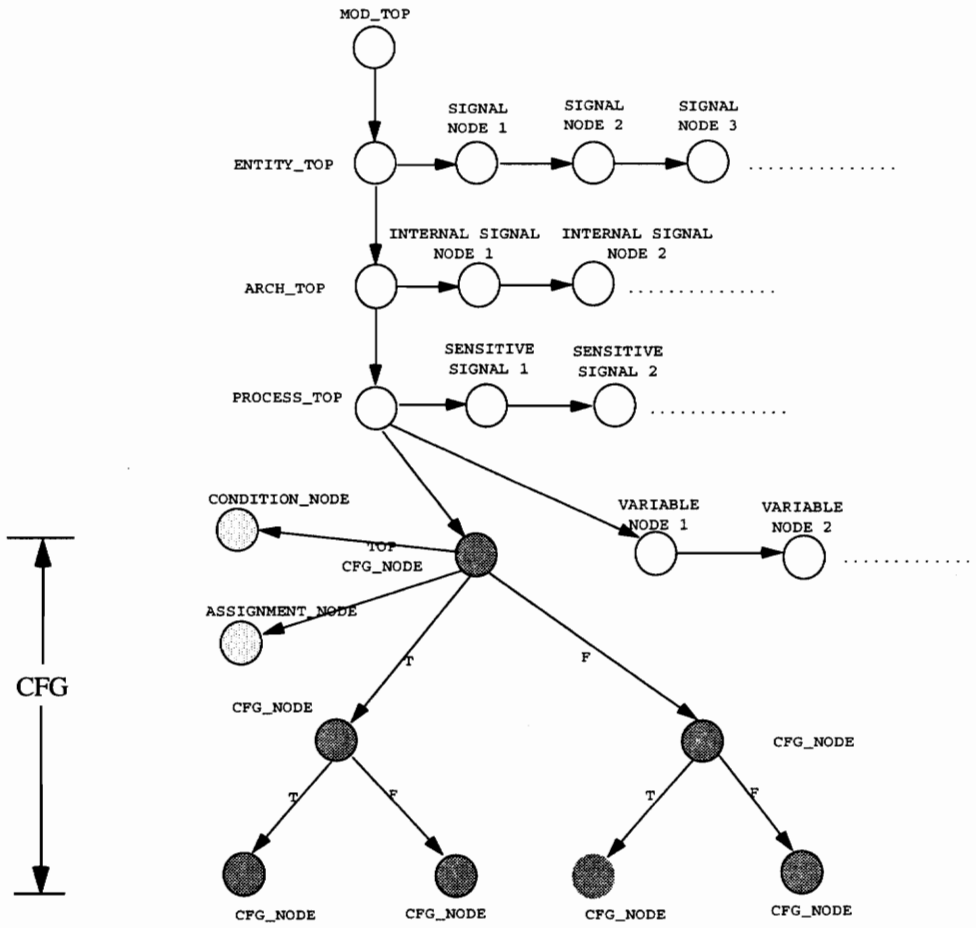


Figure 14. A Generalized DMG link list structure for a VHDL Behavioral Model

```
1|  entity DMG_EXAMPLE is
2|  port (SIG_A, SIG_B, SIG_C: in BIT_VECTOR(7 downto 0);
       SIG_D, SIG_E: in BIT;
       SIG_F, SIG_G: out BIT_VECTOR(7 downto 0);
       SIG_H: out BIT);
3|  end DMG_EXAMPLE;

4|  architecture BEHAVIOR of DMG_EXAMPLE is
5|  signal A, B: BIT;
6|  begin
7|     MODEL_1: process (SIG_A, SIG_D, SIG_E)
8|     variable X, Y: BIT;
9|      begin
10|        if (condition, e.g., SIG_B=SIG_C) then
11|             signal_assignment (say, SIG_H <= SIG_A);
12|        end if;
13|       case selector_signal (SIG_D) is
14|              when value_1 (e.g., '0') => SIG_F <= SIG_B;
15|              when value_2 (e.g., '1') => SIG_F <= SIG_C;
16|        end case;
17|        SIG_G <= SIG_E;
18|  end process MODEL_1;
19|  end BEHAVIOR;
```

Figure 15. A typical single-process VHDL Behavioral Model

The complete DMG for the model in Figure 15 is shown in Figure 16, which is generated in accordance with the generalized DMG of Figure 14. The shaded nodes represent the CFG_NODEs belonging to the control flow graph, the sub-graph of the DMG.

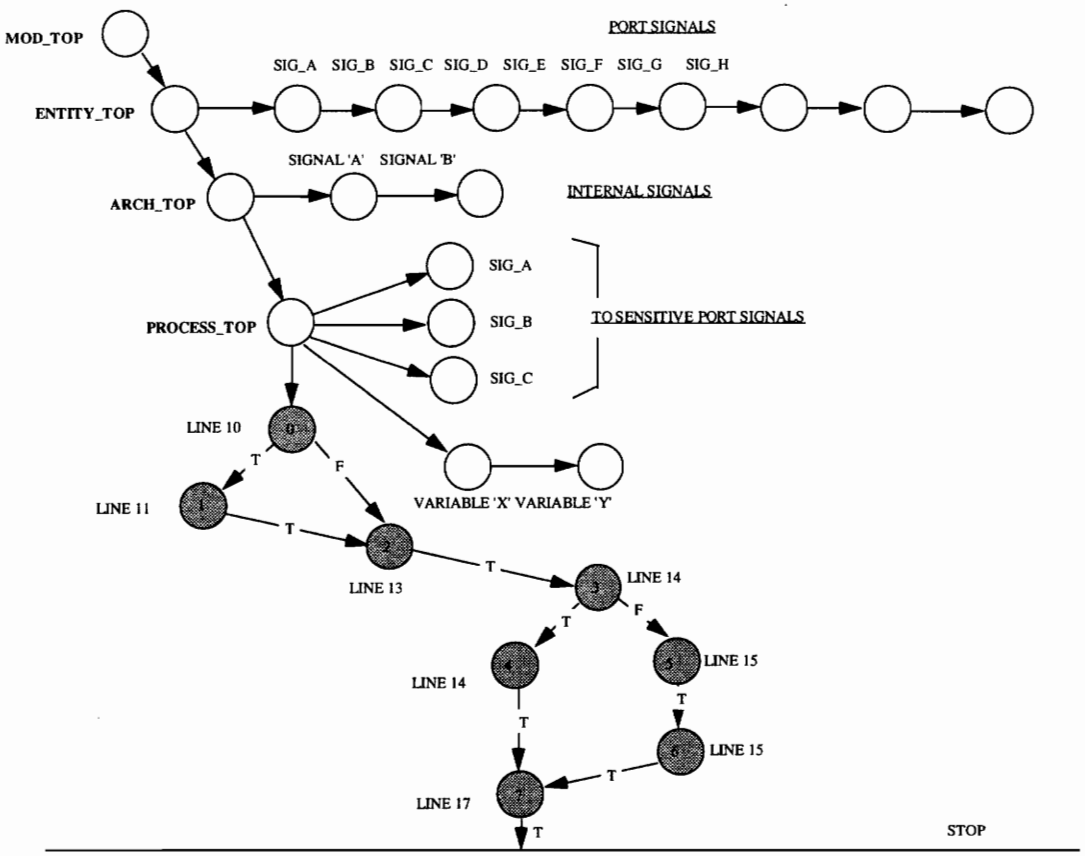


Figure 16. The DMG for the typical VHDL model of Figure 15

In the Figure 16, *mod_top* is a declaration node for the VHDL model. It holds the name of the model, i.e., the name of the VHDL model file, and has another field that points to the entity declaration. The entity declaration node is called the *entity_top* which holds the name of the entity, e.g., for the model in Figure 15, name field of entity_top is DMG_EXAMPLE.

The entity_top node also has a field pointing to the link list of port declaration nodes. Each of the port declaration nodes is represented by a data structure, called the *signal node*. The signal node holds all the relevant information pertaining to a signal, e.g., its sensitivity, range, mode, etc. In the case of the model shown in Figure 15, there are eight signal nodes linked together in a link list. These signal nodes correspond to the port signal declarations, i.e., signals SIG_A to SIG_H. These nodes have all the information about the signals, e.g., the mode field of the signal node for port SIG_A has the mode "InMode", its type field is "BIT_VECTOR", and so on.

Another field of the entity_top node points to the *arch_top* node. The arch_top node forms the locus of information for the architecture body declaration. It has a field containing the name of the architecture node and another field pointing to the link list of internal signal nodes. For the model of Figure 15, the name field of arch_top node is BEHAVIOR, and the internal signal nodes correspond to signals A and B. The internal signal nodes have the same data structure as the port signal nodes. The mode field of the internal signal nodes is "InternalMode".

A field of the arch_top node also points to the *process_top* node, i.e., to the declaration of the VHDL process block inside the architecture body. The process_top node has a field for process label, e.g., for Figure 15 model it is MODEL_1. The process_top node has another field pointing to the link list of any variables declared inside the process block. These variable nodes, called the *var nodes*, also have the same data structure as the signal nodes, but a field within the node distinguishes whether the node is a signal or a variable node. Again in the model of Figure 15, there are two variable nodes each for the variables X and Y.

A process_top node also has pointers pointing to the port signal nodes that are sensitive. The sensitivity list of the model shown in Figure 15 has three signals, SIG_A, SIG_B and SIG_C. Thus, the process_top node has three pointers pointing to the signal nodes for these signals. The *sense* field of these nodes is '1'.

The process_top node also has a field that points to the top of the control flow graph. Each node of the control flow graph is called a ***CFG_NODE***. Each CFG_NODE has pointers to two special nodes, called the *condition_node* and the *assignment_node*. As already explained, a condition_node contains the information about a condition specified by the corresponding VHDL source statement while the assignment node consists of the information corresponding to an assignment statement inside the source. A CFG_NODE is called a ***control_cfg_node*** if the assignment_node is undefined and only the condition_node is defined, i.e., if it corresponds to a control condition; it is called an ***assignment_cfg_node*** if the assignment_node is defined but the condition node is undefined, i.e., if it corresponds to a VHDL assignment statement. In the model of Figure 15, the CFG_NODEs corresponding to the statement numbers 10, 13 and "when" case clause parts of lines 14 and 15 are control_cfg_nodes, while the CFG_NODEs corresponding to lines 11, signal assignment parts of lines 14 and 15, and line 17 are assignment_cfg_nodes.

Each CFG_NODE data structure defined, has two pointers to the following CFG_NODEs, called the ***true_edge*** and the ***false_edge***, as already stated. In the case of a condition_cfg_node, the true_edge points to a cfg_node which is executed next if the condition is true, and the false_edge points to a cfg_node which is executed next if the

condition is false, else they are undefined. The false_edge of an assignment_cfg_node is always undefined while the true_edge points to the next CFG_NODE to which the execution falls through. In the example model of Figure 15, the true_edge of line number 10 CFG_NODE points to line number 11 CFG_NODE and its false_edge points to the line number 13 CFG_NODE. A case selector signal condition_cfg_node just has a true_edge pointing to the first case clause condition_cfg_node. The true_edge of the line number 13 CFG_NODE of Figure 15 points to the line number 14 case clause condition cfg_node. The true_edge of the line number 14 case clause node points to the line number 14 signal assignment node while its false_edge points to the line number 15 case clause condition_cfg_node. The last case clause condition_cfg_node just has a true_edge because if the execution reaches this node, the following signal/variable assignment statement(s) have to be executed, e.g., in the Figure 15 model, the line number 15 case clause condition_cfg_node just has a true_edge pointing to the assignment_cfg_node (i.e., node number 6) at the same line.

The true_edge of line number 11 assignment_cfg_node points to the line number 13 condition_cfg_node, line number 14 and 15 assignment_cfg_nodes point to the line number 17 assignment_cfg_node while the true_edge of the line number 17 assignment_cfg_node is undefined signaling the end of execution if this node is reached.

Besides these, a CFG_NODE has various other fields which contain the information like the CFG_NODE number, the corresponding VHDL source line number and the number of references to the node.

4.5 Hash Table Generation

The PTG algorithm, as explained in the next chapter, processes a CFG and generates stimulus/response test sets for the VHDL behavioral model. The test sets give the responses of the output ports given a certain stimulus on an input port. The PTG algorithm selects each sensitive signal of the VHDL process one by one and searches for a suitable CFG node associated[3] with that signal to create a stimulus. It then searches and finds a shortest path from this node to other CFG nodes associated with different output ports in order to find the response. The search and comparison processes to find an appropriate node could slow down the algorithm and consequently could slow down the entire test generation process. In order to overcome this, a ***hash table*** organization and search technique [34] is implemented which eliminates most unnecessary comparisons and speeds up the test generation algorithm.

A hash table is implemented as an array of fixed size. A hash function "***hash***" is used to transform a signal name into a table index '*k*' where ***k=hash(signal name)*** [34]. The table index 'k', called the ***hash of key*** [34], gives the index where the CFG nodes associated with the signal should be placed or linked. All the CFG nodes associated with the same signal are linked together in a link-list and are also linked to the hash of key index of the hash table, as given by the hash function.

However, most of the times a situation might occur that two or more signal names may map to the same hash function value and thus to the same hash table entry. In order to avoid this *clustering*, *double hashing* has been used which involves the use of two hash

[3] A signal or a variable that occurs in a VHDL statement is said to be associated with the corresponding CFG node.

functions, *hash(signal name)* and *hash2(signal name, hash_table_entry).* The psuedocode for the double hash functions used is as shown below:

```
function hash (input: signal_name)

variable    i, hash_table_entry: integer;
begin
        hash_table_entry := 0;
        while the last character of the signal_name, i.e., signal_name[i], is not NULL
do
            hash_table_entry := signal_name[i] + hash_table_entry;
            increment i;
        end while;
        hash_table_entry := hash_table_entry mod hash_table_size;
        if this hash_table_entry is already occupied then
            hash_table_entry := hash2 (signal_name, hash_table_entry);
        end if;
        return hash_table_entry;
end function hash;


function hash2 (input: signal_name, input: hash_table_entry)

begin
        hash_table_entry := 1 + (hash_table_entry mod (hash_table_size-1));
        if this hash_table_entry is occupied then
            hash_table_entry := hash2 (signal_name, hash_table_entry);
        else
            return hash_table_entry;
        end if;
end function;
```

Figure 17. The Double Hash Functions

In the function "hash", the integers for each character of the signal_name are summed and the result is divided by the size of the hash table. The remainder, which is a value from 0 to table_size-1, is the hash table index where the signal is linked. However, if the hash of key is already occupied, then a rehashing function "*hash2*" is successively called till there is no *clustering*. The *rehash function "hash2"* is shown in the pseudo-code above. The table size is decreased by one and then the modulus is calculated, and a '1' is added to the result to avoid conflicts. The table size used is a prime number (*52 in this case*) which further helps in assigning unique *hash of keys* to each signal. The "***makehash***" program is described in appendix A. This program is used to calculate a hash of key value for each signal and is used to connect the associated CFG_NODEs in a link list to this hash of key value.

The hash table organization makes the search and comparison processes considerably faster. Given a signal name, the hash function directly gives the table index of the hash table where the associated CFG nodes are linked. The PTG then selects the appropriate node from the link list. A hash table constructed for a VHDL behavioral model like the one in Figure 14, looks like as shown in Figure 18. The entry "key *i*" in the parenthesis following a signal name in the table is the hash of key index for the signal.

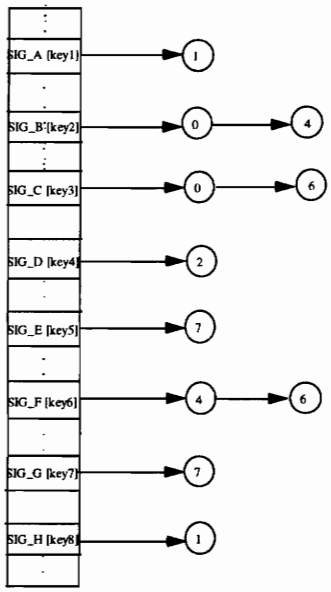


Figure 18. A Hash Table for the typical VHDL Model of Figure 14

4.5 Examples of Graph Generation

4.5.1 The Example of a Register Model

A VHDL behavioral model for a register circuit, called REG, is shown in Figure 19. The output listing with the source lines numbered, as shown, is produced when the model is analyzed with a "-list" qualifier option using VTIP VHDL analyzer [31]. A complete

DMG produced by the process test generation software for the VHDL model REG is as shown in Figure 20.

```
Levels  LINE # |----+----1----+----2----+----3----+----4----+----5----+----6-
              1 |use WORK.VHDLCAD.all, WORK.USER_TYPES.all;
              2 |entity REG is
1             3 |  port (D_OUT: out  BIT_VECTOR(7 downto 0); CLK: in  BIT; LD: in  BIT;
D_IN: in  BIT_VECTOR(7 downto 0); CLR: in  BIT);
1             4 |end REG;
              5 |architecture BEHAVIOR of REG is
1             6 |begin
1             7 |   REG_1: process (CLK,CLR)
2             8 |   begin
2             9 |      if CLR = '1' then
2 1          10 |           D_OUT <= "00000000";
2 1          11 |      elsif CLK'EVENT and CLK = '0' then
2 1          12 |        if LD = '1' then
2 2          13 |           D_OUT <= D_IN;
2 2          14 |        end if;
2 1          15 |      end if;
2            16 |   end process REG_1;
1            17 |end BEHAVIOR;
```

Figure 19. VHDL source for a process REG

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, i.e., signals D_OUT, CLR, LD, D_IN and CLK. The CFG_NODEs are shown by the shaded circles. The

CFG_NODEs 0, 2 and 3 are *condition_cfg_nodes* corresponding to the lines 9, 11 and 12 respectively in Figure 19. Similarly, the CFG nodes 1 and 4 are *assignment_cfg_nodes* corresponding to lines 10 and 13.

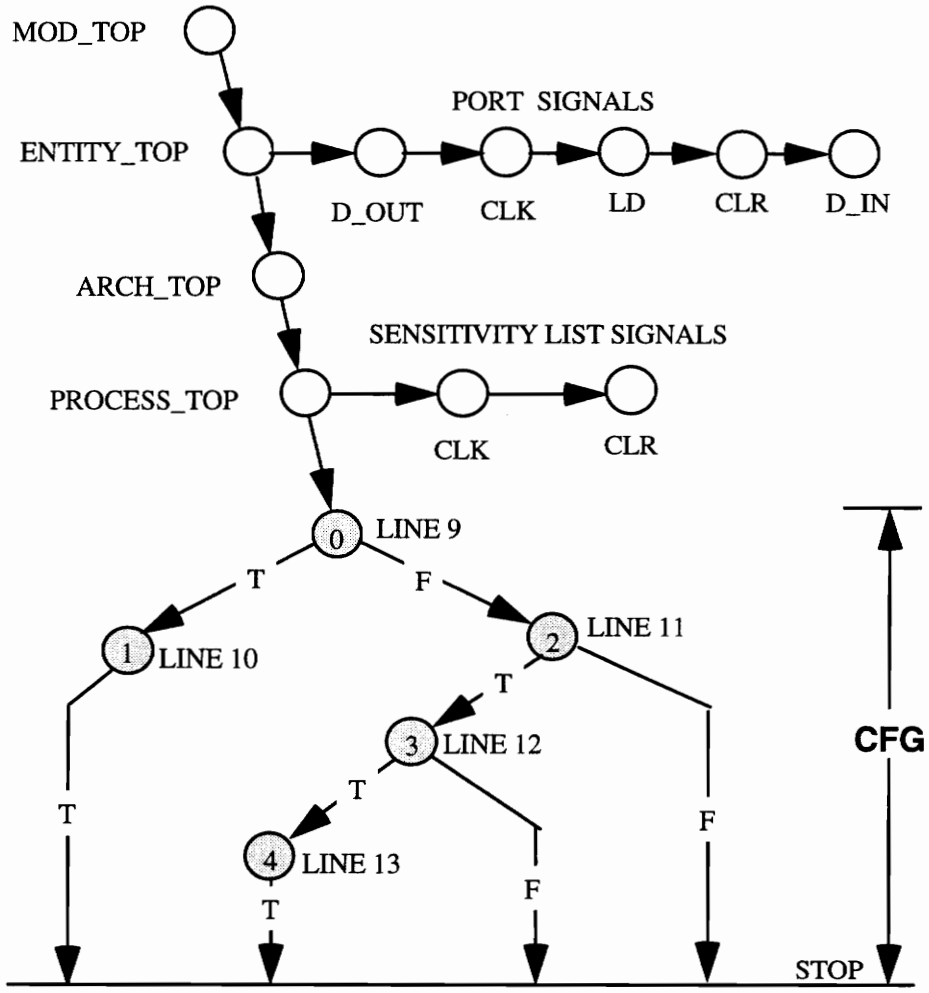


Figure 20. A Complete DMG for the process REG

The hash table organization for the process REG is as shown below in Figure 21. The numbers in the parenthesis following the signal name denote the hash of key value or the hash table index where the associated CFG_NODEs are linked. The CFG_NODEs are shown by their node numbers and the corresponding VHDL statement number.

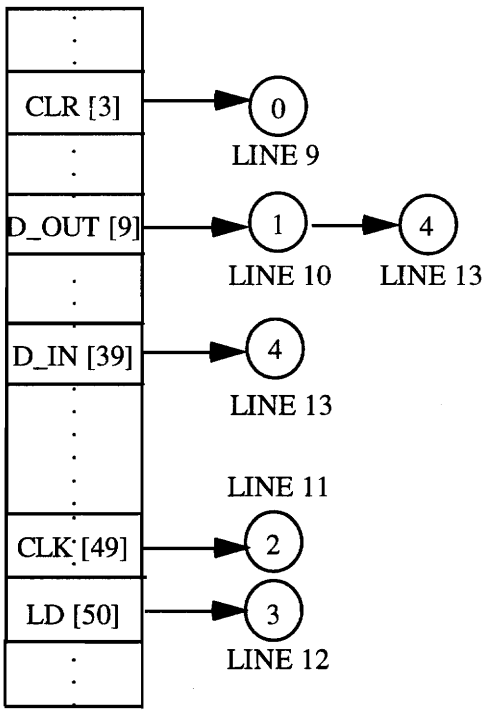


Figure 21. A Hash Table for the process REG

4.5.2 Example of a VHDL behavioral model for a Multiplexer

The VHDL behavioral model for a multiplexer, called MUX, is shown in Figure 22 below. As in the case of the process REG, the model MUX is also analyzed using VTIP VHDL analyzer with a "-list" option so that the listing obtained is as shown in the figure. It consists of a CASE statement with four case clauses.

```
Levels  LINE # |----+----1----+----2----+----3----+----4----+----5----+----6-
          1 |
          2 |use WORK.VHDLCAD.all, WORK.USER_TYPES.all;
          3 |-- ************************************************************
          4 |entity MUX is
1         5 |  port (DATA_OUT: out BIT_VECTOR(0 to 7);
1         6 |    D: in BIT_VECTOR(0 to 7);
1         7 |    C: in BIT_VECTOR(0 to 7);
1         8 |    B: in BIT_VECTOR(0 to 7);
1         9 |    A: in BIT_VECTOR(0 to 7);
1        10 |    SEL: in BIT_VECTOR(0 to 1);
1        11 |    ENB: in BIT);
1        12 |end MUX;
         13 |-- ************************************************************
         14 |
         15 |architecture BEHAVIORAL of MUX is
1        16 |
1        17 |begin
1        18 |
1        19 |  -- ---------------------------------------------------
1        20 |  -- Process Name: MUX
1        21 |  -- ---------------------------------------------------
1        22 |
1        23 |  MUX_3: process (SEL,ENB)
2        24 |  begin
2        25 |     if (ENB='1') then
2 1      26 |        case SEL is
2 2      27 |              when "00" => DATA_OUT <= A;
2 2      28 |              when "01" => DATA_OUT <= B;
2 2      29 |              when "10" => DATA_OUT <= C;
2 2      30 |              when "11" => DATA_OUT <= D;
```

```
2 2     31 |       end case;
2 1     32 |   else
2 1     33 |       DATA_OUT <= "00000000";
2 1     34 |   end if;
2       35 |
2       36 |
2       37 | end process MUX_3;
1       38 |
1       39 |
1       40 |end BEHAVIORAL;
```

Figure 22. VHDL source for a process MUX

The process test generation software transforms this model into a graph as before. The Figure 23 just shows the Control Flow Graph (CFG) that is constructed for this model. The whole DMG can be constructed as before. It is reminded that a CFG is a *sub-graph* of a DMG.

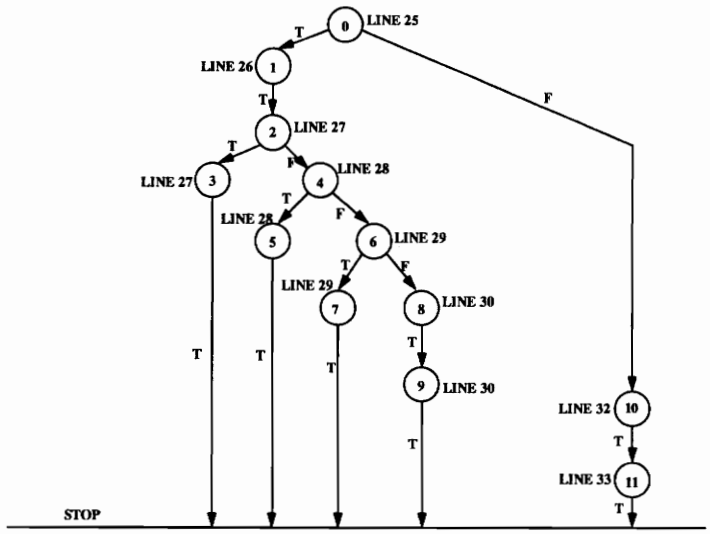


Figure 23. A CFG for the process MUX

The CFG_NODEs 0, 1, 2, 4, 6 and 8 are the condition_cfg_nodes while the nodes 3, 5, 7, 9 and 10 are the assignment_cfg_nodes. The node number 1 corresponds to the CASE SELECTOR signal statement at line 26. It has just the true_edge pointing to the first case clause condition_cfg_node 3 (i.e., *when "00"* clause) at line number 27. Similarly, the last case clause condition_cfg_node 8 also just has a true_edge pointing to the assignment_cfg_node corresponding to the signal assignment statement (i.e., *DATA_OUT <= D;*) at the same line, i.e., line number 30.

The hash table constructed by the process test generation software for the process MUX is as shown below in Figure 24. The hash table shows the signal names and the hash of key index corresponding to each port signal. It also shows the link list of all the CFG_NODEs associated with a signal, linked at the hash of key index for the signal.

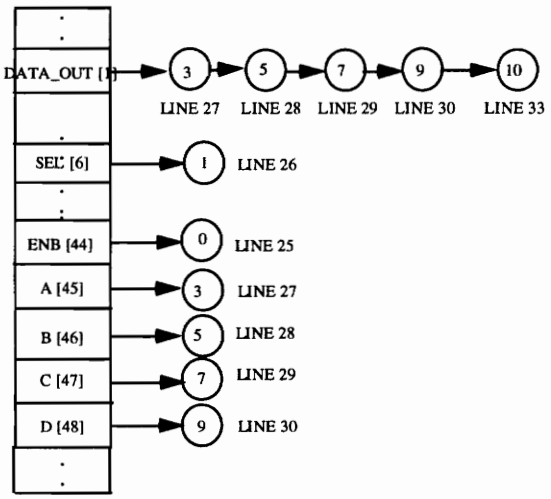


Figure 24. A Hash Table for the process MUX

Chapter 5. The Process Test Generation

5.1 Programming Environment

The process test generation software has been developed in Sun-C and is running on a Sun SPARCstation2. The software is distributed among 11 files including the header files and the C/UNIX files. These files constitute about 6500 lines of code.

The Control Flow Graph Generation algorithm of the process test generation software, as explained in the last chapter, makes an extensive use of the Software Procedural Interface (SPI) which is a fundamental part of the VTIP Design Library System (DLS) [31]. The SPI provides a standard interface to the process test generation software to access any analyzed design information that is stored in the DLS.

In order to use SPI under Sun/SunOS environment and with the Sun-C code developed for the process test generation, the entire software is compiled with "***SPI.h***" header file, consisting of the other SPI definition files, and is linked to the SPI Object Libraries of the VTIP software. This facilitates the use of these libraries and the SPI routines from within the process test generation program. The files compiled together include ***head.h, decl.h, vhdlm2.h, externs.h, macrosm.h, ptgmain.c, misc.c, makehash.c, pmg_info2.c, tgen.c and assign_tgen.c***. The compilation procedure for these files and the SPI libraries is described in appendix A. A brief introduction to VTIP/DLS and SPI software is given in appendix B.

5.2 Process Test Generation Methodology

5.2.1 Assumptions

Certain assumptions have been made during the development of the process test generation software. They are listed below:

- The VHDL behavioral model for which the process tests are to be generated , is a single architecture model and the only construct allowed inside the architectural body is a single process block.

- The model is analyzed using both the VTIP VHDL analyzer and the Synopsys VHDL analyzer [32].

- All the signal assignment statements within the model have delta delays.

- The only signal types allowed are the BIT and the BIT_VECTOR.

- The only constructs allowed for the test generation are the IF-THEN-ELSE, CASE, signal assignment and variable assignment statements. However, the Control Flow Graph Generation part of the software can also work for the FOR-LOOPs.

- The only user defined functions that can be used are the ones having just one argument, i.e., of the type: function(argument), e.g., INC(CNT), a function which increments the signal CNT by one.

- The special constructs like the concatenation ('&'), indexed signals or variables (e.g., A(I)) etc. are not allowed. But the CFG generation can handle the indexed signals.

- The IF-THEN-ELSIF-THEN-ELSE conditions that can be handled by the software are of the type: ***(operation1) operator (operation2)***. The operator can be any of the functions: "and", "or", "xor". The operation1 and operation2 are of the type: *signal_name = "signal_value"*.

  The condition can also be simply of the type *signal_name = "signal_value"*, as shown above. The signal or variable assignment statements can, however, handle the function "not" also in addition to the above.

5.2.2 Criterion of Process Test Generation

In high level circuit description, it is difficult to define fault models, such as stuck faults for gate level test generation. In higher level approaches like BTG [6] fault models like Control Faults and Micro-operation Faults have been defined. In the case of PTG algorithm, no explicit fault model is assumed.

In chapter 3, it was stated that an effective test sequence generated by HBTG is the one that activates all the sensitive ports of a VHDL behavioral model at least once. The aim is to generate a test sequence that fully exercises the model by testing all the operations of the model. The HBTG generates a test sequence that depends on the pre-computed test sets. Thus, the test sequence generated by HBTG will be complete only if the process test sets are complete. A process test file is complete if it contains stimulus/response test sets that test all the operations of the primitive VHDL behavioral model. In order to achieve this definition of completeness, the PTG tries to generate test sets that fully exercise the functionality of a process. Thus, 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.***

This notion is, infact, a measure of the effectiveness of the process test sets generated by the process test generation software.

5.3 Process Test Generator: Algorithm

5.3.1 The Concept of Controllable and Observable Nodes

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 [2]. Hence, a test set that creates an event on a sensitive input port of the model is better than the one that does not for the purpose of exercising the model. The PTG algorithm uses this to create stimulus/response test sets.

The algorithm picks each sensitive signal one by one and searches for the first CFG node, that is associated with the sensitive signal, with the help of the hash table described in the last chapter. The CFG node selected to create an event on the associated sensitive signal is called a ***Controllable 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 has not been chosen as yet, as the CN. The pseudo-code of the algorithm used to find the controllable node is shown in Figure 25.

---

procedure **find_controllable_node** *(**input**: sensitive_signal_name)*

begin
find the hash_of_key using the hash function;
get the first CFG_NODE of the link list of CFG_NODEs associated with the sensitive signal and linked at the hash_of_key hash table index;

Repeat
if *the CFG_NODE is already selected as a controllable_node,* ***then***
select the next CFG_NODE from the link list at the hash_of_key hash table entry;

```
    else
        the CFG node selected is the controllable_node (CN);
    end if;
  Until a CFG_NODE is selected which has not be selected as a controllable_node earlier, or till
all the CFG_NODEs in the link list are checked;

  if all the CFG_NODEs of the link list are checked, i.e., all are selected as a
  controllable_node before, then
    choose the first CFG_NODE of the link list as the controllable_node (CN);
  end if;

return the CN;

end procedure;
```

Figure 25. An Algorithm for finding a suitable Controllable Node (CN)

Once the controllable node is selected, a suitable event is then created on the sensitive signal. The event created depends on whether the node selected as a controllable node, is a ***condition_cfg_node*** or an ***assignment_cfg_node***. If the controllable node is a condition_cfg_node, then the event created is such that the new value is the value required to make the condition true, e.g., consider the following condition:

```
1|      if (CLK = '1') then
2|          signal_assignment;
3|      end if;
```

In the above case, let CLK be a sensitive signal. If the node corresponding to the statement number 1 is chosen as a controllable node, then the event created on the sensitive signal is a transition from '0' to '1', i.e., a rise denoted by 'R'.

Similarly, if the node selected as a controllable node is an assignment_cfg_node, then the user is prompted to assign two different values to the sensitive signal to create an event. This generates two different test sets. Consider the following case:

```
1|   OUT_SIG_X <= SEN_SIG_A and SIG_C;
2|   OUT_SIG_Y <= SEN_SIG_B;
```

Let signals SEN_SIG_A, SIG_C and OUT_SIG_X be the BIT signals, and signals OUT_SIG_Y and SEN_SIG_B be the BIT_VECTOR signals. Moreover, let SEN_SIG_A and SEN_SIG_B be the sensitive input signals. If SEN_SIG_A is selected to create an event and the assignment_cfg_node corresponding to the assignment statement number 1 is chosen as the controllable node (CN), then both the events, a fall 'F' and a rise 'R', are created on the signal at the CN. Similarly, if SEN_SIG_B is the sensitive signal selected and the assignment_cfg_node corresponding to the statement number 2 is the controllable node, then the user is prompted to assign both a first value to the signal and then a new value to it. This represents a transition and two test sets are created by the PTG software.

In order to generate the stimulus/response test set, the stimulus given to the controllable node (CN), as described above, 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 pseudo-code of the algorithm to find the observable node is shown in Figure 26.

```
procedure find_observable_node (input: controllable_node)
begin
  select the first ouput signal;

  Repeat
    find the hash of key table index for the selected output signal;
    get the first CFG_NODE from the link list of CFG_NODEs associated with the output signal
    and linked at the hash of key table index;

    while a CFG_NODE exists and the CFG_NODE number is less than the
  controllable_node number, do
      get the next CFG_NODE along the link list at the hash of key index;
    end while;
    while a CFG_NODE exists and its node number is greater than or equal to the
          controllable_node number, do
      find the shortest path (SP) from this node to the controllable_node using the Shortest_Path
      algorithm (to be described later);
       if the SP exists, then
          the selected CFG_NODE is the observable_node (ON);
      else
        get the next CFG_NODE along the link list at the hash of key table index;
      end if;
    end while;

  select the next output signal;
  Until, an observable node (ON) is chosen;

  return the ON;
end procedure;
```

Figure 26. An Algorithm to find an Observable Node (ON)

In the construction of a Control Flow Graph, the node numbers assigned to the CFG_NODES are in the ascending order from the first CFG_NODE corresponding to the first VHDL statement inside the process block of the architecture body to the last VHDL statement of the process block. The CFG_NODE representing the first VHDL statement is called the ***START NODE (SN)***.

As stated earlier, the event assigned to the sensitive input signal at the selected controllable node (CN) has to be propagated forward at an output so that the response can be observed. Thus, the event has to be propagated from a CFG_NODE with a lower node number, which is a CN, to the CFG_NODE with the same node number (i.e., the same CFG_NODE) or to a CFG_NODE with a higher node number; such a node is an ON. This condition of node numbers is checked throughout the *find_observable_node* algorithm.

The find_observable_node algorithm selects the first output signal port and finds its hash of key index with the help of the hash function. The first CFG_NODE of the link list of CFG_NODES linked at the hash of key table index is selected. If the node number of this node is less than the node number of the controllable node, then the next CFG_NODE along the link list is chosen and the node numbers are again compared. If no CFG_NODE in the link list has node number greater than or equal to the node number of CN, the next output signal is chosen and the whole process is repeated till finally a CFG_NODE associated with an output signal is found, which has node number greater than or equal to the controllable node. It is then checked to see if it is possible to construct a path from the controllable node to this node with the help of the ***Shortest_Path Algorithm*** to be discussed in the next section. If no such path exists, the next CFG_NODE along the link list at the present hash of key index is selected and the whole process is repeated. If there is no CFG_NODE in the link-list which has node number greater than or equal to the controllable node number and to which a path exists from the CN, the next output signal is selected and the process is repeated again. However, if a CFG_NODE exists such that it is possible to construct a path from the CN to this node and if the node number of this

node is not less than the CN number, then this CFG_NODE is selected as the observable node (ON) and is returned to the main program, exiting the procedure. This completes the selection of the controllable node and the observable node.

5.3.2 The Construction of the Forward Propagation and the Justification Paths

After the controllable node (CN) and the observable node (ON) are found, it is possible to propagate the activation on the sensitive signal at the CN to the ON to observe the response. In order to propagate the event to the output signal, a path is constructed from the controllable node to the observable node. This path is called the ***forward propagation path*** **(FPP)**.

Also, the event assigned to the sensitive signal at the controllable node has to be justified backwards to the first CFG_NODE representing the first VHDL statement of the process, i.e., it is required to find the conditions that satisfy the assigning of a particular value to the sensitive signal. In order to do this, a path is constructed from the START NODE to the controllable node (CN). This path is called the ***justification path*** **(JP)**.

Both the justification and the forward propagation paths are the shortest paths possible from their begin nodes to their end nodes. The ***begin node*** is the first node of the path and ***end node*** is the last node of the path, e.g., in the case of FPP the begin node is CN and the end node is ON, while in the case of JP the begin node is SN and the end node is CN. The shortest paths are constructed using a modified form of Dijkstra's famous

shortest path algorithm [34]. The pseudo-code of the *Shortest_Path Algorithm* is shown in Figure 27.

```
procedure Shortest_Path (inputs: begin_node, end_node)
variable new_distance;
begin

  new_distance := 0;
  make the distances of all the CFG_NODEs equal to 0;

  if begin_node_number is equal to the end_node_number, then
    there is only one node in the path, i.e., the begin_node or the end_node itself because they are
    same;
    return the single SP node;
  else if begin_node_number is greater than the end_node_number, then
    there is an error;
    program terminated;
  else

    let begin_node be the nut;

    while nut_number is not equal to (end_node_number-1), do

      if true_node exists and the true_node_number is less than or equal to the
      end_node_number, then
        new_distance := nut_distance + 1;
        if true_node_distance is equal to zero or greater than the new_distance, then
          true_node_distance := new_distance;
          select the true_node as the successor_node of the nut along the SP found thus far;
          select nut as the predecessor_node of the true_node along the SP found thus far;
        end if;
      end if;

      if false_node exists and the false_node_number is less than or equal to the
      end_node_number, then
        new_distance := nut_distance + 1;
        if false_node_distance is equal to zero or greater than the new_distance, then
          false_node_distance := new_distance;
          select the false_node as the successor_node of the nut along the SP found thus far;
          select nut as the predecessor_node of the false_node along the SP found thus far;
        end if;
      end if;
```

```
    make the next CFG_NODE as the nut;

    end while;

    select the end_node as a_sp_node;

    Repeat
    select the predecessor_node of a_sp_node as the next higher node along the SP;
    choose this predecessor_node as a_sp_node now;
    Until a_sp_node_number is equal to or less than the begin_node_number ;

    if a_sp_node_number is less than the begin_node_number, then
      no path exists from the begin_node to the end_node;
      return NULL;
    else
      the shortest path exists;
      return the first node of the shortest path, i.e., the begin_node;
    end if;

  end if;

end procedure;
```

Figure 27. A Shortest_Path Algorithm

Some terms used in the Shortest_Path algorithm are described below:

- The ***distance*** is defined as the distance of a node from the begin_node. In the shortest path construction, an edge between two nodes is given a *weight* 1. Thus, the distance of a node from the begin_node is the number of edges along a path that the node is away from the begin_node.

- The ***SP*** is the shortest path constructed at any time.

- The ***nut*** (*Node Under Test*) is the CFG_NODE under consideration at any time.

- The ***new_distance*** is a variable that calculates the new distance of the nodes at the edges (true and false) of the *nut* at any moment.

- The ***nut_number*** is the node number of the *nut* in the Control Flow Graph.

- The ***end_node_number*** is the node number of the ***end_node*** in the CFG.

- The ***true_node*** is the next node along the true edge of the *nut*.

- The ***true_node_number*** is the node number of a *true_node*.

- The ***nut_distance*** is the distance of the *nut* from the ***begin_node***.

- The ***true_node_distance*** is the distance of the *true_node* from the *begin_node*.

- The ***successor_node*** is the next node selected along the shortest path constructed at any time.

- The ***predecessor_node*** is the preceding node selected in the shortest path construction at any time.

- The ***a_sp_node*** is any CFG node along the final shortest path.

- The ***a_sp_node_number*** is equal to the node number of the shortest path node in the CFG.

The Shortest_Path algorithm initializes the distances of all the CFG_NODEs to zero before starting the shortest path construction. If the begin_node is the same as the end_node, the Shortest_Path algorithm returns the only node that constitutes the shortest path, i.e., the node itself. If the begin_node number is greater than the end_node number, then there is an error in the program and the program is terminated. However, if the begin_node number is greater than the end_node number, then the algorithm selects the begin node as the node under test, i.e., **nut**. The algorithm selects each CFG_NODE, starting from the begin_node to the node with node number one less than the end_node_number, as the nut by incrementing the node numbers by one at a time. It also maintains a variable *new_distance*, as already described, that calculates the new distance of the nodes at the edges (true and false edges) of the nut at any moment. For each nut selected, the algorithm compares the distance of the true_node and the false_node of the nut calculated before with the new distance. If the true_node_distance or the false_node_distance are zero, it means that the nodes are being accessed the first time and are assigned the distances equal to the new_distance. Similarly, if the true_node_distance or the false_node_distance are greater than the new_distance, it means that the new_distance calculated is less than the one calculated before and that the new path selected is shorter. Thus, the true_node_distance and the false_node_distance are changed to the new_distance. Also, the nut is selected as the predecessor node in either

case and the nut chooses either of the two as its successor node whatever be the case. The selection of the next CFG_NODEs as nuts is continued and the new selection of predecessor and successor nodes for the CFG_NODEs are made accordingly, till the end_node is reached. Now, the final linking of the shortest path nodes takes place starting from the end_node and moving backwards along the predecessor nodes till the node number equals or falls below the begin_node_number. If the node number falls below the begin_node_number, then no path exists between the two nodes, else the first shortest path node, i.e., the begin node is returned to the main program. This completes the shortest path construction.

5.3.3 Creation of Test Benches and Simulation

Once the justification path (JP) is constructed between the START NODE (SN) and the controllable node (CN) and the forward propagation path is constructed between the controllable node (CN) and the observable node (ON), the input signal values are determined along these paths that result in the propagation and justification of events towards the ON and the SN respectively.

In order to find the conditions on the input signal values along a path, the PTG algorithm selects each condition_cfg_node along the path and determines the condition required on the node so that the next node along the path is selected. If the next path node is along the true edge of the node, then the condition required is "TRUE" otherwise it is "FALSE".

After the conditions for the propagation of values are determined, the actual input signal values are found along the path. The input signals associated with the node are assigned values so as to make the condition true or false, as determined earlier. For example, consider the following condition:

if (A = '1' and B = '0') then
*< **signal/variable assignments** >;*
end if;

If the condition_cfg_node corresponding to the IF-THEN VHDL statement is on the path and the condition is determined to be true, then the signal A is assigned a value '1' and the signal B is assigned a value '0', However, if the condition is determined to be false, then the signal A is assigned the value '0' and the signal B is assigned the value '1'. Similarly, the values are found for the other input signals depending on the conditions determined along the JP or the FPP. User is prompted to assign the values to any input signals that are not determined along the two paths.

All the assignment_cfg_nodes along the JP and the FPP are marked as passed. This is done to ensure that every assignment statement inside the process is executed at least once, which is our main criterion for process test generation.

In order to observe a response to the event, the input signal values calculated above are put in a test bench and the test bench is simulated to get the responses at the outputs. A command file, called "***Model_Name.con***" is created for the simulation purposes. The simulator used is the Synopsys VHDL System Simulator [32]. A typical *.con* command

file used is shown in the Figure 28. The bold italics represent the portion that vary from model to model.

---

cd
open ***Model_Entity_Name****.out*
logtime -e ***Model_Entity_Name****.out*
echo $cwr > ***Model_Entity_Name****.out*
set BASE BINARY
monitor -n smon -o ***Model_Entity_Name****.out active* ***Model_Entity_Name****_TB/*'sig*
monitor -n smon -o ***Model_Entity_Name****.out event* ***Model_Entity_Name****_TB/*'sig*
run
quit

Figure 28. A *.con* command file used for simulation

- The command "*cd*" is used to change the current working declarative region in the model. A declarative region is a slash-punctuated path name from the top of the model to a specific declarative region within the VHDL design hierarchy. If no declarative region is specified, the current working declarative region, i.e., the home region '/', is the new declarative region.

- The command "*open*" is used to specify an output device, e.g., a file, where the results are put.

- Using the command "*logtime -e*" enables the Simulator to display the current simulation time and time units in the output file whenever a monitor fires at a new simulated time.

- The command "*echo $cwr*" directed towards the output file, is used to echo the current working region in the output file.

- The command "*set BASE BINARY*" sets the Simulator shell variable BASE to BINARY so that all the results are displayed as binary outputs.

- The command "*monitor*" creates monitors to extract results from the simulation. A monitor can detect and specify a change on a signal or the execution of a line of source code, for example. The switch "-n" followed by a *monitor_name* (e.g. smon), names the created monitor. The Simulator appends small integers to name to give each monitor created a unique name. The switch "-o" followed by an output file_name is used to redirect output from the created monitors to the specified output file. When the variable "*active*" is used in the command, the monitor fires when the specified signal is active, whereas when the variable "*event*" is used, the monitor fires when an event occurs on the specified signal. The term ***Model_Entity_Name_TB/*'sig*** specifies that all the signals in the top level test bench region will be checked by the monitor.

- The command "*run*" is used to start the execution of the VHDL simulation.

- The command "*quit*" causes the VHDL Simulator to exit.

A typical test bench created by PTG algorithm for simulation purposes is as shown in Figure 29.

```
use work.VHDLCAD.all, work.all;

entity Model_Entity_Name_TB is
end Model_Entity_Name_TB;

architecture BEHAVIOR of Model_Entity_Name_TB is
signal A, B: BIT_VECTOR(7 downto 0);
signal C, D, E: BIT;

component Model_Entity_Name_A
port(A: out BIT_VECTOR(7 downto 0); C: in BIT; D: in BIT; B: in BIT_VECTOR(7 downto
0); E: in BIT);
end component;

for all: Model_Entity_Name_A use entity work.Model_Entity_Name(Behavior);

begin

R1: Model_Entity_Name_A
  port map(A, B, C, D, E);

process
  begin

  C <= transport '1' after 0 ns;
  C <= transport '0' after 1 ns;
  C <= transport '1' after 4 ns;
  B <= transport '0' after 0 ns;
  B <= transport '0' after 1 ns;
  B <= transport "11110000" after 0 ns;
  B <= transport "11110000" after 1 ns;
  E <= transport '1' after 0 ns;
  E <= transport '1' after 1 ns;
  wait;

end process;
end BEHAVIOR;
```

Figure 29. An Example Test Bench created by PTG for simulation

From the test bench shown in Figure 29 it is clear that the simulation time specified by the PTG algorithm is 4 ns. Let C be a sensitive signal. The constant values are assigned to all the input signals at 0 or 1 ns but the transition in value, i.e., an event, is created at the 4th ns. For example, in the test bench of Figure 29, an event rise or 'R' is created on the input signal C by assigning it a value '0' at 1st ns and a value '1' at 4th ns. All the other signals have no change in their value at 4th ns.

The test bench created by the PTG algorithm is then used for simulation to get the output response. The command to start simulation, as used by PTG, is:

% vhdlsim -i Model_Name.con Model_Name_TB

The "-i" option is used to include the .con command file and to simulate the model accordingly. The simulation results generated are output in the file *Model_Name.out*. The simulation output file generated is parsed and the responses corresponding to the output signals are stored. The typical simulation results as produced by the Synopsys System Simulator are as shown in Figure 30.

```
/
0 NS
  SMON4:   ACTIVE /Model_Entity_Name_TB/E (value = '1')
  SMON9:    EVENT /Model_Entity_Name_TB/E (value = '1')
  SMON1:   ACTIVE /Model_Entity_Name_TB/B (value = B"11110000")
  SMON6:    EVENT /Model_Entity_Name_TB/B (value = B"11110000")
  SMON3:   ACTIVE /Model_Entity_Name_TB/D (value = '0')
  SMON2:   ACTIVE /Model_Entity_Name_TB/C (value = '1')
  SMON7:    EVENT /Model_Entity_Name_TB/C (value = '1')
   SMON:  ACTIVE /Model_Entity_Name_TB/A (value = B"00000000")
1 NS
  SMON4:   ACTIVE /Model_Entity_Name_TB/E (value = '1')
  SMON1:   ACTIVE /Model_Entity_Name_TB/B (value = B"11110000")
  SMON3:   ACTIVE /Model_Entity_Name_TB/D (value = '0')
  SMON2:   ACTIVE /Model_Entity_Name_TB/C (value = '0')
  SMON7:    EVENT /Model_Entity_Name_TB/C (value = '0')
4 NS
  SMON2:   ACTIVE /Model_Entity_Name_TB/C (value = '1')
  SMON7:    EVENT /Model_Entity_Name_TB/C (value = '1')
   SMON:  ACTIVE /Model_Entity_Name_TB/A (value = B"00000000")
```

Figure 30. The typical simulation results given by Synopsys Simulator

In Figure 30, the symbol '/' in the first line is displayed by using "echo $cwr" command, as shown in the *.con* file. It specifies that the current working region is HOME. The simulation timings 0 ns, 1 ns and 4 ns are displayed by using command "logtime -e" as discussed earlier. The monitor name "SMON" is as specified in the .con file of Figure 30. The variables ACTIVE or EVENT identify if a signal is active or there is an event on it, respectively. The term ***Model_Entity_Name_TB/SIGNAL_NAME*** specifies that the signal

monitored is in the test bench entity ***Model_Entity_Name_TB***. The value in the parenthesis at the end, is the output for the signal specified.

The simulation results, as shown in Figure 30, are parsed and the values assigned to the signals at the 1st ns and the 4th ns are stored and put in a data structure ***test_results***, to be described in appendix A.

This completes the activation of the first sensitive input signal. Similarly, other sensitive signals are selected one by one, events are created, other input signal conditions are found, test benches are created and simulation is performed to get the results. The simulation output files are parsed to get the output responses The whole process is continued till all the sensitive ports of the VHDL process have been activated once.

As already mentioned, the main criterion for process test generation used is that every assignment node of the CFG is executed at least once. After all the sensitive ports have been activated, the PTG algorithm checks to see if all the signal or variable assignment statements have been executed or not. 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. A suitable event is created on the first sensitive input signal encountered along the path. The other input signal values are determined along this path, put in a test bench and the model is simulated again. The simulation output file is parsed and the responses are stored as before. This continues till all assignment statements are executed at least once. This completes the activation part of the PTG algorithm.

5.3.4 The Generation of the Process Test Data File

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 HBTG 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 explained in chapter 3. 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, in accordance with the test data file format.

The symbolic values like 'R', 'F' and 'P' are generated by the software on its own but the symbols for the BIT_VECTOR signals are assigned interactively with the help of the user. A symbol 'P' is assigned when it is found that despite a change in the sensitive signal value from its value at 1 ns to its value at 4 ns, a BIT_VECTOR output signal shows no change in its value. The other symbols used for the BIT_VECTOR signals are D1, D2, D3 and D4 which are assigned by user-interaction. This restriction is because of the fact that at present only these four symbols are used by HBTG. However, PTG can accommodate a wide range of 'D' symbols.

After the mapping of the actual values to specific symbols is done, the PTG algorithm writes the symbolic test sets in a process test data file, like the one shown in chapter 3, and the process test generation is completed. The two time-frame test sets are generated from the single time-frame tests by simply assigning a constant stable signal value to signals, e.g., if a signal has a transition 'R' in a single time frame test set, it will have the same

symbol in the first time-frame of the two time frame test set but will have a value '1' in the second time frame of the same test set, and vice versa for the symbol 'F'. The BIT_VECTOR signals are assigned the same symbol in both the time-frames of the two time-frame test set as the symbol in the single time-frame test set.

The flow diagram describing the PTG algorithm is shown in Figure 31.

5.4 Process Test Generation Examples

5.4.1 Process Test Generation for a Register Model "REG"

In this section, we discuss the generation of process test sets for the process REG introduced in chapter 4. The source listing of the VHDL behavioral model for the process REG is shown in Figure 19 in chapter 4. When the signal CLR is '1', the output D_OUT of the register gets cleared. When CLR is '0' and LD is '1', and there is an event FALL (i.e. 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 shown in Figure 32 below.

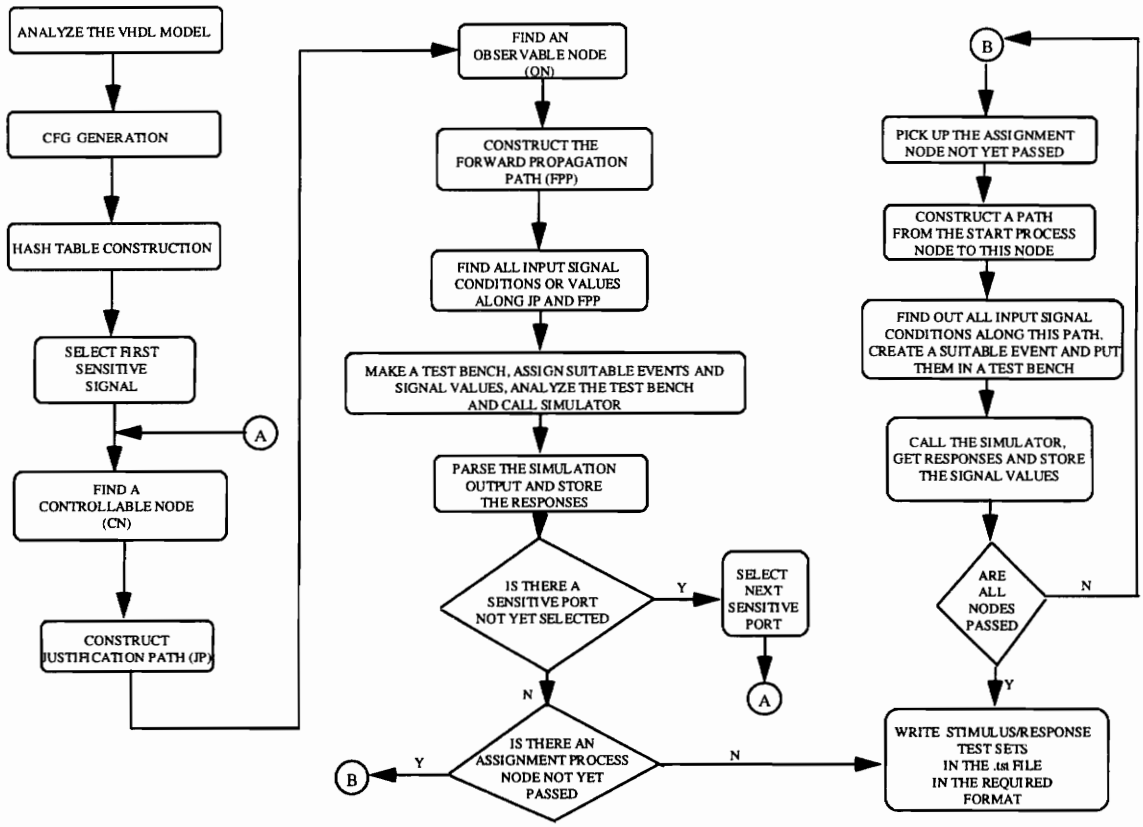


Figure 31. The Flow-Chart of PTG Algorithm

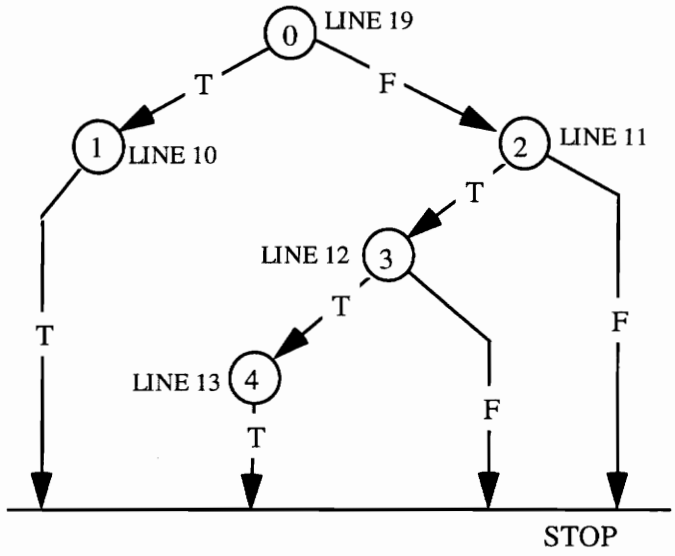


Figure 32. The CFG for process REG

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 process node 2 is selected as the controllable node (CN), as shown in the hash table in Figure 21 in chapter 4. The condition specified in the selected CN is:

(CLK'EVENT and CLK = '0'),

as shown in VHDL source for REG in Figure 19. This results in assigning an event fall, i.e., 'F', on CLK.

In order to justify the assigning of event 'F' on CLK, a justification path is constructed. The CFG_NODE 2 will be passed only if the condition on process node 0 is false. The PTG thus constructs a JP consisting 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 , assigns a value '0' to CLR.

In order to observe a response corresponding to the event on CLK at an output, the event has to be propagated forward. The only output signal in the model is D_OUT. From the hash table in Figure 21, we see that the hash of key for the signal D_OUT is 9. At this table index, two CFG_NODEs, node number 1 and node number 4, are linked in a link list. The find_observable_node algorithm picks up node 4 as an observable node (ON) because node number 1 is above the selected CN in the CFG while node number 4 is obviously greater than the node number 2 of the CN. A forward propagation path (FPP) is constructed from the controllable node 2 to the observable node 4 consisting of node numbers 2, 3 and 4, by the Shortest_Path Algorithm.

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 to the signal and also a symbolic value to it. Let the symbolic value assigned to D_IN by the user be 'D2'. All the input signal values are now known. These values are put in a test bench, the test bench is analyzed and the model is simulated to obtain the output signal values. A test bench produced by PTG for simulation with event on signal CLK, is shown in Figure 33.

```
use work.VHDLCAD.all, work.all;

entity Reg_TB is
end Reg_TB;

architecture BEHAVIOR of Reg_TB is
signal d_out, d_in: BIT_VECTOR(7 downto 0);
signal clr, ld, clk: BIT;

component reg_A
port(d_out: out BIT_VECTOR(7 downto 0); clr: in BIT; ld: in BIT; d_in: in BIT_VECTOR(7
downto 0); clk: in BIT);
end component;
for all: reg_A use entity work.reg(behavior);
begin
R1: reg_A
   port map(D_OUT, CLR, LD, D_IN, CLK);
process
   begin

   clk <= transport '1' after 0 ns;
   clk <= transport '1' after 1 ns;
   clk <= transport '0' after 4 ns;
   clr <= transport '0' after 0 ns;
   clr <= transport '0' after 1 ns;
   ld <= transport '1' after 0 ns;
   ld <= transport '1' after 1 ns;
   d_in <= transport "00001111" after 0 ns;
   d_in <= transport "00001111" after 1 ns;
   wait;

end process;
end BEHAVIOR;
```

Figure 33. A Test Bench generated by PTG for event 'F' on CLK in process REG

The simulation results given by the Synopsys Simulator are shown in Figure 34.

```
/
0 NS
  SMON1:   ACTIVE /REG_TB/D_IN (value = B"00001111")
  SMON6:    EVENT /REG_TB/D_IN (value = B"00001111")
  SMON3:   ACTIVE /REG_TB/LD (value = '1')
  SMON8:    EVENT /REG_TB/LD (value = '1')
  SMON2:   ACTIVE /REG_TB/CLR (value = '0')
  SMON4:   ACTIVE /REG_TB/CLK (value = '1')
  SMON9:    EVENT /REG_TB/CLK (value = '1')
1 NS
  SMON1:   ACTIVE /REG_TB/D_IN (value = B"00001111")
  SMON3:   ACTIVE /REG_TB/LD (value = '1')
  SMON2:   ACTIVE /REG_TB/CLR (value = '0')
  SMON4:   ACTIVE /REG_TB/CLK (value = '1')
4 NS
  SMON4:   ACTIVE /REG_TB/CLK (value = '0')
  SMON9:    EVENT /REG_TB/CLK (value = '0')
   SMON:  ACTIVE /REG_TB/D_OUT (value = B"00001111")
  SMON5:    EVENT /REG_TB/D_OUT (value = B"00001111")
```

Figure 34. Simulation Output File for event 'F' on CLK in process REG

From the simulation output it is obvious that the output D_OUT gets the same bit_vector value as on the input D_IN at the 4th ns. So this is also assigned the symbolic value 'D2'. The final stimulus/response pair generated is as shown below:

D_OUT	CLR	LD	D_IN	CLK
D2	0	1	D2	F

The above event/action pair clearly tests an operation of the process REG. It states that when CLR is '0' and LD is '1', then an event fall, i.e., 'F' on the signal CLK will result in the value 'D2' at the input D_IN getting assigned to the output D_OUT. This completes the propagation of event on the signal CLK. The symbolic values are written to the test data file in the *portorder*. The ***portorder*** is obtained from the PMG database of the Modeler's Assistant. The *portorder* of process REG is *D_OUT - CLR - LD - D_IN - CLK*.

The PTG next selects the other sensitive signal CLR and creates an event rise, i.e., 'R', on it. The process node number 0, i.e., the start process node is chosen as the controllable node with the help of the hash table shown in Figure 21. There is no justification path since node 0 is the first CFG node.

In order to propagate this event to the output, the process node number 1 is selected as the observable node. This node is selected in this case because it is below the selected CN, i.e., node number 0, in the CFG. The forward propagation path (FPP) constructed consists of nodes 0 and 1. In this case, the values on other input signals are "don't cares" as there are no input signals associated with the nodes along the FPP. Let the values on the other inputs be '0' on LD, 'D2' on D_IN and '1' on CLK.

Again a test bench is formed, analyzed and simulator is invoked. A test bench formed in this case is shown in Figure 35.

```
use work.VHDLCAD.all, work.all;

entity Reg_TB is
end Reg_TB;

architecture BEHAVIOR of Reg_TB is
signal d_out, d_in: BIT_VECTOR(7 downto 0);
signal clr, ld, clk: BIT;


component reg_A
port(d_out: out BIT_VECTOR(7 downto 0); clr: in BIT; ld: in BIT; d_in: in BIT_VECTOR(7
downto 0); clk: in BIT);
end component;

for all: reg_A use entity work.reg(behavior);

begin

R1: reg_A
   port map(D_OUT, CLR, LD, D_IN, CLK);

process
  begin

  clr <= transport '1' after 0 ns;
  clr <= transport '0' after 1 ns;
  clr <= transport '1' after 4 ns;
  ld <= transport '0' after 0 ns;
  ld <= transport '0' after 1 ns;
  d_in <= transport "11110000" after 0 ns;
  d_in <= transport "11110000" after 1 ns;
  clk <= transport '1' after 0 ns;
  clk <= transport '1' after 1 ns;
  wait;
end process;
end BEHAVIOR;
```

Figure 35. A Test Bench generated by PTG for event 'R' on CLR in process REG

The simulation of the VHDL behavioral model of the process REG with the above test bench, gives the output results as shown in Figure 36.

```
/
0 NS
  SMON4:   ACTIVE /REG_TB/CLK (value = '1')
  SMON9:    EVENT /REG_TB/CLK (value = '1')
  SMON1:   ACTIVE /REG_TB/D_IN (value = B"11110000")
  SMON6:    EVENT /REG_TB/D_IN (value = B"11110000")
  SMON3:   ACTIVE /REG_TB/LD (value = '0')
  SMON2:   ACTIVE /REG_TB/CLR (value = '1')
  SMON7:    EVENT /REG_TB/CLR (value = '1')
   SMON:   ACTIVE /REG_TB/D_OUT (value = B"00000000")
1 NS
  SMON4:   ACTIVE /REG_TB/CLK (value = '1')
  SMON1:   ACTIVE /REG_TB/D_IN (value = B"11110000")
  SMON3:   ACTIVE /REG_TB/LD (value = '0')
  SMON2:   ACTIVE /REG_TB/CLR (value = '0')
  SMON7:    EVENT /REG_TB/CLR (value = '0')
4 NS
  SMON2:   ACTIVE /REG_TB/CLR (value = '1')
  SMON7:    EVENT /REG_TB/CLR (value = '1')
   SMON:   ACTIVE /REG_TB/D_OUT (value = B"00000000")
```

Figure 36. Simulation Results for event 'R' on CLR in process REG

From the results of Figure 36, it is observed that the signal D_OUT is cleared at the 4th ns, when an event rise, i.e., 'R', occurs on CLR. The value "00000000" assigned to D_OUT is given a symbolic value 'D1' and the test generation is completed. The second

event/action test set inserted into the test data file corresponding to the event on signal CLR is as follows:

D_OUT	CLR	LD	D_IN	CLK
D1	R	0	D2	1

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 37.

---

portorder: D_OUT CLR LD D_IN CLK

1
D2 0 1 D2 F
1
D1 R 0 D2 1
2
D2 0 1 D2 F
D2 0 1 D2 0
2
D1 R 0 D2 1
D1 1 0 D2 1

Figure 37. The Process Test Data File for process REG

5.4.2 Process test generation for the sequential primitive LATCH

The VHDL source listing for the process LATCH is shown in Figure 38 and the CFG for the process LATCH is shown in Figure 39. When CLK is '0', the data value at the input DATA is assigned to the output OP, otherwise the output retains its last value.

```
Levels  LINE # |----+----1----+----2----+----3----+----4----+----5----+----6-
            1 |use WORK.VHDLCAD.all, WORK.USER_TYPES.all;
            2 |entity LATCH is
1           3 |  port (OP: inout  BIT_VECTOR(0 to 7); CLK: in  BIT; DATA: in  BIT_VECTOR(0
                    to 7));
1           4 |end LATCH;
            5 |architecture BEHAVIOR of LATCH is
1           6 |begin
1           7 |   LATCH_1: process (DATA, CLK)
2           8 |   begin
2           9 |      if CLK = '0' then
2 1        10 |            OP <=  DATA;
2 1        11 |      else
2 1        12 |            OP <= OP;
2 1        13 |      end if;
2          14 |   end process LATCH_1;
1          15 |end BEHAVIOR;
```

Figure 38. VHDL source for the process LATCH

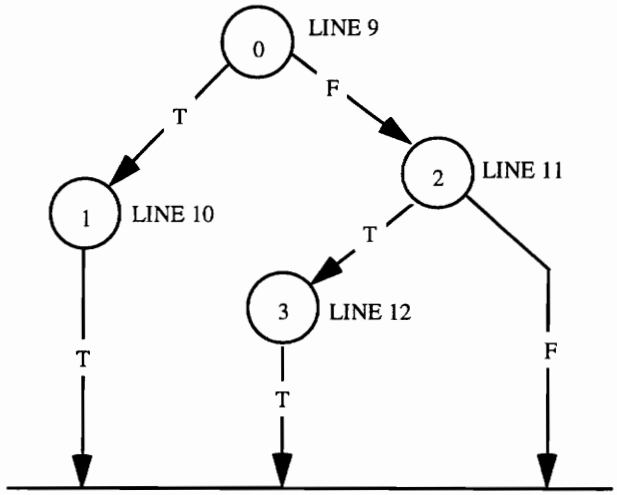


Figure 39. A CFG for the process LATCH

The PTG algorithm picks up the first sensitive signal, i.e., signal DATA, from the sensitivity list of the process LATCH. The only CFG_NODE associated with the signal DATA is the node number 1, corresponding to the statement number 10. Thus, the node number 1 is selected as the controllable node (CN). A justification path (JP) is constructed from the start node (SN) to this node. It consists of the nodes 0 and 1, at lines 9 and 10, respectively. The user is prompted to assign an initial and a new actual values, as well as, an initial and a new symbolic values, to the sensitive signal DATA, in order to represent a transition. Let the initial and the new symbolic values assigned to the signal be 'D2' and 'D3'. The justification process along the JP assigns a value '0' to the signal CLK. The observable node selected in this case is the same as the CN, i.e., the

node number '1'. Thus, the forward propagation path (FPP) consists of just one node, the node number '1'. In this case, the propagation along FPP assigns no new values to any input signals. The input signal values are put in two test benches, corresponding to values 'D2' and 'D3' on signal DATA, respectively. The test benches are then analyzed and simulations are performed. The two test sets created in this case, corresponding to the symbolic value 'D2' and 'D3' assigned to the signal DATA, are:

OP	DATA	CLK
D2	D2	0

OP	DATA	CLK
D3	D3	0

The PTG algorithm next selects the sensitive signal CLK and chooses the CFG_NODE 0 as the CN. An event fall (i.e., 'F') is created on the signal CLK. The node number 0 is the first node of the CFG, so no JP is constructed in this case. In order to observe a response to the stimulus 'F', PTG picks up node number 1 as the ON, and constructs an FPP consisting of the nodes 0 and 1. The user is prompted to assign a data value to the input signal DATA. A symbolic value, say 'D2', is assigned to DATA. The input signal values are again put in a test bench and simulation is performed. In this case, the output signal OP is assigned the same value as DATA, and so OP also gets the symbolic value 'D2'. The stimulus/response pair produced is as follows:

OP	DATA	CLK
D2	D2	F

The activation of both the sensitive signals is now completed. In the next step, the PTG algorithm checks to see if all the assignment nodes have been executed or not. In order to execute the node number 3, not executed as yet, it selects this node as an observable node (ON) and constructs a path from the start node (SN) to this ON. The first sensitive signal found along this path is CLK. It is also found that in order to reach the ON along this path, the condition on the node number '0', associated with signal CLK, is FALSE. Thus, an event rise, i.e., 'R', is created on the signal CLK. The user is once again prompted to assign a value to the other input signal, i.e., signal DATA. Let the symbolic value assigned to DATA be 'D2'. The values are again put in a test bench and the model is simulated. In this case, the value at the output OP stays at its last value, i.e., the value of OP at 4th ns is the same as its value at 1st ns, as found from the simulation results. Thus, a symbol 'P' is inserted in the test data file for this test, in accordance with the process test data file format. The test set generated is as shown below:

OP	DATA	CLK
P	D2	R

All the assignment nodes are now passed and the process test generation is completed. The user is finally prompted to assign an initial value to the output signal if the model is a sequential primitive. In this case, a symbolic value, say 'D2', is assigned as the initial value to the sequential output port OP. This value is inserted in the initialization frame of the test data file and the final test file generated is as shown in Figure 40.

```
portorder: OP DATA CLK
D2
1
D2 D2 0
1
D3 D3 0
1
D2 D2 F
1
P D2 R
2
D2 D2 0
D2 D2 0
2
D3 D3 0
D3 D3 0
2
D2 D2 F
D2 D2 0
2
P D2 R
P D2 1
```

Figure 40. The Process Test Data File for process LATCH

Chapter 6. The Test Bench Generation

The verification of the correctness of a circuit is incomplete till a stimuli data is applied to it and till a response data is collected from it [28]. This is done by simulating the circuit by giving it the stimuli data in a particular format. In the Standard VHDL 1076 Support Environment, a Test Bench forms the top-level design unit [28] that provides the input stimuli data in a suitable format that can be used for simulation.

6.1 Purpose of the Test Bench

In the section on Hierarchical Behavioral Test Generation in chapter 3, it was discussed that coverage feature of the Synopsys VHDL System Simulator has been used to evaluate the effectiveness of the final test sequence generated by HBTG. The final test sequence produced by HBTG is converted into a test bench before it can be used to perform

simulation and get the coverage results. 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, within the Standard VHDL 1076 Support Environment.

6.2 Test Bench Generator (TBG) program

Previously, the test sequence generated by HBTG was converted into a test bench manually. A time-period of 5 ns was chosen for each time-frame of the final test sequence produced by HBTG [3]. The various primary inputs of the VHDL behavioral model were assigned values at times 1 ns, 4 ns, 6 ns, 9 ns, etc., i.e., the signal assignments were made at times (5*t + 1) ns and (5*t + 4) ns where t ε {0, 1 2, 3 .....} is the lexical order of the time frame starting from 0.

The method of manually converting the final test sequence into a test bench is very time-consuming and inefficient. The test bench generation involves assigning of data values to symbols like "D_i's" and "C_i's", and also assigning proper values to events like 'R' and 'F' within the 5 ns time period of a time frame, so as to represent a transition in value. The technique of manually finding the primary inputs through the almost unreadable test sequences and assigning proper signal values to them is very laborious. Also, in the case of models with a feedback loop, it is not clear which signal, whether the source port or the destination port of the feedback signal, is to be included in the test bench. The Test Bench Generator (TBG) program removes these difficulties and enables the designer to get a test bench directly from the test sequence.

The TBG parses the HBTG test sequence and the VHDL source file and also extracts the useful information from the PMG database of the Modeler's Assistant. The primary inputs are found from the port declaration in the VHDL model source file dumped by Modeler's Assistant. The test sequence columns corresponding to these signals are selected and the values in the different rows or time frames are stored. In the case of the feedback signals, the source port and the destination port of the signal are found out from the PMG database and the port of mode "in" is picked up as the primary input. The output signal names are also noted down from the port declaration in the VHDL source and are stored. Thus, all the information required for the test bench entity, architecture and signal declarations, and the component instantiation, is obtained.

The next step is the conversion of the symbolic values stored to the actual signal values. 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 and that all signal assignments and transitions within a time-frame take delta delay time.

In order to convert the symbols like D_i's (or C_i's) for multi-bit data values to specific signal values, the user is prompted to assign actual values to these symbols. 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 fall '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 (e.g., '0' or '1'), TBG assigns the same value to it in both the 1st ns and the 4th ns of the

time-frame period. All the signal assignments and the declarations are put in a file, appropriately named, and the test bench generation is completed. This test-bench file can then be used for the simulation of the model. The procedures to compile and execute the "tbg" program are explained in appendix A. Figure 41 shows a form of test bench generated by TBG. The bold italics represent the portion of the test bench that differs from one model to another.

---

*use work.**VHDLCAD**.all, work.all;*

*entity **model_entity_name**_TEST_BENCH is*
*end **model_entity_name**_TEST_BENCH;*

*architecture BEHAVIOR of **model_entity_name**_TEST_BENCH is*

-- Signal declarations
*signal **sig_1, sig_2, ..., sig_i**: BIT;*
*signal **sig_A, sig_B, ..., sig_X**: BIT_VECTOR(**integer range <>**);*

-- Component Declaration
*component **model_entity_name**_A*
  ***port declaration**;*
end component;

*for all: **model_entity_name**_A use entity work.**model_entity_name(model_arch_name)**;*

begin

component instantiation

process

begin

input test vectors (transport signal assignment statements);

end process;

end BEHAVIOR;

Figure 41. A typical Test Bench generated by TBG

- It is assumed that a package VHDLCAD exists that consists of some useful type declarations and functions to be used for modeling.

- The ***model_entity_name*** is the device name for the model. It is the same name as the entity name in the VHDL source for the model generated by Modeler's Assistant or give by the user.

- The component declaration declares the interface to the source model. The ***model_entity_name_A*** is the component name, as given by TBG. The port declaration part is a copy of the port declaration in the entity body of the VHDL source for the model.

- The use statement associates binding information with the component labels representing specific instances of a given component. The ***model_arch_name*** is the architectural body name of the source model, as given by Modeler's Assistant or by the user.

- The component instantiation block contains the port map statements required to interconnect the model with the test bench.

- The input test vectors consist of VHDL transport signal assignment statements where the test values are assigned to the input ports. These test values are obtained from the HBTG test sequence and are mapped to the actual values by TBG with user interaction.

6.3 An Example

The Test Bench Generation by TBG program will be clear if we consider an example. The example considered is of the 8-bit LATCH circuit discussed in chapter 3. The PMG for the circuit is shown in Figure 10 and the VHDL source is shown in Figure 11. The final results created by HBTG for the circuit are shown in the Figure below.

Enter name of the unit:
Unit name is latch8
No. of signals = 7

Sensitive path 0:
sp[0][0]=13(NDS2)
sp[0][1]=11(ENBLD)
sp[0][2]=7(ENBLD)
sp[0][3]=5(DO)

Sensitive path 1:
sp[1][0]=14(DS1)
sp[1][1]=11(ENBLD)
sp[1][2]=7(ENBLD)
sp[1][3]=5(DO)

Sensitive path 2:
sp[2][0]=19(CLK)
sp[2][1]=17(OP)
sp[2][2]=8(REG)
sp[2][3]=5(DO)

No. of Spath = 3

A complete test sequence for model latch8:

frame	*sig(0)*	*sig(1)*	*sig(2)*	*sig(3)*	*sig(4)*	*sig(5)*	*sig(6)*
0	*X*	*X*	*X*	*X*	*D2*	*F*	*D2*
1	*X*	*X*	*X*	*X*	*D2*	*0*	*D2*
2	*D2*	*R*	*F*	*1*	*D2*	*0*	*D2*
3	*D4*	*F*	*0*	*F*	*D2*	*0*	*D2*
5	*D2*	*R*	*F*	*1*	*D2*	*0*	*D2*
4	*D2*	*1*	*0*	*1*	*D2*	*R*	*D3*

sig(0) = DO
sig(1) = ENBLD
sig(2) = NDS2
sig(3) = DS1
sig(4) = OP
sig(5) = CLK
sig(6) = DATA

PortAct(7) = 2 PortAct(8) = 1
PortAct(13) = 1 PortAct(14) = 2
PortAct(19) = 1

Figure 42. A Test Sequence generated by HBTG for 8-bit latch model

The results show the sensitive paths through the model, the test sequence and the signal names. The "*PortAct(i)*" at the end represents the number of times a signal has been activated during the test generation.

The TBG parses the above results and the VHDL source file shown in Figure 11 to create the test bench as shown in the Figure 43 below.

```
use work.VHDLCAD.all, work.all;

entity LTEST1_TEST_BENCH is
end LTEST1_TEST_BENCH;

architecture BEHAVIOR of LTEST1_TEST_BENCH is
signal DO: MVL_VECTOR(0 to 7);
signal NDS2, DS1, CLK: BIT;
signal DATA: BIT_VECTOR(0 to 7);

component LTEST1_A
port( DO: out MVL_VECTOR(0 to 7); NDS2: in BIT; DS1: in BIT; CLK: in BIT; DATA: in
BIT_VECTOR(0 to 7));
end component;

for all: LTEST1_A use entity work.LTEST1(BEHAVIORAL);

begin

R1: LTEST1_A
  port map(DO, NDS2, DS1, CLK, DATA);

 process
  begin


  NDS2 <= transport '0' after 1 ns;
  DS1 <= transport '0' after 1 ns;
  CLK <= transport '1' after 1 ns;
  DATA <= transport "10010011" after 1 ns;
```

```
NDS2 <= transport '0' after 4 ns;
DS1 <= transport '0' after 4 ns;
CLK <= transport '0' after 4 ns;
DATA <= transport "10010011" after 4 ns;

NDS2 <= transport '0' after 6 ns;
DS1 <= transport '0' after 6 ns;
CLK <= transport '0' after 6 ns;
DATA <= transport "10010011" after 6 ns;

NDS2 <= transport '0' after 9 ns;
DS1 <= transport '0' after 9 ns;
CLK <= transport '0' after 9 ns;
DATA <= transport "10010011" after 9 ns;

NDS2 <= transport '0' after 11 ns;
DS1 <= transport '0' after 11 ns;
CLK <= transport '1' after 11 ns;
DATA <= transport "10000011" after 11 ns;

NDS2 <= transport '0' after 14 ns;
DS1 <= transport '0' after 14 ns;
CLK <= transport '0' after 14 ns;
DATA <= transport "10000011" after 14 ns;

NDS2 <= transport '1' after 16 ns;
DS1 <= transport '1' after 16 ns;
CLK <= transport '0' after 16 ns;
DATA <= transport "10000011" after 16 ns;

NDS2 <= transport '0' after 19 ns;
DS1 <= transport '1' after 19 ns;
CLK <= transport '0' after 19 ns;
DATA <= transport "10000011" after 19 ns;

NDS2 <= transport '0' after 21 ns;
DS1 <= transport '1' after 21 ns;
CLK <= transport '0' after 21 ns;
DATA <= transport "10000011" after 21 ns;

NDS2 <= transport '0' after 24 ns;
DS1 <= transport '0' after 24 ns;
CLK <= transport '0' after 24 ns;
DATA <= transport "10000011" after 24 ns;
```

```
    NDS2 <= transport '0' after 26 ns;
    DS1 <= transport '0' after 26 ns;
    CLK <= transport '0' after 26 ns;
    DATA <= transport "10000011" after 26 ns;

    NDS2 <= transport '0' after 29 ns;
    DS1 <= transport '1' after 29 ns;
    CLK <= transport '0' after 29 ns;
    DATA <= transport "10000011" after 29 ns;

    NDS2 <= transport '0' after 31 ns;
    DS1 <= transport '1' after 31 ns;
    CLK <= transport '0' after 31 ns;
    DATA <= transport "10000011" after 31 ns;

    NDS2 <= transport '0' after 34 ns;
    DS1 <= transport '1' after 34 ns;
    CLK <= transport '1' after 34 ns;
    DATA <= transport "10000011" after 34 ns;

    wait;

  end process;
end BEHAVIOR;
```

Figure 43. The Test Bench generated by TBG for 8-bit latch circuit

It is obvious from the test sequence results shown in Figure 42 and the test bench shown in Figure 43, that an event 'R' is represented by a transition in value from '0' at 1st ns to '1' at 4th ns of a 4 ns time frame period, e.g., in order to represent an event 'R' on CLK in time frame 5, the signal CLK is assigned a value '0' at 31st ns and a value '1' at 34th ns, as shown in the test bench. Similarly, the event 'F' is represented in the test bench. The values "10010011" and "10000011" are assigned to signal DATA corresponding to the symbols D1 and D2, with the help of user interaction.

Chapter 7. Results

The PTG software has been implemented in Sun-C on a Sun SPARCstation2. It consists of about 6500 lines of source code. The complete system includes a Control Flow Graph (CFG) generator that derives the graph from the VHDL behavioral model of a circuit, and the PTG-algorithm that generates the stimulus/response test sets for the model. The stimulus/response test sets are such that they test all the operations of a single-process VHDL behavioral model. Appendix A gives an explanation on how to use the PTG software.

The process test sets for various models have been generated with the help of the PTG program. The models tested were of reasonable complexity ranging from simple logic circuits like AND, OR, and INVERTER to RTL circuits like multiplexers, registers, counters, etc. Several larger MSI circuits like a controlled counter and a priority encoder

have also been tested. The VHDL behavioral models for thirteen circuits, their Control Flow Graphs and the process test results are shown in Appendix C.

The Test Bench Generator (TBG) has also been discussed in the previous chapter. It is used to convert a test sequence generated by the Hierarchical Behavioral Test Generator (HBTG) into a test bench that can be used for simulation and for the coverage measurement. TBG is also implemented in Sun-C and consists of about 1300 lines of code. Appendix C also shows two test benches created by TBG from the HBTG test sequences.

7.1 Circuit Models Used

A total of fifteen circuits were used for process test generation to provide initial results. Two of the circuits, a Register model "REG" and a "LATCH", have been discussed in the chapter 6. The VHDL source listings, the Control Flow Graphs generated and the process test results for the other thirteen circuit models are given in Appendix C. A brief description of these models is presented in this section.

BUFFER

This circuit is a simple buffer model. It demonstrates how process test results are generated for a model in which a BIT_VECTOR signal is in the sensitivity list and is also in the source part of a signal assignment statement, e.g., signal BUF. Two test sets are generated to represent transition in signal values for the signal BUF.

SERVRQ

The circuit SERVRQ is a simple service request flip-flop circuit which is a part of the Intel 8212 buffered latch system. This circuit has been selected because it demonstrates the ability of the PTG software to handle circuits with three IF-ELSIF-ELSE clauses. Also, it is a sequential circuit and validates the testing of PTG for sequential primitives. In the third test set it is observed that the value of SRQ stays at its last value denoted by the symbol 'P'.

VECTOR

The circuit VECTOR is a priority encoder. It is a part of an interrupt controller used for the Intel 8212 buffered latch system. It converts the highest-priority active interrupt into an interrupt vector. This model is selected because it has five IF-ELSIF-ELSE clauses and it also handles vector signals. The stimulus/response test sets for all the five cases are generated, and so all the conditions are processed. Thus, it demonstrates the handling of vector signals and a number of IF clauses by the PTG software.

CKTA

This is a simple input-ouput circuit. It is used to show the working of the PTG software for models where "/=" condition is used. Also, it demonstrates the execution of two signal assignment statements sequentially, once the condition is determined. It shows that only two test sets are required to handle four signal assignment statements like these.

CKTB

This is a circuit which has been selected to demonstrate the handling of variable and signal assignment statements by the PTG software. All these assignment statements are executed sequentially once the condition is satisfied. Only two test sets are required in this case to execute all the assignment statements.

DECODER

This circuit is a 2_to_4 decoder. It demonstrates how the PTG program generates test sets for models where BIT_VECTOR signals of different ranges are used. The symbolic notation used for one signal is independent from the notation used for the other signal, and also the same notation can be used for the same signal in different test sets. The circuit also shows the handling of the CASE statement and its clauses by the PTG. It is observed that all the clauses of the CASE statement are processed by the PTG.

MUX

The circuit MUX is a 4_TO_1 multiplexer. The circuit also has an enable signal ENB. This model demonstrates the handling of a CASE statement nested inside an IF statement and also an ELSE clause with that. It is found that tests are generated for all the case and the if clauses separately.

DFF

This is a D Flip-Flop model, a sequential circuit with asynchronous set and clear. It again demonstrates the working of the PTG algorithm for cases where more than two IF clauses are used and also for each condition two signal assignment statements are executed sequentially. It is found that in this case only three test sets are needed to cover all signal assignment statements.

ADDER

This is a simple one-bit full adder. This shows that the PTG algorithm also works for a process which just has signal assignment statements and also consists of a source expression with a large number of operators in a signal assignment. Test sets are generated for both a rise 'R' and a fall 'F' on a single-bit sensitive input port that occurs in the source expression of a signal assignment statement.

COUNT

This is an example of a four-bit up/down counter with parallel load and a reset. This model has three degree of IF statement nesting. It also uses certain user defined, one argument functions like INC and DEC. The results were successfully generated for this model which executed all the signal assignment statements and created events on all the sensitive input ports. This demonstrates the handling of user defined functions (only those having one argument) and of IF statement nesting by PTG. The symbol 'CN' represents

the next value in the sequence of control signals, while the symbol 'CP' represents the previous value in the sequence.

JKFF

This is one of the bigger models tested by PTG. It represents a JK Flip-Flop circuit. It employs the use of nested IF statements, a large number of input signals and conditions, and execution of two signal assignment statements for each condition. The stimulus/response test sets were created for this model which successfully tested all the operations of the circuit by executing every signal assignment statement at least once. The symbol 'NP' denotes the "not" of the signal value in the previous time frame during hierarchical behavioral test generation.

CKTC

This circuit consists of a multiplexer, an input register, an output register and control signals. This circuit also uses a variable which is a control variable for the loading or clearing of the output register. The circuit uses the logic operators like "and" and "or" within the CASE signal assignment statements. The model results in a "*deeper*" Control Flow Graph (CFG) and demonstrates that PTG can form test sets even for such large CFGs.

CCNT3

This circuit is a three-bit up/down control counter with a parallel load. The reset, load and up/down count conditions are all generated by a control signal which forms a primary sensitive input port to the circuit. This model results in the construction of a large CFG with a lot of conditions. It demonstrates a good justification and propagation of signal values along the selected paths. The stimulus/response test sets generated test all the operations of the model.

7.2 Summary

The current implementation of the PTG algorithm performed extremely well on the circuits of reasonable complexity, as defined. The test results showed that the PTG algorithm exercises a model thoroughly and generates stimulus/response test sets that test all the operations of a model.

However, the user interaction is essential in order to map the signal values to symbolic notation as required by the HBTG algorithm, to which the software is interfaced. The user should have a complete understanding of the functioning of the model for which the tests are being generated.

The PTG software considerably increases the speed and the ability of the Automatic Test Generation system to generate a test bench that tests different functions of a model.

Chapter 8. Analysis and Suggestions

While the Process Test Generation software is a powerful tool that creates stimulus/response test sets for all the operations of a VHDL model, its present implementation contains certain inadequacies. This chapter reviews the important weaknesses, provides suggestions for improvements and discusses areas for future work.

8.1 Expansion of the VHDL Subset

The experimental results indicate that the process test generation software has the potential to be used for generating tests for larger circuits. The software considers only a restricted VHDL subset which makes it difficult to construct larger models. The VHDL subset used can be expanded to include other logical and arithmetic operations so that tests for larger models can be generated more efficiently. The bus resolution functions can

be used to handle the assigning of the data values by two drivers to the same signal. This can also remove the initialization problem in the case of sequential circuits. In addition to this, the ability of the software to handle user defined functions can also be increased. At present, only one-argument user defined functions are taken care of; the data structures used can be slightly modified to include more functions.

8.2 Additional Data Types

The PTG software can handle only the BIT or BIT_VECTOR data types with ease. A table of other data types, possible ranges and other signal constraints can be built up so that the domain of test generation can be increased.

8.3 Possibilities of Delay Fault Testing

The present version of process test generation software does not handle any timing constraints. However, it is possible to incorporate timing models and construct Control Flow Graphs for these models based on the same principles as discussed in this thesis. The Control Flow Graph constructed for such a VHDL behavioral model can be analyzed to find the functional paths. By incorporating the timing constraints in the CFG, these functional paths can be tested for path delay faults at the behavioral level.

8.4 Expansion into a Complete Test Generation System

The Process Test Generation (PTG) software is used to generate stimulus/response test sets for single-process VHDL behavioral models. It is also possible to expand the system into a complete test generation system that can generate tests for multi-process models. A provision has been made in the data structures used for Control Flow Graph generation that can construct graphs for a model having at most ten processes. This can be further expanded to handle even larger number of processes. Using the same principles, as introduced in this thesis, it is possible to generate tests for a multi-process model that can directly take care of executing every statement at least once. A significant work will be to expand the scope of this system to include multiple entities in test generation.

8.5 Increasing Test Effectiveness

One of the major problems in the process test generation and its interface to HBTG software is the handling of the symbolic notation for data signal values. The HBTG has defined a symbolic notation that is followed in the entire test generation system. In the case of PTG, the user has to be prompted time and again to assign actual and symbolic signal values to the multi-bit signals in order to perform simulation and get the responses for specific events. The test vectors assigned by the user during process test generation are the key to an effective test generation system. In order to get a good gate-level fault coverage, it is inevitable that there be a control over the signal values that are assigned to the system. The test vectors assigned should be such that they activate as many paths and provide a good coverage at the gate level. One suggestion is to synthesize the VHDL

behavioral model at the gate level and try to optimize the method of selection of test vectors. It is easier to synthesize a single-process VHDL behavioral model and try to find the suitable test vectors rather than synthesize a bigger multi-process model. This method of test generation should result in a high gate level fault coverage for the entire system.

Chapter 9. Conclusion

This thesis has presented a systematic process test generation (PTG) software that can generate stimulus/response test sets for VHDL behavioral models. The VHDL behavioral model is generated interactively using the Modeler's Assistant. The PTG software transforms the VHDL description into a Control Flow Graph (CFG) that gives the data and control flow information within the VHDL model. The Control Flow Graph constructed is then used to generate process test sets for the VHDL model. The PTG software tests all the operations of a single-process VHDL behavioral model.

As discussed in chapter 3, the PTG software forms a part of the Automatic Test Generation System being developed at Virginia Tech. The process test sets generated by the PTG are given as input to HBTG along with a Process Model Graph (PMG), generated by Modeler's Assistant. The HBTG, developed earlier, uses these inputs to construct a test sequence for the entire multi-process VHDL behavioral model represented by the PMG.

Another program, called Test Bench Generator (TBG), has been presented in this thesis which converts the test sequence generated by HBTG into a VHDL Test Bench that can be used for the simulation of the model. Thus, the two software, PTG and TBG, complement the automatic test generation system and considerably speed up the test generation process for VHDL behavioral models.

Bibliography

[1] Tom R. Halfhill, "Intel Launches Rocket in a Socket," *Byte Magazine*, pp. 92-108, May 1993.

[2] IEEE Standard VHDL Language Reference Manual, 1988.

[3] S. Rao, B-Y Pan and J. Armstrong, "Hierarchical Test Generation for VHDL Behavioral Models," *Proc. of EDAC*, pp. 175-182, 1993.

[4] Y. H. Levendel and P. R. Menon, "Test Generation Algorithms for Computer Hardware Description Languages," *IEEE Transactions on Computers*, vol. c-31, pp.577-588, July 1982.

[5] H. D. Hummer, H. Veit, H. Topfer, "Functional Tests for Hardware derived from VHDL Description," *10th Intl. Symposium on CHDLs and their Applications*, pp. 433 445, April 1991.

[6] M. D. O'Neill, D. D. Jani, C. H. Cho and J. R. Armstrong, "BTG: A Behavioral Test Generator," *9th Intl. Symposium on CHDLs and their Applications*, June 1987.

[7] Chang H. Cho, James R. Armstrong, "VHDL Semantics for Behavioral Test Generation," *10th Intl. Symposium on CHDLs and their Applications*, April 1991, pp. 395-412.

[8] A. K. Gupta and J. R. Armstrong, "Functional Fault Modeling and Simulation for VLSI Devices," 22nd Design Automation Conference, pp. 720-726, June 1985.

[9] D. S. Barclay and J. R. Armstrong, "A Heuristic Chip-Level Test Generation Algorithm," 23rd Design Automation Conference, pp. 257-262, June 1986.

[10] Forrest E. Norrod, "An Automatic Test Generation Algorithm for Hardware Description Languages," *26th ACM/IEEE Design Automation Conference*, pp. 429-434, 1989.

[11] P. Wodey, C. Robach, "Using a VHDL description to generate hardware test," *10th Intl. Symposium on CHDLs and their Applications*, April 1991, pp. 413-431.

[12] J. P. Roth, "Diagnosis of Automata Failures: A Calculus and a Method," *IBM Journal of Research and Development*, vol. 10, pp. 278-291, July 1966.

[13] C. W. Cha, W. E. Donath and F. Ozguner, "9-V Algorithm for Test Pattern Generation of Combinational Digital Circuits," IEEE Transactions on Computers, vol. c-27, no. 3, pp. 193-200, March 1978.

[14] P. Goel, "An Implicit Enumeration Algorithm to generate Tests in Combinational Logic Circuits," *IEEE Transactions on Computers*, vol. c-30, March 1981.

[15] H. Fujiwara and T. Shimono, "On the Acceleration of Test Generation Algorithms," *IEEE Transactions on Computers*, pp. 1137-1144, December 1983.

[16] J. R. Armstrong and D. Brunette, "Automated Assists to the Behavioral Modeling Process," Proceedings of the First International Workshop on Rapid System Prototyping, Research Triangle Park, NC, June 1990.

[17] R. P. Kunda, P. Narain, J. A. Abraham, B. D. Rathi, "Speed up of Test Generation using High-Level Primitives," *27th ACM/IEEE Design Automation Conference*, pp. 594-599, June 1990.

[18] Tonysheng Lin, Stephen Y. H. Su, "The S-Algorithm: A Promising Solution for Systematic Functional Test Generation," *IEEE Transaction of Computer-Aided Design*, pp. 250-263, Vol. CAD-4. No. 3, July 1985.

[19] M. Kawai, H. Shibano, S. Funatsu, S. Kato, T. Kurobe, K. Ookawa and T. Sasaki, "A High Level Test Pattern Generation Algorithm," *IEEE International Test Conference*, pp. 346-352, 1983.

[20] S. B. Akerss, "Binary Decision Diagrams," IEEE Transactions on Computers, pp. 506-516, June 1978.

[21] M. S. Abadir and H. K. Reghbati, "Functional Test Generation for LSI circuits described by Binary Decision Diagrams," IEEE International Test Conference, pp. 483-492, Nov. 1985.

[22] H. P. Chang, W. A. Rogers, and J. A. Abraham, "Structured Functional Level Test Generation Using Binary Decision Diagrams," IEEE International Test Conference, pp. 97-104, Sep. 1986.

[23] C. Robach and G. Saucier, "Microprocessor Functional Testing," IEEE International Test Conference, 1980.

[24] S. M. Thatte and J. A. Abraham, "Test Generation for Microprocessors," IEEE Transactions on Computers, vol. c-29, pp. 429-441, June 1980.

[25] B. T. Murray and J. P. Hayes, "Hierarchical Test Generation using Precomputed Tests for Modules," *IEEE International Test Conference*, pp. 221-229, April 1988.

[26] G. Kruger, "A Tool for Hierarchical Test Generation," *IEEE Transactions on Computer-Aided Design,* pp. 519-524, Vol. 10, April 1991.

[27] T. Sarfert, R. Markgraf, E. Trischler and M. Schultz, "Hierarchical Test-Pattern Generation based on High-Level Primitives," *IEEE Transactions on Computer-Aided Design,* pp. 34-44, Vol. 11, No. 1, January 1992.

[28] P. C. Ward, J. R. Armstrong, "Behavioral Fault Simulation in VHDL," *27th ACM/IEEE Design Automation Conference*, 1990.

[29] B. Singh, J. Wicks, P. Wright and J. R. Armstrong, "The Modeler's Assistant: A CAD Tool for Behavioral Model Development," *Proc. of CHDL '93*, April 1993, pp. 347-354.

[30] James R. Armstrong, F. Gail Gray, "Structured Logic Design With VHDL," PTR Prentice-Hall, Inc., 1993.

[31] VHDL Tool Integration Platform (VTIP)/Design Library System (DLS) Manual, CAD Language Systems, Inc., 1993.

[32] The VHDL System Simulator User's Manual, Synopsys, Inc., 1990.

[33] Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman, "Data Structures and Algorithms," Addison-Wesley Publishing Company, 1983.

[34] Aaron M. Tenenbaum, Yedidyah Langsam, Moshe J. Augenstein, "Data Structures Using C," Prentice Hall of India Private Limited, 1992.

[35] M. Abramovici, M. A. Breuer, A. D. Friedman, "Digital Systems Testing and Testable Design," Computer Science Press, 1990.

Appendix A: Programmer's Manual for PTG, TBG and their Environment

Introduction

This programmer's manual explains the various source files and the header files that constitute the Process Test Generation (PTG) software and the Test Bench Generation (TBG) program. It illustrates the various data structures that are used in the construction of Control Flow Graphs, Hash Tables, etc. The manual also explains the compilation procedures for the two programs and guides how the user can run these programs to get results.

The Process Test Generation Software Files

The source code for the process test generation software, i.e., the construction of Control Flow Graphs, the construction of Hash Tables and the Process Test Generator (PTG) algorithm, is contained in six C files and five header files. A brief description of each of the files and their contents is given in this section.

head.h

Various "#include" and "#define" statements, and all the data structures globally used by the process test generation software are included in this file. The "#include" statements are used to include the various standard 'C' header files, like the "stdio.h", "stdlib.h", "ctype.h", "malloc.h", and also to include the SPI.h header file, as described in chapter 5. These files are included so that the program can be compiled with these and so that the routines and procedures defined in these files could be used, e.g., the inclusion of SPI.h header file allows us to use all the routines of Software Procedural Interface package, that is used to access the design information stored in the Design Library System (DLS) of the VTIP software. Similarly, "#define" statements have been included to define constants that can be globally used in all the PTG software files. The data structures defined and declared in the head.h file are explained in greater detail in the next section. This file is included in all the six 'C' files of the PTG software.

decl.h

This file is also included in all the six 'C' files of the PTG software. Various "#define" statements have been included to define some other constants which can be globally used. Certain global variables, of the general 'C' data types or of the data structure types defined in file head.h, are also declared in this file. Finally, all the functions defined in all the six files of the PTG software are also declared in *decl.h*.

vhdlm2.h

This is a modified version of the header file created earlier for Modeler's Assistant. Globally used data structures are defined which are used to store the graphical and design information as needed by Modeler's Assistant. Various "#define" statements are used to define constants for use by Modeler's Assistant software. Some variables which are used in some other files are declared as *externs*.

These files have been included in the PTG software because at times it is required to access the geometrical and other design information related with a Process Model Graph (PMG) that is still the base of the entire system.

The original file (i.e., vhdlm.h) was modified to get the new file vhdlm2.h. The original file declared two variables, "Node" of special structure type *a_node* (typedef structure defined for a PMG), and FreeList of type *int*. Both the variables are predefined in the VTIP/DLS and SPI software which is extensively used throughout the PTG software.

Thus, these variables had to be changed to a_Node and FreeList1 respectively to avoid any conflicts, and so a new header file called "vhdlm2.h" was created.

macrosm2.h

This file defines various macros which are used to reference the fields of the data structure *a_node* defined in vhdlm2.h. These macros are used during process test generation to access useful data in the PMG database. As in the case of vhdlm2.h, the original file used to define the macros was called "macrosm.h", but because of the reason explained in the last section on "vhdlm2.h", the original file was modified and named macrosm2.h

externs.h

This file contains a few extern declarations for some functions that are called from other PTG software files other than that in which the functions are declared. This is mainly used by the file ***tgen.c*** and the functions used are actually declared in file pmg_info2.c.

pmg_info2.c

The functions used in this file were again previously developed to store information about the Process Model Graph created using Modeler's Assistant. As explained in the case of "vhdlm2.h", the functions were slightly modified and the new file was called

"pmg_info2.c". It contains the function *Init_nodes* which is used to initialize the pointers in the PMG nodes and to create an elementary link-list. The function *load_unit* stores information about the top PMG node, called a unit. The functions *load_subunit* and *load_module* are then called to store information about individual processes of the PMG. The functions in this file are used to access information in the PMG database during process test generation, e.g., to get the *portorder*, i.e., the order in which the ports of a PMG process are linked together in a link-list.

mvhd.c

This file is not a part of the PTG software. It generates the VHDL behavioral model of a process that belongs to a PMG previously produced. It is an independent program and is used to create a complete VHDL source for the model before the PTG can be invoked. It helps in creating the source directly from the process block VHDL code dumped by the Modeler's Assistant.

ptgmain.c

This is the main file of the PTG software and the functions defined in the other files are either called directly or indirectly from the "main" part of this file. This file includes all the functions that are used to create a Control Flow Graph. It consists of the specially defined functions used to extract information from the Design Library System of the VTIP software where the analyzed VHDL design information is stored. After the Control Flow

Graph generation is completed, the process test generator program is invoked from this file and the function to store the results is also finally called from this file.

makehash.c

This file consists of the *hash* and other related functions that are used to generate a *hash table*, as described in chapter 4. The functions in this file are called by the functions in the main process test generation file, called ***tgen.c***.

tgen.c

This file consists of almost all the functions that are used for the process test generation. It consists of functions described earlier, like *find_controllable_node*, finding *observable node* function, finding *shortest path* function, and many more. This is the most important file and consists of the most crucial functions that manipulate the data stored in different CFG data structures to generate and store the test sets.

assign_tgen.c

This file consists of a few more functions that are used for the process test generation when the event is created at an assignment_cfg_node. These are called by the functions defined in the main test generation file, called *tgen.c*.

misc.c

This file consists of the functions that are not directly related to process test generation but are otherwise needed for successful programming. This file includes functions like "*strrev*", used for reversing a string, "*strupr*" used for converting a string to upper case and "*atoi*" to convert a string to its equivalent integer value.

The Data Structures used by the PTG software

The data structures used by the process test generation software are described in this section. There are in all sixteen data structures used, each with a number of fields to hold different kind of data information. These data structures are used for Control Flow Graph generation, Hash Table generation, and for Process Test Generation and storage of results.

The data structures used are either defined using 'C' language construct "*typedef*", which creates a new data type name so that any variable can then be declared to be of this data type, or they are simply defined using the construct "*struct*". All the sixteen data structures are described in this section. Most of the data structures are self-referential and have double link pointers that help in linking these structures with other structures both in the forward and the backward directions to make a double-link list.

mod_top

The data structure mod_top serves as a declaration node for the VHDL model. It is also explained in chapter 4. The data structure is defined as follows:

```
typedef struct mod_node *mod_top;
typedef struct mod_node {
char *name;
decl_top decl_ptr[MAX_DECLS];
};
```

A structure, called mod_node, is first defined and then using *typedef* a data type name, called *mod_top* is created. The data type *mod_top* has two fields: *name*, which is a string of type *char*, and *decl_ptr*, which is of type *decl_top*, another data type name yet to be discussed. The field *name* is used to store the name of the file containing the VHDL source for the model. The field *decl_ptr* is used to point to the entity declaration of the model.

decl_top

This data structure is used to hold the information about the entity declaration within the VHDL model. It is defined as follows:

```
typedef struct decl_node *decl_top;
typedef struct decl_node {
char name[MAX_CHARS];
sig_var_node sig_ptr;
mod_top mod_link;
body_top body_ptr;};
```

The data structure, called decl_node is defined and it is given a data type name decl_top using typedef. The field name is used to store the name of the entity. The field sig_ptr, of data type sig_var_node (to be discussed), points to the link list of port signal nodes carrying information about the port signal declarations within the entity. The field mod_link links the entity back to the model declaration node, i.e., the node mod_top, described above, while the attribute body_ptr, of data type body_top, points to the architecture body declaration within the VHDL model.

sig_var_node

This data structure is used to store information about a signal or a variable. It is defined as follows:

```
typedef struct signal_var_type *sig_var_node;
typedef struct signal_var_type {
 sig_var_node pre_link;
char name[MAX_CHARS];
 int sig_var;
char type[15];
ObjectType mode;
int range;
char up_down[16];
 int sense;
char signal_value[MAX_CHARS];
char new_signal_value[MAX_CHARS];
int ref;
sig_var_node next_link;

};
```

The above data structure is self-referential because the field pre_link is of the same data type as the structure to which it belongs and so does the field next_link. The field pre_link links a sig_var_node structure to another sig_var_node structure created earlier or in the

backward direction, while the field next_link links it to a similar structure in the forward direction. In this way a link-list of sig_var_nodes can be maintained that holds the information for all the signals and the variables in the VHDL model.

The field name stores the name of the signal, integer sig_var tells us whether the structure is for a signal or a variable ('0' for signal and '1' for variable) and string type gives the type of the signal (e.g., BIT, MVL4, etc.). The data type ObjectType is a predefined type of VTIP/SPI software. The field mode is defined to be of type ObjectType and gives the type of the signal, i.e., InMode, OutMode or InternalMode. The integer field range gives the range of the signal or the variable ('1' for single-bit), the field informs whether the range is ascending or descending and the integer code sense is used to identify whether the signal is in the sensitivity list or not ('1' for sensitivity). the character string fields signal_value and new_signal_value are used to store the first and the last signal values at the 1st and the 4th ns during simulation, respectively, within the process test generation. The integer code ref is just a measure to calculate how many times each signal is accessed during test generation.

body_top

This structure stores the information about the architecture body declaration. It is defined as:

```
typedef struct body_node *body_top;
typedef struct body_node  {
char name[MAX_CHARS];
mod_top mod_link;
decl_top extend_link;
```

```
sig_var_node sig_ptr;
pro_top pro_ptr;
};
```

The character string name stores the name of the architecture body, while the field mod_link points to the model declaration node, i.e., mod_top. The field extend_link links the architecture body to its extension, the entity body declaration, i.e., the structure decl_top, while the sig_ptr pointer points to the link list of internal signal nodes, the data structures of type sig_var_node. The field pro_ptr points to the top of the process block declaration within the architecture body.

pro_top

This data structure stores the information about the process declaration within an architecture body. It is defined as:

```
typedef struct process_top *pro_top;
typedef struct process_top {
body_top parent_node;
char pro_label[MAX_CHARS];
sig_var_node sig_ptr[MAX_SENSE];
sig_var_node var_node[MAX_VAR];
pro_node top_pro_stats[10];
};
```

The field parent_node points to the architecture bode declaration within which the process lies. The character string field label holds the name of the process label, if any. An array field, called sig_ptr, of length MAX_SENSE (i.e., 10 as defined in head.h file) is defined, each of the ten array pointers pointing to at most ten sensitive input signals among the port signal declaration nodes. The array field var_node similarly points to at most MAX_VAR number (i.e., 25) of variable nodes corresponding to the variables declared

inside a VHDL process. The array field top_pro_stats, of length 11, is not fully used at present. At present we are just creating Control Flow Graphs and test sets for single process behavioral models, so only top_pro_stats[0] is used and it points to the first CFG_NODE inside the process representing the first process VHDL statement. This provision has been made to facilitate the construction of graph for a VHDL behavioral model with at most 11 processes inside the architecture body.

pro_node (CFG_NODE)

This is the most important data structure that is defined to store the crucial information about a CFG_NODE. It has a number of fields, but only the most important ones are described in this section. The other fields are properly documented in the source code and are only used for ease in programming.

```
typedef struct process_node *pro_node;
typedef struct process_node  {
 int node_num;
pro_top top_to_pro;
pro_node pre_link[50];
char kind[MAX_CHARS];
int type;
int line;
cont_node cont_ptr;
ass_node ass_ptr;
int ref;
struct in_sig_link_list *in_sig_node_ptr;
int distance;
int succeed;
int precede;
int control_select;
int pass_mark;
pro_node next_tnode;
pro_node next_fnode;

};
```

The integer node_num is the number of the CFG_NODE in the Control Flow Graph (CFG). The top_to_pro pointer, of data type pro_top, points to the process declaration. The pro_node or the CFG_NODE is also a self-referential node. The array pointers pre_link, of same data type, are used to maintain links back from this node to all other nodes that have one of their true or false edges pointing towards this node. The character string kind stores the type of VHDL statement that the CFG_NODE represents, e.g., CaseStatement for a CASE VHDL statement, IfStatement for an IF VHDL statement, etc. The integer code type is used to identify the type of corresponding VHDL statement represented by the node like the field kind, e.g., if the code is '1', then the statement is a CASE statement , and so on. The integer line represents the corresponding statement number of the VHDL statement that the CFG_NODE represents. The field cont_ptr, of data type cont_node, is used to hold information about a condition , if any, specified by the corresponding VHDL statement, while the field ass_ptr, of data type ass_node, is used to store information about an assignment, if any, specified by the corresponding VHDL statement. The integer ref gives the number of times a CFG_NODE is referred during process test generation. The pointer in_sig_node_ptr points to the link-list of nodes that store the information about the associated input signal nodes with the CFG_NODE. The integer distance, as described in the explanation of hash functions in chapter 4, gives the distance of the CFG_NODE from another CFG_NODE during hash table construction. Similarly, the integer fields, succeed and precede, give the numbers of the successor and predecessor CFG_NODES during the shortest path construction between two CFG_NODEs. The integer code pass_mark, if equal to '1', informs that the node has already been passed during the process test generation. The two forward pointing edges of the CFG_NODE are: next_tnode which points to the CFG_NODE that will be reached next during execution if the condition on the CFG_NODE is true, and next_fnode which

points to the CFG_NODE that will be reached next during execution if the condition on the CFG_NODE is false.

cont_node

This data structure is used to store any information pertaining to a VHDL condition statement or to the source part (i.e., expression, signal value, etc.) of an assignment statement. Some of the important fields of the structure are defined here.

```
typedef struct condition_node *cont_node;
typedef struct condition_node {
pro_node main_link;
char top_oper_link[MAX_CHARS];
char value[MAX_CHARS];
sig_var_node arg1;
sig_var_node arg2;
char val1[MAX_CHARS];
char val2[MAX_CHARS];
cont_node oper1;
cont_node oper2;
};
```

The field main_link links the node to the pro_node or the CFG_NODE to which the condition belongs. Let us consider two cases that explain the meanings of the fields more clearly. The two example cases are:

```
1|    A='1' and B='0';
2|    A or B;
```

The top_oper_link field stores the top-most VHDL operator that is encountered, e.g., in the case 1 it is the operator 'and', while in the case 2 it is operator 'or'. The field value

holds the signal value specified in a CASE statement clause, e.g., the value "00" in the clause when "00". The field arg1 points to the signal node to the left of the top-most operator in the condition, while the field arg2 points to the signal node to the right of the top-most operator in the condition, e.g., in the case 2 above, arg1 points to the signal node for the signal 'A' and arg2 points to that of signal 'B'. The fields oper1 and oper2 are of significance in cases like case 1 above where the arguments of the top-most operator (in this case: 'and') are operations themselves, e.g., in case 1, there are two such operations: A='1' and B='0'. Both of these are interpreted as cont_nodes again. The oper1 points to the cont_node for operation on the left, i.e., A='1', while oper2 points to B='0'. Now, each of these operations are again represented by cont_nodes. The value '1' assigned to the signal 'A' in the oper1 cont_node will be stored in the field val1, to denote that it belongs to the operation towards the left of the top-most operator, while the value '0' assigned to the signal 'B' in the oper2 cont_node will be stored in the field val2 to denote that it belongs to the operation towards the right of the top-most operator.

ass_node

This data structure holds the information regarding an assignment statement within the VHDL process. It is defined as:

```
typedef struct assign_node *ass_node;
typedef struct assign_node {
sig_var_node target_node;
cont_node source_node;
};
```

The field target_node points to the signal node corresponding to the signal that is the target of the assignment, whereas the field source_node points to the cont_node that stores the information about the source expression, signal value, etc. assigned to the target, as explained in the last section.

HASH_LISTPTR

This data structure defines a node that is used to point to a signal and its associated CFG_NODEs that are linked at the hash of key hash table index. The data structure is:

```
typedef struct hash_list *HASH_LISTPTR;
typedef struct hash_list
{
HASH_LISTPTR last;
sig_var_node sig_link;
pro_node pro_link;
int ref;
HASH_LISTPTR next;

};
```

This is a self-referential data structure. The field last is a pointer pointing to the previous HASH_LISTPTR node, while the field next points to the next HASH_LISTPTR in a link list. The field sig_link points to the signal node corresponding to the signal for which the hash node is constructed. The field pro_link points to a CFG_NODE that is associated with the signal pointed to by sig_link. The integer ref gives the number of times that this node has been referred during test generation.

in_sig_link_list

This is a data structure that points to an input signal node and to a next in_sig_link_list node, i.e., it is also a self-referential data structure. It is defined as:

```
struct in_sig_link_list
{
sig_var_node sig_node_link;
struct in_sig_link_list *next;

};
```

This is used to make a link list of data structures that store the input signals linked with a CFG_NODE. They are used during the hash table construction.

port_order_node

This is defined as:

```
struct port_order_node
{
struct port_order_node *last;
char port_name[MAXLINE];
sig_var_node sig_decl_link;
int arch_sig_decl;
char signal_value[64];
char symbol[16];
char new_signal_value[64];
char initialize_symbol[8];
struct port_order_node *next;

} port_order_info;
```

The nodes of the type of this data structure are used to store the signals and the related information, in the order in which signals are stored in the PMG database of Modeler's Assistant. This is necessary because the final test sets that are generated are written in this order in accordance with the process test data file format discussed in chapter 3.

This is also a self-referential data structure. The field last is used to point to the previous port_order_node that stores the information for the signal occurring before the current signal in the link list of PMG database, while the field next points to the port_order_node for the next signal along the PMG database link list. The sig_decl_link field points to the signal node for which the information is stored. The integer code arch_sig_decl identifies whether the signal concerned is an internal signal or a port signal. The character string field signal_value is used to store the last value of the signal (at 1 ns) while the field new_signal_value holds the value at the 4ns. The field symbol stores the symbol for the signal in accordance with the notation defined in chapter 3. The field initialize_symbol stores the initial value for the signal, as specified by the user, if the signal is the output port of a sequential circuit.

sense_signals

The data structure is defined as:

```
struct sense_signals
{
struct sense_signals *last;
sig_var_node sig_node_link;
struct sense_signals *next;

};
```

This is used to create a link list of data structures with pointers to the sensitive input signals of the model. It is also a self-referential data structure with the last and the next fields pointing to the previous and the next sense_signals nodes in the link list.

in_signals

This structure is defined as:

```
struct in_signals
{
struct in_signals *last;
sig_var_node sig_node_link;
struct in_signals *next;

};
```

This is used to create a link list of data structures with pointers to the input signals of the model. It is also a self-referential data structure with the last and the next fields pointing to the previous and the next in_signals nodes in the link list.

out_signals

The data structure is defined as:

```
struct out_signals
{
struct out_signals *last;
sig_var_node sig_node_link;
struct out_signals *next;};
```

This is used to create a link list of data structures with pointers to the output signals of the model. It is also a self-referential data structure with the last and the next fields pointing to the previous and the next out_signals nodes in the link list.

short_path

This structure is defined as:

```
struct short_path
{
struct short_path *last;
pro_node pro_link;
char condition[8];
int cond_or_ass;
struct short_path *next;

};
```

It is used to store the information about a pro_node or the CFG_NODE along a shortest path selected between two CFG_NODEs. This is also a self-referential structure, the last pointer pointing to the previous short_path node along the selected shortest path, while the next pointer pointing to the next node along the shortest path. The field pro_link points to the pro_node (or CFG_NODE) that is included in the shortest path selected. The character string field condition stores either "TRUE" or "FALSE" depending on if the condition needs to be true or false in going from this node to the next node along the shortest path. The integer code field cond_or_ass identifies whether the corresponding CFG_NODE is a condition_cfg_node (when it is '0') or it is an assignment_cfg_node (when it is '1').

test_results

This structure is again a self-referential structure that forms a link-list to store the final process test sets. It is defined as:

```
struct test_results
{
struct test_results *last;
int test_set_nu;
int nu_of_frames;
struct port_order_node *event_link;
struct test_results *next;

};
```

The last and the next pointers point to the previous and the next test_results nodes. The test_set_nu gives the number of the test set that is generated. The integer nu_of_frames gives the number of time frames in the test set. The field event_link of the structure type port_order_node points to the link list of signals, linked in their port orders, that store the relevant information like the signal values and the symbols to be inserted in the results file.

The Files used by TBG program

The Test Bench Generator (TBG) program uses the files vhdlm.h, macrosm.h, externs.h, pmg_info.c and tbg.c. All the files except tbg.c have been explained earlier. The file ***tbg.c*** is the main file and consists of about 1300 lines of 'C' code. It consists of the "main" program and the functions that extract the information from the PMG database using the header files and the pmg_info.c file. The functions to parse the final output test sequence generated by HBTG, are also included.

The Compilation Procedure for PTG software

There is only one executable file, called ***ptg***, that constructs the Control Flow Graph, produces the Hash Table and generates the final test sets. As stated earlier, the system is compiled with SPI object libraries that help in extracting the information from the VTIP/DLS. Also, the PTG software itself has seven 'C' source files. In order to compile these source files with the SPI libraries, a special Makefile was created for fast and easy compilation. The command used to compile the files and generate the executable ptg is:

% *make*

The Makefile created to facilitate this is shown in the figure below:

```
CC = cc -I/cad/clsi/vtip/dls/dev/include -L/cad/clsi/vtip/dls/lib/ -lSPI
ptg: ptgmain.c misc.o makehash.o pmg_info2.o tgen.o assign_tgen.o
        $(CC) misc.o makehash.o pmg_info2.o tgen.o assign_tgen.o ptgmain.c -o ptg
misc.o: misc.c
        $(CC) -c misc.c
makehash.o: makehash.c
        $(CC) -c makehash.c
pmg_info2.o: pmg_info2.c
        $(CC) -c pmg_info2.c
tgen.o: tgen.c
        $(CC) -c tgen.c
assign_tgen.o: assign_tgen.c

        $(CC) -c assign_tgen.c
```

Figure 44. The Makefile used for compiling the PTG Software

In the Makefile shown above, the -I switch specifies the directory to search for "#include" files, "-L" switch specifies the directory to search for object libraries, "-l" switch specifies the SPI object library, "-c" switch specifies to generate an object file and not an executable, and "-o" switch specifies the name of the output executable file.

The Compilation Procedure for TBG program

The TBG program also has just one executable file, called ***tbg***. The compilation procedure for the TBG program is:

% cc pmg_info.c -g -o tbg

Executing the PTG and the TBG software

The PTG software is executed using the command:

% ptg

The user is prompted to enter the following information before the results are generated.

Enter the filename: ***<the name of the file containing the VHDL model>***
Enter unit name: ***<the name of the PMG unit that contains the process>***
Enter process name: ***<the name of the process as in the PMG>***
Enter the Library name: ***<the name of the working library, "work" by default>***
Enter the Entity name: ***<the name of the entity of the VHDL behavioral model>***
Enter the Architecture name: ***<the name of the architecture of the VHDL behavioral model>***

The TBG program is executed using the command:

% tbg

The user is prompted to enter the following information before the test bench is generated.

TEST BENCH GENERATOR - VER 2.0: MAY, 1993

BRADLEY DEPT. OF ELECTRICAL ENGINEERING

VIRGINIA POLYTECHNIC INSTITUTE & STATE UNIVERSITY

Specifications.....

VHDL (.vhd) file: ***<the name of the VHDL model file>***
ENTITY NAME(default: filename_TEST_BENCH): ***<the name of the entity desired>***
ARCHITECTURE NAME(default: BEHAVIOR): ***<the name of the architecture desired>***
TEST SEQUENCE file (from HBTG; default: filename.test): ***<the file containing the final HBTG test sequence>***

Appendix B: The VTIP/DLS and SPI Software: PTG Related User's Manual

Introduction

This user's guide has been written to help the user analyze a VHDL behavioral model with proper options, using the VHDL Tool Integration Platform (VTIP) VHDL analyzer, and to enable the user to access the design information stored in the VTIP Design Library System (DLS) with the help of the Software Procedural Interface (SPI) package.

VTIP Design Library System

The VTIP Design Library System (DLS) provides the intermediate storage for analyzed VHDL design information. It consists of Design Libraries and Design Library Units. The

Design Library Units are the basic unit of storage for design data in the DLS. The Design Libraries are implemented as system directories and the Design Library Units are implemented as files stored in these directories.

The Design Library Units are represented by library logical names, to facilitate reference to each other and for portability. A library logical name may be either permanent or temporary. A permanent library logical name is mapped to a directory using the setenv command, as follows:

% *setenv permanent_library_name system_directory*

The temporary logical name used is generally WORK and it is mapped to a permanent logical name, using the setenv command as before.

% *setenv WORK permanent_logical name*

The temporary name is mapped to the permanent name instead of directly mapping it to the system directory, because this facilitates the moving of the library units from one temporary directory to another.

VTIP VHDL analyzer

The VTIP VHDL analyzer always requires access to two libraries: STD and WORK. The library STD is included in the VTIP package and the library WORK is the temporary

working directory as described. All the analyzed design library units are stored in the library WORK. The VHDL analyzer is invoked using the following command:

% *vhdl file_name[.ext] [-qualifiers]*

The .ext is the extension of the vhdl file_name, e.g., .vhd or .vhdl. The option "-qualifiers" allows the user to specify various switches to get the analyzed output of choice. For example, using th e"-list" qualifier switch results in the numbered source listing output to the file file_name.lis.

The DLS Architecture

The DLS consists of two basic components:

1. The *abstract* component, called ***DLS Data Definition***
2. The *concrete* component, called ***Software Procedural Interface*** **(SPI)**

The DLS Data Definition specifies the data elements, objects, and structures that are supported by the Design Library System, together with the abstract operations that can be performed upon them.

The SPI is a software implementation of DLS. It consists of the routines that can be used to access the data in the DLS.

The DLS Data Definition

The DLS Data Definition has three layers:

1. The Data Model

This defines the most primitive data types (e.g., BOOLEAN, INTEGER,etc.) and the generic data types (like NODES, ATTRIBUTES, LISTS, ITEMS) that are intended to represent the simplest concepts in any design data, e.g., numbers and character strings.

2. The DLS Schema

This is defined in terms of the primitive and generic data types of the DLS Data Model. It creates specific instances of the generic types, e.g., ***ObjectType*** is defined as an instance of the generic type Node. Various attributes are defined for each ObjectType and constraints are defined for each attribute.

The DLS Schema defines four categories of *ObjectTypes*: ***General ObjectTypes, Expression ObjectTypes, Miscellaneous ObjectTypes*** and ***Library Level ObjectTypes***. The top-level organization of the design data information stored by DLS consists of LIBRARIES that contain LIBRARY UNITS. A library is modeled by ObjectType ***LibraryRegion***.

3. DLS Information Model

It describes how ObjectTypes interconnect to form meaningful structures. It mainly describes how libraries and library units store design data information in a domain with a hierarchy of region nodes forming the backbone of the domain.

A black-box kind of diagram for the DLS is shown in Figure 45.

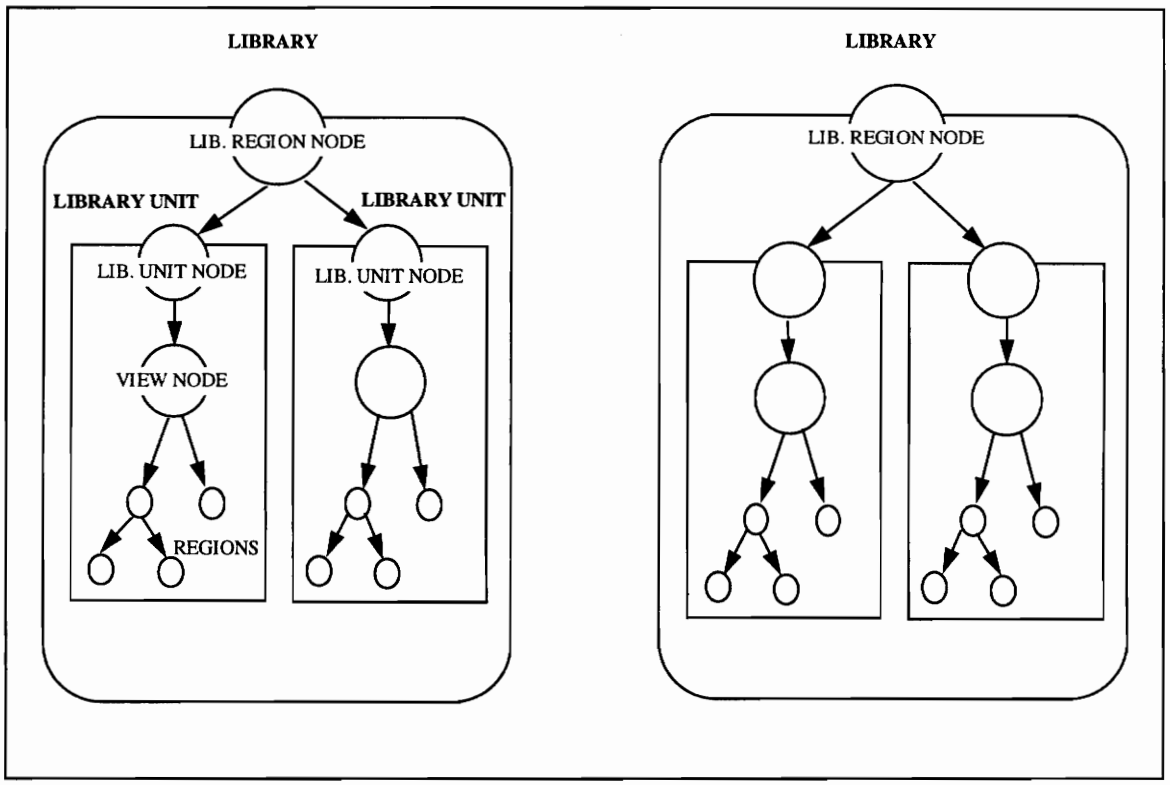


Figure 45. The VTIP Design Library System (DLS)

DLS Browser

The DLS Browser is a screen-oriented utility that allows the user to examine the DLS library units. The Browser enables a user to open library units, and traverse and examine the nodes, the lists and other data structures within a library unit.

The Browser is invoked by the command:

% *dlsbrowse [unit_name]*

The unit_name is the name of the library unit to be opened. It is also possible to use just the command word "dlsbrowse" and the utility guides the user to select different units and traverse the structures in them. The left arrow key is used to go to a previous screen of the DLS Browser while the right arrow key is used to go to the next one. The up and the down keys are similarly used to go up or down within a screen.

A typical DLS Browser screen looks like as shown in Figure 46.

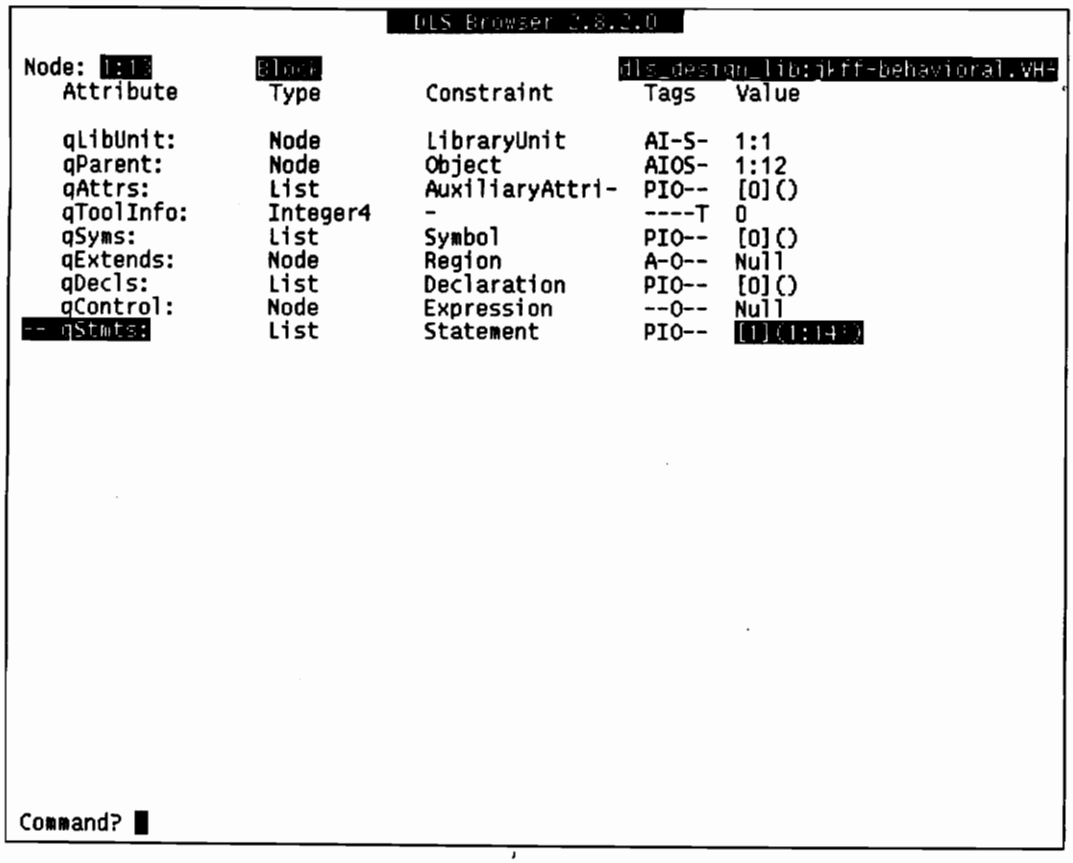


Figure 46. A DLS Browser Screen

Software Procedural Interface (SPI)

The SPI consists of data types and callable routines that implement the data types and operations of the DLS. It is used to access and manipulate design data stored within the DLS. Using SPI facilities, a design tool can open a library unit, traverse its internal structure, examine data or objects within the structure, and even add or modify data [31].

In order to be able to use the SPI routines, the SPI libraries are compiled with the PTG software files, as explained in appendix A. The SPI routines can then be included in the 'C' code of the process test generation software to facilitate their use.

There are a number of routines defined in the SPI package. Most of the routines used to access different ObjectTypes are named as the *ObjectType names* themselves. The DLS Browser, explained in the previous section, is an important tool in accessing the data stored in the DLS. Once a VHDL model is analyzed and stored in the DLS, the DLS Browser can then be used to traverse this design information and find out the ObjectTypes and the attributes needed to extract the information. Thus, using DLS Browser a programmer can know what functions or SPI routines he needs to extract the required design information. A portion of the PTG 'C' code used to generate Control Flow Graphs from the information stored in the DLS is shown in Figure 47.

```
OpenDLS();
Lib = NewLibrarySymbol(LibName);
OpenLibrary(Lib);

Pri_Ent_Unit = OpenUnit(Lib, EntityName, "", VHDLView, ModifyMode);
Arch_Unit = OpenUnit(Lib, EntityName, ArchName, VHDLView, ModifyMode);
                        .
                        .
                        .
  Ent_region = qRegion(qView(Pri_Ent_Unit));
  Ent_decl = Value(LastItem(qDecls(Ent_region)));
  Ent_region = qRegion(Ent_decl);
  Port_list = qDecls(Ent_region);
```

Figure 47. A portion of 'C' code to illustrate the use of SPI functions

The first routine that needs to be called before any other SPI calls can be made, is *OpenDLS*. This initializes the Design Library System and the SPI. A variable "*Lib*" is then given a value by calling a function *NewLibrarySymbol* with the name of the library supplied as a parameter. This function creates a *LibrarySymbol node* that can be used as a parameter to procedure *OpenLib*, which makes available the library units within it. Similarly, an appropriate library unit is opened by using the function *OpenUnit*. Function *OpenUnit* locates the specified library unit in the library represented by *LibrarySymbol* "*Lib*" and returns a pointer to the *LibraryUnit node* that is the root of the specified unit. The parameter *VHDLView* is a value of type *ObjectType* that identifies the kind of *View* we want to examine. The parameter *ModifyMode* specifies that the library unit is to be

opened with write access, which will allow modifications to be saved. Let "Pri_Ent_unit" be a pointer to the entity declaration unit. Using the *qView* attribute (which is also a function call) we get to the *VHDLView node* of the unit, and from here the *qRegion* attribute gets us to the library-level *Region node*. All of the library logical names declared by the unit are listed here. The last item in the *qDecls* list of this node is the declaration of the design unit itself. The *qRegion* attribute of that Declaration node gives us the *Region* of the entity body which consists of the port declarations and generics, if any.

It is obvious from the 'C' code portion and the explanation above that using the DLS Browser tool one can find out the attributes required to access certain information and with the help of the SPI routines it becomes possible to access any kind of VHDL information stored in the Design Library System.

This manual just gives an introduction to a convenient method of extracting the design data information lying in the DLS. A detailed description of the SPI routines can be obtained from the VTIP/DLS and SPI manuals.

Appendix C: Circuit Models, Control Flow Graphs and Process Test Results

This appendix contains the VHDL source listings for some circuits, the Control Flow Graphs generated for these models, and the Process Test Sets produced for these models. The text description of the Control Flow Graphs generated by the Process Test Generation software and a pictorial representation of CFGs in the form of diagrams, both are included. Finally, the stimulus/response process test sets generated by PTG program for each VHDL behavioral model, are shown. The process test sets are given by a process test data file with an extension *.tst*. The symbolic notation used is the one discussed in chapter 3.

Circuit: BUFFER

```
Levels  LINE # |----+----1----+----2----+----3----+----4----+----5----+----6-
          1 |use WORK.VHDLCAD.all, WORK.USER_TYPES.all;
          2 |
          3 |
          4 |entity Buff is
1         5 |  port (DO: out  BIT_VECTOR(0 to 7); ENBLD: in  BIT; BUF: in  BIT_VECTOR(0 to
                    7));
1         6 |end Buff;
          7 |
          8 |architecture BEHAVIOR of Buff is
1         9 |begin
1        10 |   Buff_1: process (ENBLD,BUF)
2        11 |   begin
2        12 |      if (ENBLD='1') then
2 1      13 |         DO <= BUF;
2 1      14 |      else
2 1      15 |         DO <= "11111111";
2 1      16 |      end if;
2        17 |
2        18 |
2        19 |   end process Buff_1;
1        20 |end BEHAVIOR;
```

Figure 48. The VHDL behavioral model of the circuit BUFFER

THE Control Flow Graph (CFG) Nodes and the edges are: .....

CFG Node # corresponding to statement # 12 is: 0
.... its True_edge points to node #: 1(line# 13)
.... its False_edge points to node #: 2(line# 14)

CFG Node # corresponding to statement # 13 is: 1
.... its True_edge is UnDefined
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 14 is: 2
.... its True_edge points to node #: 3(line# 15)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 15 is: 3
.... its True_edge is UnDefined
.... its False_edge is UnDefined

Figure 49. The CFG specifications for the model BUFFER

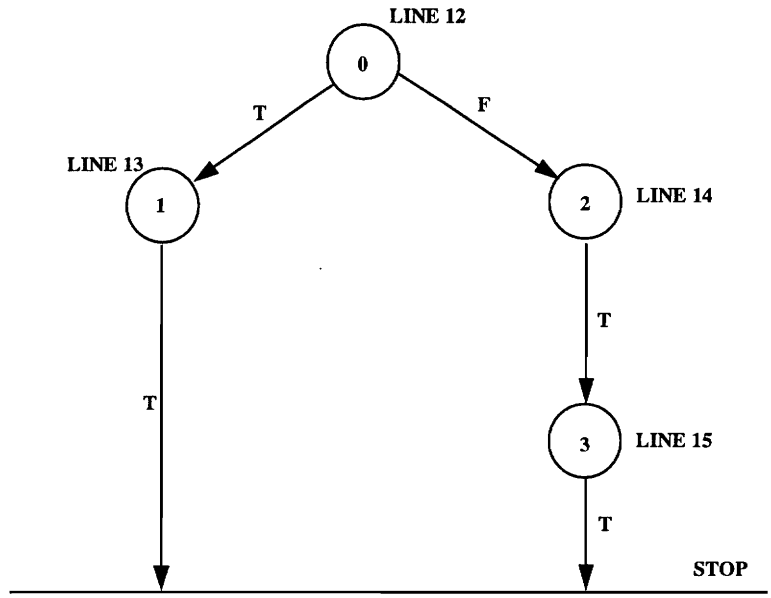


Figure 50. Control Flow Graph for the model BUFFER

portorder: DO ENBLD BUF

1
D1 R D1
1
D2 1 D2
1
D3 1 D3
1
DZ F D4*
2
D1 R D1
D1 1 D1
2
D2 1 D2
D2 1 D2
2
D3 1 D3
D3 1 D3
2
DZ F D4
DZ 0 D4

Figure 51. Process Test Results for the circuit BUFFER

* *'DZ' represents the high impedance value, i.e., "11111111".*

Circuit: SERVRQ

```
Levels  LINE # |----+----1----+----2----+----3----+----4----+----5----+----6-
         1 |
         2 |use WORK.VHDLCAD.all, WORK.USER_TYPES.all;
         3 |-- ************************************************************
         4 |entity SERVRQ is
1        5 | port (SRQ: inout BIT;
1        6 |   STRB: in BIT;
1        7 |   S: in BIT);
1        8 |end SERVRQ;
         9 |-- ************************************************************
        10 |
        11 |architecture BEHAVIORAL of SERVRQ is
1       12 |
1       13 |begin
1       14 |
1       15 | -- ---------------------------------------------------
1       16 | -- Process Name: SERVRQ
1       17 | -- ---------------------------------------------------
1       18 |
1       19 | SERVRQ_2: process (STRB,S)
2       20 | begin
2       21 |    if STRB = '0' and S = '1' then
2 1     22 |      SRQ <= '0';
2 1     23 |    elsif S = '0' then
2 1     24 |      SRQ <= '1';
2 1     25 |    else
2 1     26 |      SRQ <= SRQ;
2 1     27 |    end if;
2       28 |
2       29 |
2       30 | end process SERVRQ_2;
1       31 |
1       32 |
1       33 |end BEHAVIORAL;
```

Figure 52. The VHDL behavioral model of the circuit SERVRQ

THE Control Flow Graph (CFG) Nodes and the edges are: .....

CFG Node # corresponding to statement # 21 is: 0
.... its True_edge points to node #: 1(line# 22)
.... its False_edge points to node #: 2(line# 23)

CFG Node # corresponding to statement # 22 is: 1
.... its True_edge is UnDefined
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 23 is: 2
.... its True_edge points to node #: 3(line# 24)
.... its False_edge points to node #: 5(line# 27)

CFG Node # corresponding to statement # 24 is: 3
.... its True_edge points to node #: 4(line# 25)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 25 is: 4
.... its True_edge is UnDefined
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 27 is: 5
.... its True_edge points to node #: 6(line# 28)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 28 is: 6
.... its True_edge is UnDefined
.... its False_edge is UnDefined

Figure 53. The CFG specifications for the model SERVRQ

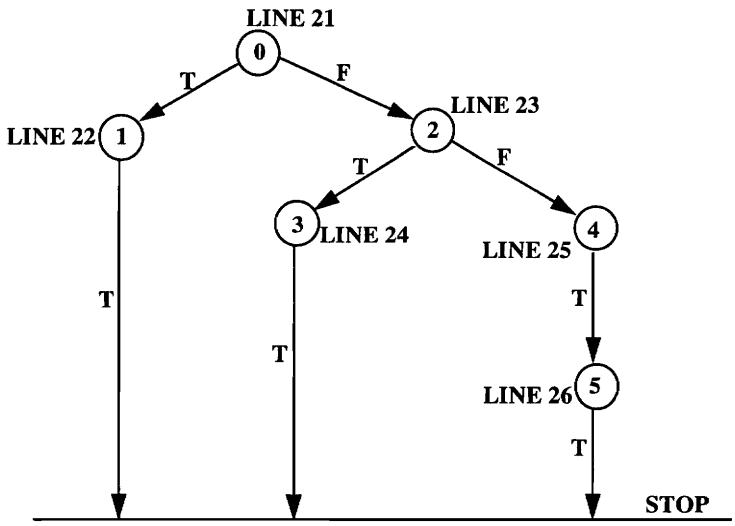


Figure 54. Control flow Graph for the model SERVRQ

portorder: SRQ STRB S

1
F F 1
1
R 1 F
1
P R 0
2
F F 1
0 0 1
2
R 1 F
1 1 0
2
P R 0
P 1 0

Figure 55. Process Test Results for the circuit SERVRQ

Circuit: VECTOR

```
Levels  LINE # |----+----1----+----2----+----3----+----4----+----5----+----6-
         1 |
         2 |use WORK.VHDLCAD.all, WORK.USER_TYPES.all;
         3 |-- ************************************************************
         4 |entity VECTOR is
1        5 | port (VECT: out BIT_VECTOR(7 downto 0);
1        6 |   NINT4: in BIT;
1        7 |   NINT3: in BIT;
1        8 |   NINT2: in BIT;
1        9 |   NINT1: in BIT);
1       10 |end VECTOR;
        11 |-- ************************************************************
        12 |
        13 |architecture BEHAVIORAL of VECTOR is
1       14 |
1       15 |begin
1       16 |
1       17 | -- ---------------------------------------------------
1       18 | -- Process Name: VECTOR
1       19 | -- ---------------------------------------------------
1       20 |
1       21 | VECTOR_2: process (NINT4,NINT3,NINT2,NINT1)
2       22 | begin
2       23 |   if NINT1 = '0' then
2 1     24 |     VECT <= "00011100";
2 1     25 |   elsif NINT2 = '0' then
2 1     26 |     VECT <= "00011101";
2 1     27 |   elsif NINT3 = '0' then
2 1     28 |     VECT <= "00011110";
2 1     29 |   elsif NINT4 = '0' then
2 1     30 |     VECT <= "00011111";
2 1     31 |   else
2 1     32 |     VECT <= "00011100";
2 1     33 |   end if;
2       34 |
2       35 |
2       36 | end process VECTOR_2;
1       37 |
1       38 |
1       39 |end BEHAVIORAL;
```

Figure 56. The VHDL behavioral model of the circuit VECTOR

THE Control Flow Graph (CFG) Nodes and the edges are: .....

CFG Node # corresponding to statement # 23 is: 0
.... its True_edge points to node #: 1(line# 24)
.... its False_edge points to node #: 2(line# 25)

CFG Node # corresponding to statement # 24 is: 1
.... its True_edge is UnDefined
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 25 is: 2
.... its True_edge points to node #: 3(line# 26)
.... its False_edge points to node #: 4(line# 27)

CFG Node # corresponding to statement # 26 is: 3
.... its True_edge is UnDefined
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 27 is: 4
.... its True_edge points to node #: 5(line# 28)
.... its False_edge points to node #: 6(line# 29)

CFG Node # corresponding to statement # 28 is: 5
.... its True_edge is UnDefined
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 29 is: 6
.... its True_edge points to node #: 7(line# 30)
.... its False_edge points to node #: 8(line# 31)

CFG Node # corresponding to statement # 30 is: 7
.... its True_edge is UnDefined
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 31 is: 8
.... its True_edge points to node #: 9(line# 32)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 32 is: 9
.... its True_edge is UnDefined
.... its False_edge is UnDefined

Figure 57. The CFG specifications for the model VECTOR

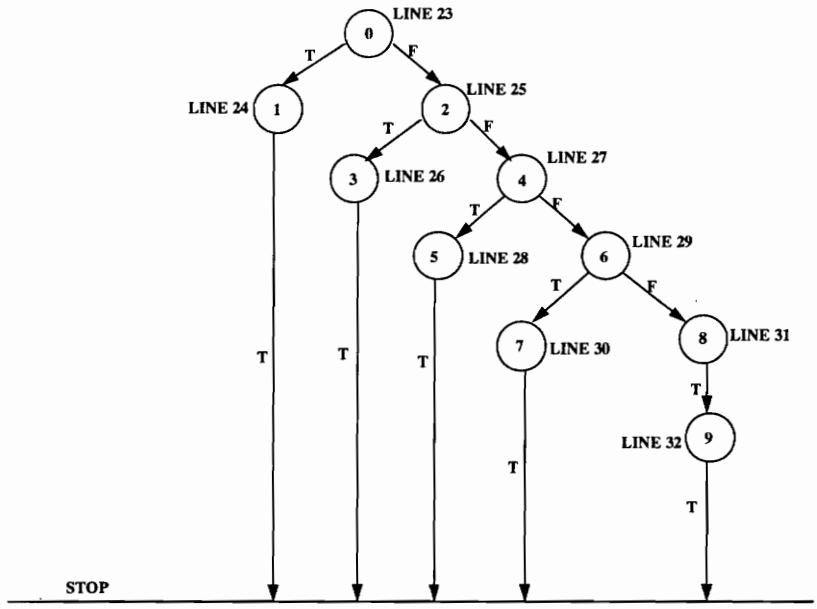


Figure 58. Control Flow Graph for the model VECTOR

```
portorder: VECT NINT4 NINT3 NINT2 NINT1

1
D1 F 1 1 1
1
D2 1 F 1 1
1
D3 1 1 F 1
1
D4 1 1 1 F
1
D4 1 1 1 R
2
D1 F 1 1 1
D1 0 1 1 1
2
D2 1 F 1 1
D2 1 0 1 1
2
D3 1 1 F 1
D3 1 1 0 1
2
D4 1 1 1 F
D4 1 1 1 0
2
D4 1 1 1 R
D4 1 1 1 1
```

Figure 59. Process Test Results for the circuit VECTOR

Circuit: CKTA

```
Levels  LINE # |----+----1----+----2----+----3----+----4----+----5----+----6-
          1 |
          2 |use WORK.VHDLCAD.all, WORK.USER_TYPES.all;
          3 |-- ************************************************************
          4 |entity Output is
1         5 |  port (D_OUT2: out BIT;
1         6 |    D_OUT1: out BIT;
1         7 |    D_IN2: in BIT;
1         8 |    D_IN1: in BIT;
1         9 |    EN: in BIT);
1        10 |end Output;
         11 |-- ************************************************************
         12 |
         13 |architecture BEHAVIORAL of Output is
1        14 |
1        15 |begin
1        16 |
1        17 |  -- ---------------------------------------------------
1        18 |  -- Process Name: Output
1        19 |  -- ---------------------------------------------------
1        20 |
1        21 |  Output_2: process (EN)
2        22 |  begin
2        23 |     if EN /= '0' then
2 1      24 |       D_OUT1 <= D_IN1;
2 1      25 |       D_OUT2 <= not D_IN2;
2 1      26 |     else
2 1      27 |       D_OUT1 <= not D_IN1;
2 1      28 |       D_OUT2 <= D_IN2;
2 1      29 |     end if;
2        30 |
2        31 |
2        32 |  end process Output_2;
1        33 |
1        34 |
1        35 |end BEHAVIORAL;
```

Figure 60. The VHDL behavioral model of the circuit CKTA

THE Control Flow Graph (CFG) Nodes and the edges are: .....

CFG Node # corresponding to statement # 23 is: 0
.... its True_edge points to node #: 1(line# 24)
.... its False_edge points to node #: 3(line# 26)

CFG Node # corresponding to statement # 24 is: 1
.... its True_edge points to node #: 2(line# 25)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 25 is: 2
.... its True_edge is UnDefined
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 26 is: 3
.... its True_edge points to node #: 4(line# 27)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 27 is: 4
.... its True_edge points to node #: 5(line# 28)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 28 is: 5
.... its True_edge is UnDefined
.... its False_edge is UnDefined

Figure 61. The CFG specifications for the model CKTA

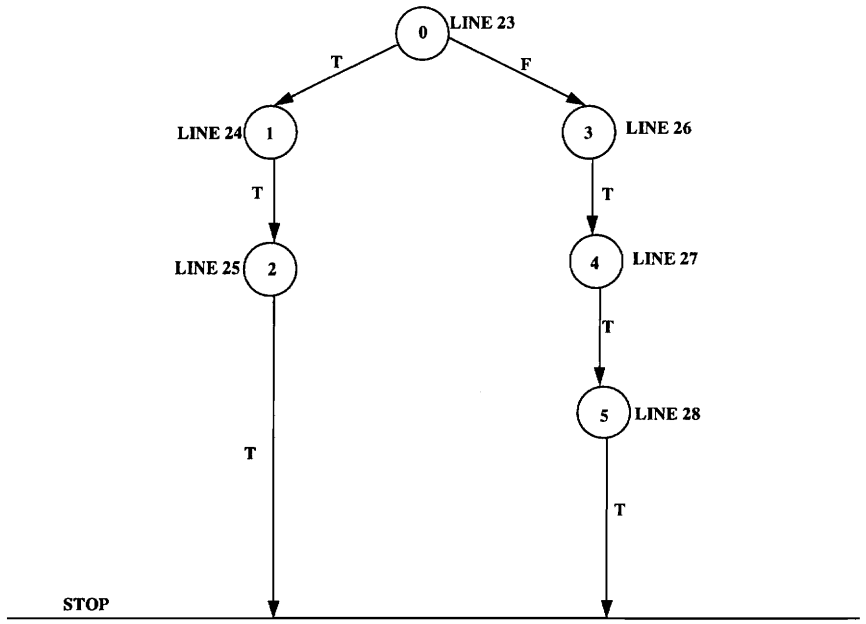


Figure 62. Control Flow Graph for the model CKTA

---

portorder: D_OUT2 D_OUT1 D_IN2 D_IN1 EN

1
F R 1 1 R
1
F F 1 0 R
2
F R 1 1 R
0 1 1 1 1
2
F F 1 0 R
0 0 1 0 1

Figure 63. Process Test Results for the circuit CKTA

Circuit: CKTB

```
Levels  LINE # |----+----1----+----2----+----3----+----4----+----5----+----6-
          1 |
          2 |use WORK.VHDLCAD.all, WORK.USER_TYPES.all;
          3 |-- ************************************************************
          4 |entity combo is
1         5 |  port (C: out BIT;
1         6 |    E: in BIT;
1         7 |    B: in BIT;
1         8 |    A: in BIT);
1         9 |end combo;
         10 |-- ************************************************************
         11 |
         12 |architecture BEHAVIORAL of combo is
1        13 |
1        14 |begin
1        15 |
1        16 |  -- ---------------------------------------------------
1        17 |  -- Process Name: combo
1        18 |  -- ---------------------------------------------------
1        19 |
1        20 |  combo_2: process (E,B,A)
2        21 |     variable Y: BIT;
2        22 |     variable X: BIT;
2        23 |  begin
2        24 |     if E='1' then
2 1      25 |        X := A;
2 1      26 |        Y := B;
2 1      27 |        C <= X and Y;
2 1      28 |     else
2 1      29 |        C <= '0';
2 1      30 |     end if;
2        31 |
2        32 |
2        33 |
2        34 |  end process combo_2;
1        35 |
1        36 |
1        37 |end BEHAVIORAL;
```

Figure 64. The VHDL behavioral model of the circuit CKTB

THE Control Flow Graph (CFG) Nodes and the edges are: .....

CFG Node # corresponding to statement # 24 is: 0
.... its True_edge points to node #: 1(line# 25)
.... its False_edge points to node #: 4(line# 28)

CFG Node # corresponding to statement # 25 is: 1
.... its True_edge points to node #: 2(line# 26)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 26 is: 2
.... its True_edge points to node #: 3(line# 27)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 27 is: 3
.... its True_edge is UnDefined
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 28 is: 4
.... its True_edge points to node #: 5(line# 29)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 29 is: 5
.... its True_edge is UnDefined
.... its False_edge is UnDefined

Figure 65. The CFG specifications for the model CKTB

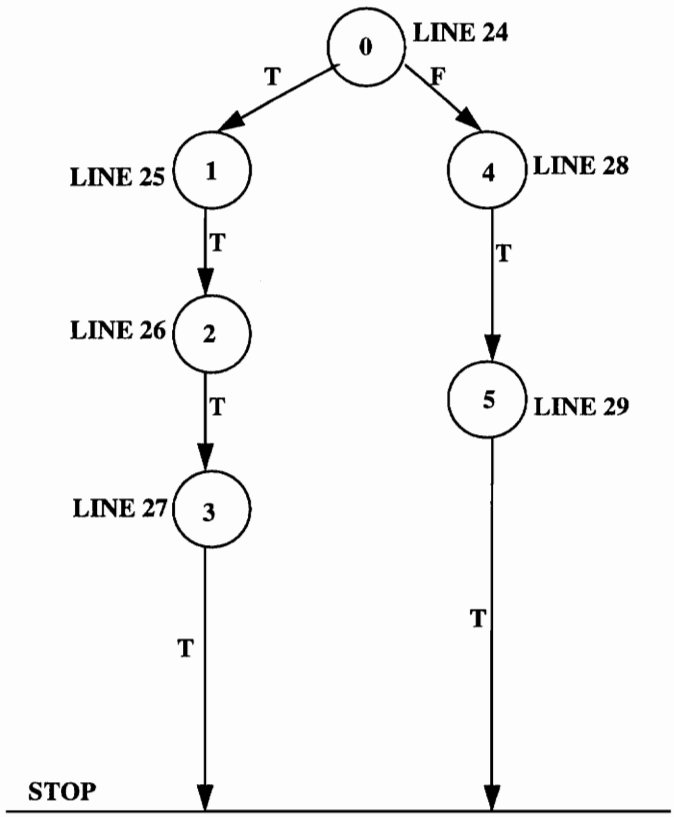


Figure 66. Control Flow Graph for the model CKTB

portorder: C E B A

```
1
0 R 0 1
1
0 1 F 1
1
R 1 R 1
1
0 1 1 F
1
R 1 1 R
1
0 F 0 0
2
0 R 0 1
0 1 0 1
2
0 1 F 1
0 1 0 1
2
R 1 R 1
1 1 1 1
2
0 1 1 F
0 1 1 0
2
R 1 1 R
1 1 1 1
2
0 F 0 0
0 0 0 0
```

Figure 67. Process Test Results for the circuit CKTB

Circuit: DECODER

```
Levels  LINE # |----+----1----+----2----+----3----+----4----+----5----+----6-
          1 |
          2 |use WORK.VHDLCAD.all, WORK.USER_TYPES.all;
          3 |-- ************************************************************
          4 |entity Decoder is
1         5 |  port (O: out BIT_VECTOR(3 downto 0);
1         6 |    I: in BIT_VECTOR(1 downto 0);
1         7 |    EN: in BIT);
1         8 |end Decoder;
          9 |-- ************************************************************
         10 |
         11 |architecture BEHAVIORAL of Decoder is
1        12 |
1        13 |begin
1        14 |
1        15 |  -- ---------------------------------------------------
1        16 |  -- Process Name: Decoder
1        17 |  -- ---------------------------------------------------
1        18 |
1        19 |  Decoder_2: process (I,EN)
2        20 |  begin
2        21 |     if EN = '1' then
2 1      22 |       case I is
2 2      23 |         when "00" => O <= "0001";
2 2      24 |         when "01" => O <= "0010";
2 2      25 |         when "10" => O <= "0100";
2 2      26 |         when "11" => O <= "1000";
2 2      27 |       end case;
2 1      28 |     end if;
2        29 |
2        30 |
2        31 |  end process Decoder_2;
1        32 |
1        33 |
1        34 |end BEHAVIORAL;
```

Figure 68. The VHDL behavioral model of the circuit DECODER

THE Control Flow Graph (CFG) Nodes and the edges are: .....

CFG Node # corresponding to statement # 21 is: 0
.... its True_edge points to node #: 1(line# 22)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 22 is: 1
.... its True_edge points to node #: 2(line# 23)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 23 is: 2
.... its True_edge points to node #: 3(line# 23)
.... its False_edge points to node #: 4(line# 24)

CFG Node # corresponding to statement # 23 is: 3
.... its True_edge is UnDefined
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 24 is: 4
.... its True_edge points to node #: 5(line# 24)
.... its False_edge points to node #: 6(line# 25)

CFG Node # corresponding to statement # 24 is: 5
.... its True_edge is UnDefined
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 25 is: 6
.... its True_edge points to node #: 7(line# 25)
.... its False_edge points to node #: 8(line# 26)

CFG Node # corresponding to statement # 25 is: 7
.... its True_edge is UnDefined
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 26 is: 8
.... its True_edge points to node #: 9(line# 26)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 26 is: 9
.... its True_edge is UnDefined
.... its False_edge is UnDefined

Figure 69. The CFG specifications for the model DECODER

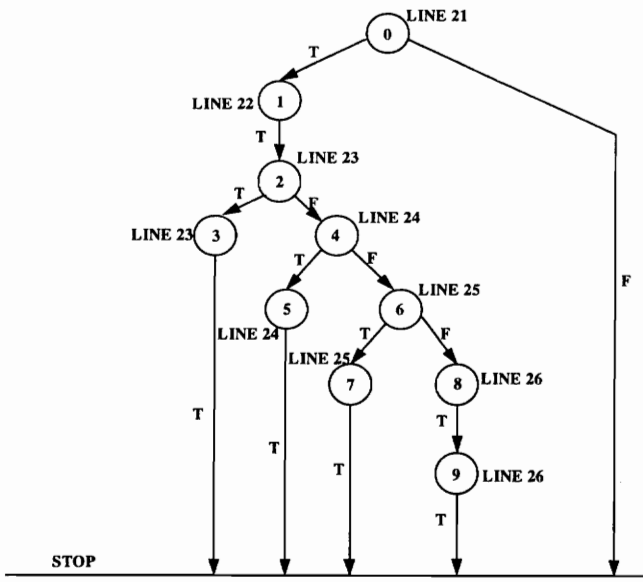


Figure 70. Control Flow Graph for the model DECODER

portorder: O I EN

1
D2 C1 1
1
D1 C0 R
1
D2 C1 R
1
D3 C2 R
1
D4 C3 R
2
D2 C1 1
D2 C1 1
2
D1 C0 R
D1 C0 1
2
D2 C1 R
D2 C1 1
2
D3 C2 R
D3 C2 1
2
D4 C3 R
D4 C3 1

Figure 71. Process Test Results for the circuit DECODER

Circuit: MUX

```
Levels  LINE # |----+----1----+----2----+----3----+----4----+----5----+----6-
          1 |
          2 |use WORK.VHDLCAD.all, WORK.USER_TYPES.all;
          3 |-- ************************************************************
          4 |entity muxx is
1         5 | port (DATA_OUT: out BIT_VECTOR(0 to 7);
1         6 |   D: in BIT_VECTOR(0 to 7);
1         7 |   C: in BIT_VECTOR(0 to 7);
1         8 |   B: in BIT_VECTOR(0 to 7);
1         9 |   A: in BIT_VECTOR(0 to 7);
1        10 |   SEL: in BIT_VECTOR(0 to 1);
1        11 |   ENB: in BIT);
1        12 |end muxx;
         13 |-- ************************************************************
         14 |
         15 |architecture BEHAVIORAL of muxx is
1        16 |
1        17 |begin
1        18 |
1        19 | -- ---------------------------------------------------
1        20 | -- Process Name: muxx
1        21 | -- ---------------------------------------------------
1        22 |
1        23 | muxx_3: process (SEL,ENB)
2        24 | begin
2        25 |    if (ENB='1') then
2 1      26 |       case SEL is
2 2      27 |             when "00" => DATA_OUT <= A;
2 2      28 |             when "01" => DATA_OUT <= B;
2 2      29 |             when "10" => DATA_OUT <= C;
2 2      30 |             when "11" => DATA_OUT <= D;
2 2      31 |       end case;
2 1      32 |    else
2 1      33 |       DATA_OUT <= "00000000";
2 1      34 |    end if;
2        35 |
2        36 |
2        37 | end process muxx_3;
1        38 |
1        39 |
1        40 |end BEHAVIORAL;
```

Figure 72. The VHDL behavioral model of the circuit MUX

THE Control Flow Graph (CFG) Nodes and the edges are: .....

CFG Node # corresponding to statement # 25 is: 0
.... its True_edge points to node #: 1(line# 26)
.... its False_edge points to node #: 10(line# 32)

CFG Node # corresponding to statement # 26 is: 1
.... its True_edge points to node #: 2(line# 27)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 27 is: 2
.... its True_edge points to node #: 3(line# 27)
.... its False_edge points to node #: 4(line# 28)

CFG Node # corresponding to statement # 27 is: 3
.... its True_edge is UnDefined
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 28 is: 4
.... its True_edge points to node #: 5(line# 28)
.... its False_edge points to node #: 6(line# 29)

CFG Node # corresponding to statement # 28 is: 5
.... its True_edge is UnDefined
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 29 is: 6
.... its True_edge points to node #: 7(line# 29)
.... its False_edge points to node #: 8(line# 30)

CFG Node # corresponding to statement # 29 is: 7
.... its True_edge is UnDefined
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 30 is: 8
.... its True_edge points to node #: 9(line# 30)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 30 is: 9
.... its True_edge is UnDefined
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 32 is: 10

.... its True_edge points to node #: 11(line# 33)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 33 is: 11
.... its True_edge is UnDefined
.... its False_edge is UnDefined

Figure 73. The CFG specifications for the model MUX

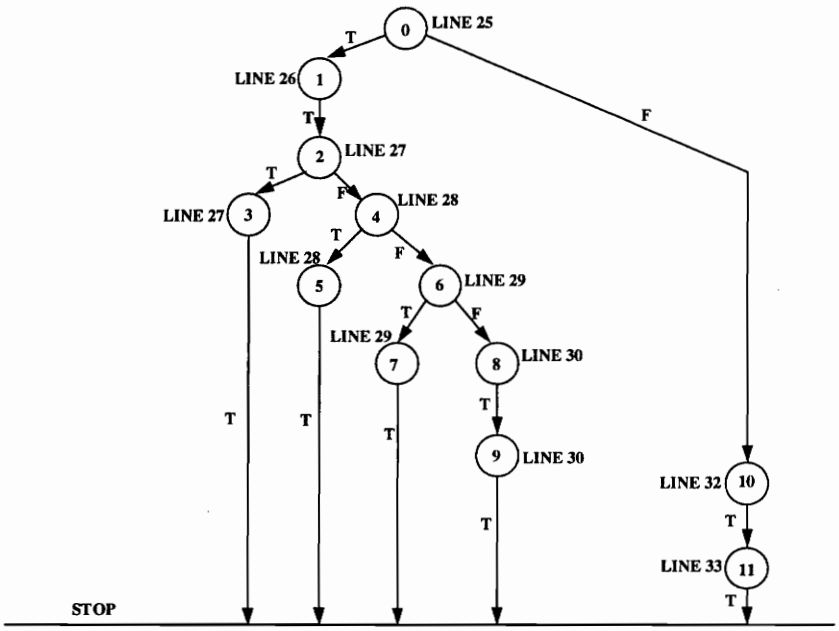


Figure 74. Control Flow Graph for the model MUX

portorder: DATA_OUT D C B A SEL ENB

1
D1 D2 D3 D1 D4 C1 1
1
D4 D2 D3 D1 D4 C0 R
1
D1 D2 D3 D1 D4 C1 R
1
D3 D2 D3 D1 D4 C2 R
1
D2 D2 D3 D1 D4 C3 R
1
D0 D2 D3 D1 D4 C3 F
2
D1 D2 D3 D1 D4 C1 1
D1 D2 D3 D1 D4 C1 1
2
D4 D2 D3 D1 D4 C0 R
D4 D2 D3 D1 D4 C0 1
2
D1 D2 D3 D1 D4 C1 R
D1 D2 D3 D1 D4 C1 1
2
D3 D2 D3 D1 D4 C2 R
D3 D2 D3 D1 D4 C2 1
2
D2 D2 D3 D1 D4 C3 R
D2 D2 D3 D1 D4 C3 1
2
D0 D2 D3 D1 D4 C3 F
D0 D2 D3 D1 D4 C3 0

Figure 75. Process Test Results for the circuit MUX

Circuit: DFF

```
Levels  LINE # |----+----1----+----2----+----3----+----4----+----5----+----6-
         1 |
         2 |use WORK.VHDLCAD.all, WORK.USER_TYPES.all;
         3 |-- ************************************************************
         4 |entity DFF is
1        5 | port (SET_BAR: in BIT;
1        6 |   CLR_BAR: in BIT;
1        7 |   Q_BAR: out BIT;
1        8 |   Q: out BIT;
1        9 |   CLK: in BIT;
1       10 |   D: in BIT);
1       11 |end DFF;
        12 |-- ************************************************************
        13 |
        14 |architecture BEHAVIORAL of DFF is
1       15 |
1       16 |begin
1       17 |
1       18 | -- ---------------------------------------------------
1       19 | -- Process Name: DFF
1       20 | -- ---------------------------------------------------
1       21 |
1       22 | DFF_2: process (SET_BAR,CLR_BAR,CLK)
2       23 | begin
2       24 |   if CLR_BAR = '0' then
2 1     25 |     Q <= '0';
2 1     26 |     Q_BAR <= '1';
2 1     27 |   elsif SET_BAR = '0' then
2 1     28 |     Q <= '1';
2 1     29 |     Q_BAR <= '0';
2 1     30 |   elsif CLK = '1' then
2 1     31 |     Q <= D;
2 1     32 |     Q_BAR <= not D;
2 1     33 |   end if;
2       34 |
2       35 |
2       36 |
2       37 | end process DFF_2;
1       38 |
1       39 |
1       40 |end BEHAVIORAL;
```

Figure 76. The VHDL behavioral model of the circuit DFF

THE Control Flow Graph (CFG) Nodes and the edges are: .....

CFG Node # corresponding to statement # 24 is: 0
.... its True_edge points to node #: 1(line# 25)
.... its False_edge points to node #: 3(line# 27)

CFG Node # corresponding to statement # 25 is: 1
.... its True_edge points to node #: 2(line# 26)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 26 is: 2
.... its True_edge is UnDefined
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 27 is: 3
.... its True_edge points to node #: 4(line# 28)
.... its False_edge points to node #: 6(line# 30)

CFG Node # corresponding to statement # 28 is: 4
.... its True_edge points to node #: 5(line# 29)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 29 is: 5
.... its True_edge is UnDefined
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 30 is: 6
.... its True_edge points to node #: 7(line# 31)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 31 is: 7
.... its True_edge points to node #: 8(line# 32)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 32 is: 8
.... its True_edge is UnDefined
.... its False_edge is UnDefined

Figure 77. The CFG specifications for the model DFF

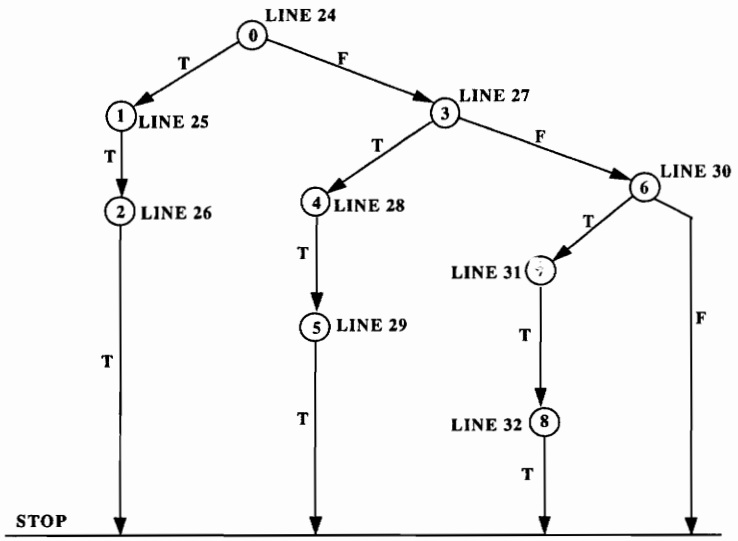


Figure 78. Control Flow Graph for the model DFF

---

portorder: SET_BAR CLR_BAR Q_BAR Q CLK D

1
F 1 0 R 0 1
1
1 F R 0 0 0
1
1 1 0 R R 1
2
F 1 0 R 0 1
0 1 0 1 0 1
2
1 F R 0 0 0
1 0 1 0 0 0
2
1 1 0 R R 1
1 1 0 1 1 1

Figure 79. Process Test Results for the circuit DFF

Circuit: ADDER

```
Levels  LINE # |----+----1----+----2----+----3----+----4----+----5----+----6-
         1 |
         2 |use WORK.VHDLCAD.all, WORK.USER_TYPES.all;
         3 |-- ************************************************************
         4 |entity ADDER is
1        5 |  port (SUM: out BIT;
1        6 |    COUT: out BIT;
1        7 |    CIN: in BIT;
1        8 |    B: in BIT;
1        9 |    A: in BIT);
1       10 |end ADDER;
        11 |-- ************************************************************
        12 |
        13 |architecture BEHAVIORAL of ADDER is
1       14 |
1       15 |begin
1       16 |
1       17 |  -- ---------------------------------------------------
1       18 |  -- Process Name: ADDER
1       19 |  -- ---------------------------------------------------
1       20 |
1       21 |  ADDER_2: process (CIN,B,A)
2       22 |  begin
2       23 |     SUM <= A xor B xor CIN;
2       24 |     COUT <= (A and B) or (A and CIN) or (B and CIN);
2       25 |
2       26 |
2       27 |  end process ADDER_2;
1       28 |
1       29 |
1       30 |end BEHAVIORAL;
```

Figure 80. The VHDL behavioral model of the circuit ADDER

THE Control Flow Graph (CFG) Nodes and the edges are: .....

CFG Node # corresponding to statement # 23 is: 0
.... its True_edge points to node #: 1(line# 24)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 24 is: 1
.... its True_edge is UnDefined
.... its False_edge is UnDefined

Figure 81. The CFG specifications for the model ADDER

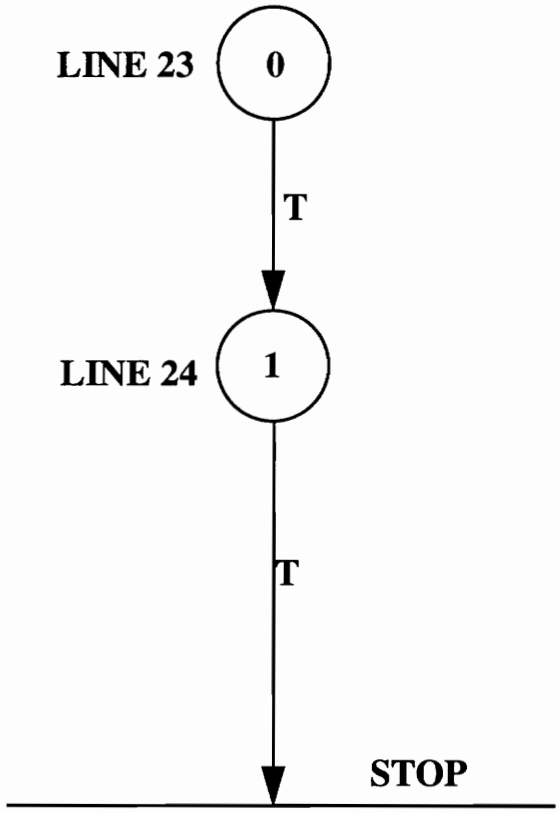


Figure 82. Control Flow Graph for the model ADDER

portorder: SUM COUT CIN B A

1
0 R F 1 1
1
R 1 R 1 1
1
R 0 0 F 1
1
F R 1 R 0
1
0 R 1 1 F
1
R 0 0 0 R
2
0 R F 1 1
0 1 0 1 1
2
R 1 R 1 1
1 1 1 1 1
2
R 0 0 F 1
1 0 0 0 1
2
F R 1 R 0
0 1 1 1 0
2
0 R 1 1 F
0 1 1 1 0
2
R 0 0 0 R
1 0 0 0 1

Figure 83. Process Test Results for the circuit ADDER

Circuit: COUNT

```
Levels  LINE # |----+----1----+----2----+----3----+----4----+----5----+----6-
         1 |
         2 |use WORK.VHDLCAD.all, WORK.USER_TYPES.all;
         3 |-- ************************************************************
         4 |entity Count is
1        5 |  port (CNT: inout BIT_VECTOR(3 downto 0);
1        6 |    DATA_IN: in BIT_VECTOR(3 downto 0);
1        7 |    UP: in BIT;
1        8 |    COUNT: in BIT;
1        9 |    LOAD: in BIT;
1       10 |    RESET: in BIT;
1       11 |    CLK: in BIT);
1       12 |end Count;
        13 |-- ************************************************************
        14 |
        15 |architecture BEHAVIORAL of Count is
1       16 |
1       17 |begin
1       18 |
1       19 |  -- ---------------------------------------------------
1       20 |  -- Process Name: Count
1       21 |  -- ---------------------------------------------------
1       22 |
1       23 |  Count_2: process (CLK)
2       24 |  begin
2       25 |     if CLK = '1' then
2 1     26 |       if RESET = '1' then
2 2     27 |         CNT <= "0000";
2 2     28 |       elsif LOAD = '1' then
2 2     29 |         CNT <= DATA_IN;
2 2     30 |       elsif COUNT = '1' then
2 2     31 |         if UP = '1' then
2 3     32 |           CNT <= INC(CNT);
2 3     33 |         else
2 3     34 |           CNT <= DEC(CNT);
2 3     35 |         end if;
2 2     36 |       end if;
2 1     37 |     end if;
2       38 |
2       39 |
```

```
2       40 |  end process Count_2;
1       41 |
1       42 |
1       43 |end BEHAVIORAL;
```

Figure 84. The VHDL behavioral model of the circuit COUNT

THE Control Flow Graph (CFG) Nodes and the edges are .....

CFG Node # corresponding to statement # 25 is: 0
.... its True_edge points to node #: 1(line# 26)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 26 is: 1
.... its True_edge points to node #: 2(line# 27)
.... its False_edge points to node #: 3(line# 28)

CFG Node # corresponding to statement # 27 is: 2
.... its True_edge is UnDefined
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 28 is: 3
.... its True_edge points to node #: 4(line# 29)
.... its False_edge points to node #: 5(line# 30)

CFG Node # corresponding to statement # 29 is: 4
.... its True_edge is UnDefined
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 30 is: 5
.... its True_edge points to node #: 6(line# 31)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 31 is: 6
.... its True_edge points to node #: 7(line# 32)
.... its False_edge points to node #: 8(line# 33)

CFG Node # corresponding to statement # 32 is: 7
.... its True_edge is UnDefined
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 33 is: 8
.... its True_edge points to node #: 9(line# 34)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 34 is: 9
.... its True_edge is UnDefined
.... its False_edge is UnDefined

Figure 85. The CFG specifications for the model COUNT

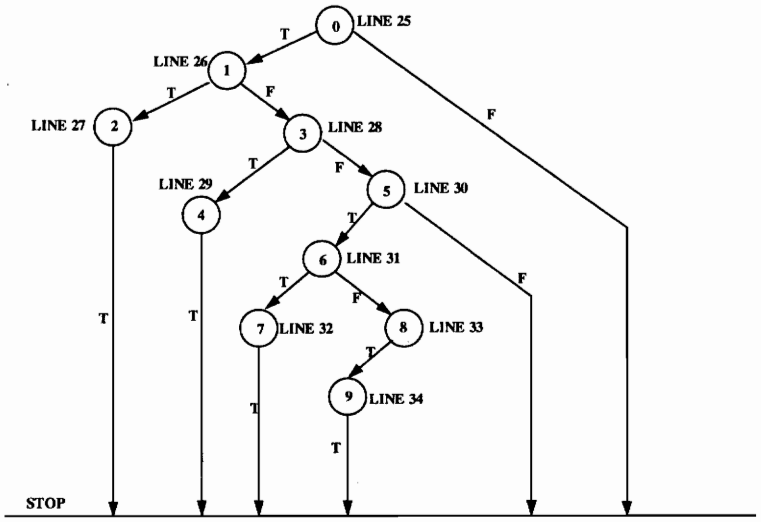


Figure 86. Control Flow Graph for the model COUNT

---

portorder: CNT DATA_IN UP COUNT LOAD RESET CLK

1
C0 D9 1 0 0 1 R
1
C7 D7 0 0 1 0 R
1
CN D8 1 1 0 0 R
1
CP D15 0 1 0 0 R
2
C0 D9 1 0 0 1 R
C0 D9 1 0 0 1 1
2
C7 D7 0 0 1 0 R
C7 D7 0 0 1 0 1
2
CN D8 1 1 0 0 R
CN D8 1 1 0 0 1
2
CP D15 0 1 0 0 R
CP D15 0 1 0 0 1

Figure 87. Process Test Results for the circuit COUNT

Circuit: JKFF

```
Levels  LINE # |----+----1----+----2----+----3----+----4----+----5----+----6-
          1 |
          2 |use WORK.VHDLCAD.all, WORK.USER_TYPES.all;
          3 |-- ************************************************************
          4 |entity JKFF is
1         5 |  port (NOT_Q: inout BIT;
1         6 |    Q: inout BIT;
1         7 |    K: in BIT;
1         8 |    J: in BIT;
1         9 |    S: in BIT;
1        10 |    R: in BIT;
1        11 |    CLK: in BIT);
1        12 |end JKFF;
         13 |-- ************************************************************
         14 |
         15 |architecture BEHAVIORAL of JKFF is
1        16 |
1        17 |begin
1        18 |
1        19 |  -- ---------------------------------------------------
1        20 |  -- Process Name: JKFF
1        21 |  -- ---------------------------------------------------
1        22 |
1        23 |  JKFF_2: process (S,R,CLK)
2        24 |  begin
2        25 |    if CLK = '1' then
2 1      26 |      if S='0' and R='0' then
2 2      27 |        if J='1' and K='0' then
2 3      28 |          Q <= '1';
2 3      29 |          NOT_Q <= '0';
2 3      30 |        elsif J='0' and K='1' then
2 3      31 |          Q <= '0';
2 3      32 |          NOT_Q <= '1';
2 3      33 |        elsif J='1' and K='1' then
2 3      34 |          Q <= not Q;
2 3      35 |          NOT_Q <= not NOT_Q;
2 3      36 |        end if;
2 2      37 |      end if;
2 1      38 |    elsif S='1' and R='0' then
2 1      39 |      Q <= '1';
2 1      40 |      NOT_Q <= '0';
```

```
2 1      41 |    elsif S='0' and R='1' then
2 1      42 |      Q <= '0';
2 1      43 |      NOT_Q <= '1';
2 1      44 |    end if;
2        45 |
2        46 |
2        47 | end process JKFF_2;
1        48 |
1        49 |
1        50 |end BEHAVIORAL;
```

Figure 88. The VHDL behavioral model of the circuit JKFF

```
THE Control Flow Graph (CFG) Nodes and the edges are: .....

CFG Node # corresponding to statement # 25 is: 0
.... its True_edge points to node #: 1(line# 26)
.... its False_edge points to node #: 11(line# 38)

CFG Node # corresponding to statement # 26 is: 1
.... its True_edge points to node #: 2(line# 27)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 27 is: 2
.... its True_edge points to node #: 3(line# 28)
.... its False_edge points to node #: 5(line# 30)

CFG Node # corresponding to statement # 28 is: 3
.... its True_edge points to node #: 4(line# 29)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 29 is: 4
.... its True_edge is UnDefined
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 30 is: 5
.... its True_edge points to node #: 6(line# 31)
.... its False_edge points to node #: 8(line# 33)
```

CFG Node # corresponding to statement # 31 is: 6
.... its True_edge points to node #: 7(line# 32)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 32 is: 7
.... its True_edge is UnDefined
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 33 is: 8
.... its True_edge points to node #: 9(line# 34)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 34 is: 9
.... its True_edge points to node #: 10(line# 35)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 35 is: 10
.... its True_edge is UnDefined
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 38 is: 11
.... its True_edge points to node #: 12(line# 39)
.... its False_edge points to node #: 14(line# 41)

CFG Node # corresponding to statement # 39 is: 12
.... its True_edge points to node #: 13(line# 40)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 40 is: 13
.... its True_edge is UnDefined
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 41 is: 14
.... its True_edge points to node #: 15(line# 42)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 42 is: 15
.... its True_edge points to node #: 16(line# 43)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 43 is: 16
.... its True_edge is UnDefined
.... its False_edge is UnDefined

Figure 89. The CFG specifications for the model JKFF

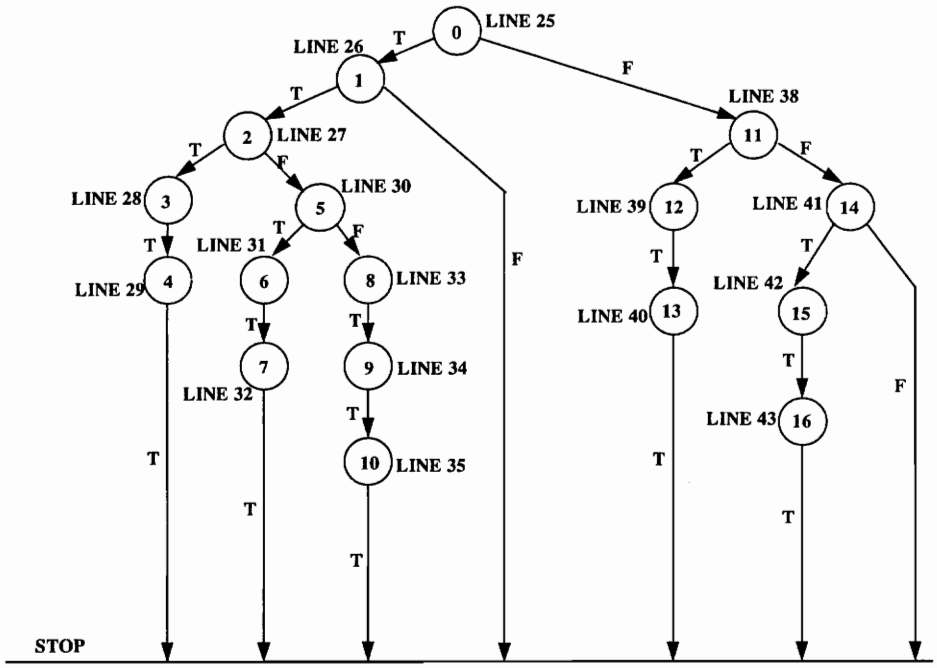


Figure 90. Control Flow Graph for the model JKFF

portorder: NOT_Q Q K J S R CLK

1
0 R 0 1 F 0 1
1
0 R 1 1 1 F 0
1
0 R 0 1 0 0 R
1
R 0 1 0 0 0 R
1
NP NP 1 1 0 0 R
1
R 0 1 1 0 1 F
2
0 R 0 1 F 0 1
0 1 0 1 0 0 1
2
0 R 1 1 1 F 0
0 1 1 1 1 0 0
2
0 R 0 1 0 0 R
0 1 0 1 0 0 1
2
R 0 1 0 0 0 R
1 0 1 0 0 0 1
2
NP NP 1 1 0 0 R
NP NP 1 1 0 0 1
2
R 0 1 1 0 1 F
1 0 1 1 0 1 0

Figure 91. Process Test Results for the circuit JKFF

Circuit: CKTC

```
Levels  LINE # |----+----1----+----2----+----3----+----4----+----5----+----6-
          1 |
          2 |use WORK.VHDLCAD.all, WORK.USER_TYPES.all;
          3 |-- ************************************************************
          4 |entity BIG is
1         5 |  port (C: out BIT_VECTOR(7 downto 0);
1         6 |    DATA: in BIT_VECTOR(7 downto 0);
1         7 |    B: in BIT;
1         8 |    A: in BIT;
1         9 |    SEL: in BIT_VECTOR(1 downto 0);
1        10 |    ENB: in BIT);
1        11 |end BIG;
         12 |-- ************************************************************
         13 |
         14 |architecture BEHAVIORAL of BIG is
1        15 |
1        16 |begin
1        17 |
1        18 |  -- ---------------------------------------------------
1        19 |  -- Process Name: BIG
1        20 |  -- ---------------------------------------------------
1        21 |
1        22 |  BIG_3: process (B,A,SEL,ENB)
2        23 |     variable C_VAR: BIT;
2        24 |  begin
2        25 |     if ENB = '1' then
2 1      26 |        case SEL is
2 2      27 |             when "00" => C_VAR := A;
2 2      28 |             when "01" => C_VAR := B;
2 2      29 |             when "10" => C_VAR := A or B;
2 2      30 |             when "11" => C_VAR := A and B;
2 2      31 |        end case;
2 1      32 |
2 1      33 |      if C_VAR = '1' then
2 2      34 |           C <= DATA;
2 2      35 |      else
2 2      36 |           C <= "00000000";
2 2      37 |      end if;
2 1      38 |     end if;
2        39 |
2        40 |
```

```
2       41 |
2       42 |  end process BIG_3;
1       43 |
1       44 |
1       45 |end BEHAVIORAL;
```

Figure 92. The VHDL behavioral model of the circuit CKTC

THE Control Flow Graph (CFG) Nodes and the edges are: .....

CFG Node # corresponding to statement # 25 is: 0
.... its True_edge points to node #: 1(line# 26)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 26 is: 1
.... its True_edge points to node #: 2(line# 27)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 27 is: 2
.... its True_edge points to node #: 3(line# 27)
.... its False_edge points to node #: 4(line# 28)

CFG Node # corresponding to statement # 27 is: 3
.... its True_edge points to node #: 10(line# 33)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 28 is: 4
.... its True_edge points to node #: 5(line# 28)
.... its False_edge points to node #: 6(line# 29)

CFG Node # corresponding to statement # 28 is: 5
.... its True_edge points to node #: 10(line# 33)

.... its False_edge is UnDefined

CFG Node # corresponding to statement # 29 is: 6
.... its True_edge points to node #: 7(line# 29)
.... its False_edge points to node #: 8(line# 30)

CFG Node # corresponding to statement # 29 is: 7
.... its True_edge points to node #: 10(line# 33)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 30 is: 8
.... its True_edge points to node #: 9(line# 30)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 30 is: 9
.... its True_edge points to node #: 10(line# 33)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 33 is: 10
.... its True_edge points to node #: 11(line# 34)
.... its False_edge points to node #: 12(line# 35)

CFG Node # corresponding to statement # 34 is: 11
.... its True_edge is UnDefined
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 35 is: 12
.... its True_edge points to node #: 13(line# 36)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 36 is: 13
.... its True_edge is UnDefined
.... its False_edge is UnDefined

Figure 93. The CFG specifications for the model CKTC

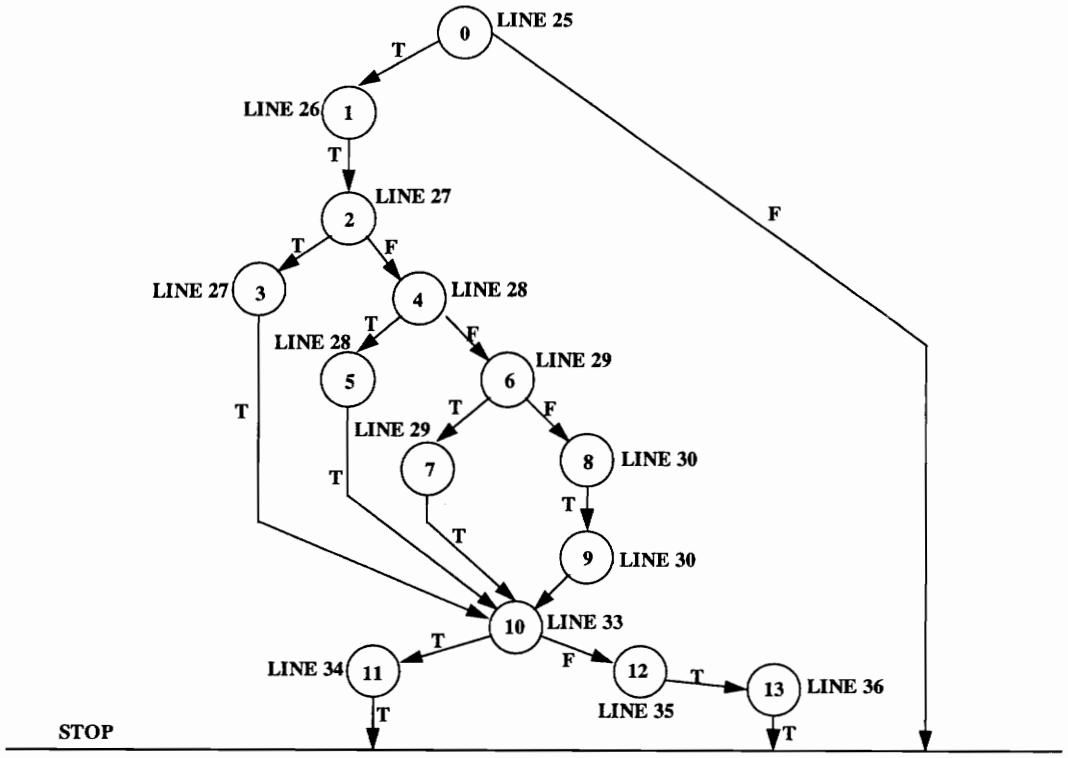


Figure 94. Control Flow Graph for the model CKTC

portorder: C DATA B A SEL ENB

1
D0 D1 F 0 C1 1
1
D2 D2 R 1 C1 1
1
D0 D3 1 F C0 1
1
D4 D4 0 R C0 1
1
D5 D5 1 0 C1 1
1
D0 D6 0 0 C0 R
1
D3 D3 0 1 C2 R
1
D7 D7 1 1 C3 R
1
D0 D8 0 0 C0 R
2
D0 D1 F 0 C1 1
D0 D1 0 0 C1 1
2
D2 D2 R 1 C1 1
D2 D2 1 1 C1 1
2
D0 D3 1 F C0 1
D0 D3 1 0 C0 1
2
D4 D4 0 R C0 1
D4 D4 0 1 C0 1
2
D5 D5 1 0 C1 1
D5 D5 1 0 C1 1
2
D0 D6 0 0 C0 R
D0 D6 0 0 C0 1
2
D3 D3 0 1 C2 R
D3 D3 0 1 C2 1
2
D7 D7 1 1 C3 R
D7 D7 1 1 C3 1

2
D0 D8 0 0 C0 R
D0 D8 0 0 C0 1

Figure 95. Process Test Results for the model CKTC

Circuit: CCNT3

```
Levels  LINE # |----+----1----+----2----+----3----+----4----+----5----+----6-
         1 |
         2 |use WORK.VHDLCAD.all, WORK.USER_TYPES.all;
         3 |-- ************************************************************
         4 |entity CCNT3 is
1        5 |  port (CNT: inout BIT_VECTOR(2 downto 0);
1        6 |    DATA_IN: in BIT_VECTOR(2 downto 0);
1        7 |    CLK: in BIT;
1        8 |    CON: in BIT_VECTOR(1 downto 0);
1        9 |    STRB: in BIT);
1       10 |end CCNT3;
        11 |-- ************************************************************
        12 |
        13 |architecture BEHAVIORAL of CCNT3 is
1       14 |
1       15 |begin
1       16 |
1       17 |  -- ---------------------------------------------------
1       18 |  -- Process Name: CCNT3
1       19 |  -- ---------------------------------------------------
1       20 |
1       21 |  CCNT3_2: process (CLK,CON,STRB)
2       22 |     variable UP: BIT;
2       23 |     variable COUNT: BIT;
2       24 |     variable LD: BIT;
2       25 |     variable RST: BIT;
2       26 |  begin
2       27 |     if STRB = '1' then
2 1     28 |       if CLK = '1' then
2 2     29 |          case CON is
2 3     30 |                when "00" => COUNT := '1';
2 3     31 |                        UP := '1';
2 3     32 |                when "01" => COUNT := '1';
2 3     33 |                        UP := '0';
2 3     34 |                when "10" => LD := '1';
2 3     35 |                when "11" => RST  := '1';
2 3     36 |          end case;
2 2     37 |
2 2     38 |
2 2     39 |         if RST = '1' then
2 3     40 |           CNT <= "000";
```

```
2 3     41 |       elsif LD = '1' then
2 3     42 |         CNT <= DATA_IN;
2 3     43 |       elsif COUNT = '1' then
2 3     44 |         if UP = '1' then
2 4     45 |           CNT <= INC(CNT);
2 4     46 |         else
2 4     47 |           CNT <= DEC(CNT);
2 4     48 |         end if;
2 3     49 |       end if;
2 2     50 |     end if;
2 1     51 |   end if;
2       52 |
2       53 |
2       54 | end process CCNT3_2;
```

Figure 96. The VHDL behavioral model of the circuit CCNT3

THE Control Flow Graph (CFG) Nodes and the edges are: .....

CFG Node # corresponding to statement # 27 is: 0
.... its True_edge points to node #: 1(line# 28)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 28 is: 1
.... its True_edge points to node #: 2(line# 29)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 29 is: 2
.... its True_edge points to node #: 3(line# 30)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 30 is: 3
.... its True_edge points to node #: 4(line# 30)
.... its False_edge points to node #: 6(line# 32)

CFG Node # corresponding to statement # 30 is: 4
.... its True_edge points to node #: 5(line# 31)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 31 is: 5
.... its True_edge points to node #: 13(line# 39)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 32 is: 6
.... its True_edge points to node #: 7(line# 32)
.... its False_edge points to node #: 9(line# 34)

CFG Node # corresponding to statement # 32 is: 7
.... its True_edge points to node #: 8(line# 33)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 33 is: 8
.... its True_edge points to node #: 13(line# 39)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 34 is: 9
.... its True_edge points to node #: 10(line# 34)
.... its False_edge points to node #: 11(line# 35)

CFG Node # corresponding to statement # 34 is: 10
.... its True_edge points to node #: 13(line# 39)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 35 is: 11
.... its True_edge points to node #: 12(line# 35)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 35 is: 12
.... its True_edge points to node #: 13(line# 39)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 39 is: 13
.... its True_edge points to node #: 14(line# 40)
.... its False_edge points to node #: 15(line# 41)

CFG Node # corresponding to statement # 40 is: 14
.... its True_edge is UnDefined
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 41 is: 15
.... its True_edge points to node #: 16(line# 42)
.... its False_edge points to node #: 17(line# 43)

CFG Node # corresponding to statement # 42 is: 16
.... its True_edge is UnDefined
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 43 is: 17
.... its True_edge points to node #: 18(line# 44)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 44 is: 18
.... its True_edge points to node #: 19(line# 45)
.... its False_edge points to node #: 20(line# 46)

CFG Node # corresponding to statement # 45 is: 19
.... its True_edge is UnDefined
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 46 is: 20
.... its True_edge points to node #: 21(line# 47)
.... its False_edge is UnDefined

CFG Node # corresponding to statement # 47 is: 21
.... its True_edge is UnDefined
.... its False_edge is UnDefined

Figure 97. The CFG specifications for the model CCNT3

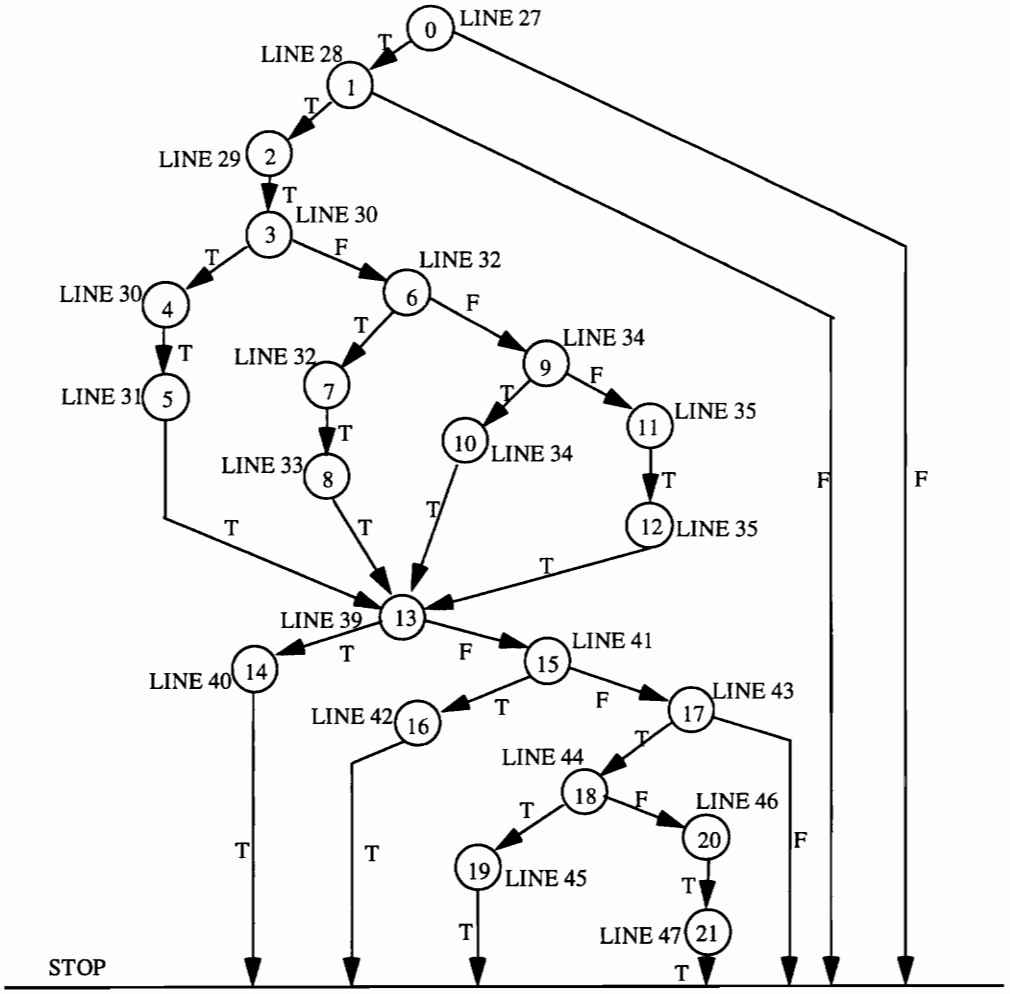


Figure 98. Control Flow Graph for the model CCNT3

---

portorder: CNT DATA_IN CLK CON STRB

1
CN D5 R C0 1
1
CP D4 1 C1 1
1
CN D3 1 C0 R
1
CP D2 1 C1 R
1
C7 D7 1 C2 R
1
C0 D3 1 C3 R
1
CN D5 1 C0 R
1
CN D6 1 C0 R
1
CN D1 1 C0 R
2
CN D5 R C0 1
CN D5 1 C0 1
2
CP D4 1 C1 1
CP D4 1 C1 1
2
CN D3 1 C0 R
CN D3 1 C0 1
2
CP D2 1 C1 R
CP D2 1 C1 1
2
C7 D7 1 C2 R
C7 D7 1 C2 1
2
C0 D3 1 C3 R
C0 D3 1 C3 1

2
CN D5 1 C0 R
CN D5 1 C0 1
2
CN D6 1 C0 R
CN D6 1 C0 1
2
CN D1 1 C0 R
CN D1 1 C0 1

Figure 99. Process Test Results for the circuit CCNT3

Test Bench Generation Examples

In this section, two examples are given that illustrate the generation of a VHDL Test Bench from an HBTG test sequence by the Test Bench Generator (TBG) program. For both the examples, the VHDL source, the HBTG test results and the Test Bench generated by TBG are shown.

Model: MUX-REG

```
use WORK.VHDLCAD.all, WORK.USER_TYPES.all;
-- ************************************************************
entity eg7 is
  port (NI: in BIT;
    D_OUT: out BIT_VECTOR(7 downto 0);
    LD: in BIT;
    CLK: in BIT;
    C2: in BIT;
    C1: in BIT;
    IN2: in BIT_VECTOR(7 downto 0);
    IN1: in BIT_VECTOR(7 downto 0));
end eg7;
-- ************************************************************

architecture BEHAVIORAL of eg7 is

  signal SEL: BIT;
  signal CLEAR: BIT;
  signal DATA: BIT_VECTOR(7 downto 0);
begin

  -- -----------------------------------------------------
  -- Process Name: INV
  -- -----------------------------------------------------
```

```
INV_4: process (NI)
begin
   SEL <= not NI;


end process INV_4;


-- ---------------------------------------------------
-- Process Name: Reg
-- ---------------------------------------------------

Reg_8: process (CLEAR,CLK)
begin
   if CLEAR = '1' then
         D_OUT <= "00000000";
   elsif CLK'EVENT and CLK = '1' then
     if LD = '1' then
         D_OUT <= DATA;
     end if;
   end if;


end process Reg_8;


-- ---------------------------------------------------
-- Process Name: AND
-- ---------------------------------------------------

AND_15: process (C2,C1)
begin
   CLEAR <= C1 and C2;


end process AND_15;


-- ---------------------------------------------------
-- Process Name: Mux
-- ---------------------------------------------------

Mux_20: process (SEL)
begin
   if SEL = '0' then
```

```
    DATA <= IN1;
  else
    DATA <= IN2;
  end if;


 end process Mux_20;


end BEHAVIORAL;
```

Figure 100. VHDL source for a PMG MUX-REG

```
Enter name of the unit:
Unit name is eg7
No. of signals = 11

Sensitive path 0:
sp[0][0]=7(NI)
sp[0][1]=5(NO)
sp[0][2]=29(S)
sp[0][3]=24(OP)
sp[0][4]=14(D_IN)
sp[0][5]=10(D_OUT)


Sensitive path 1:
sp[1][0]=15(CLK)
sp[1][1]=10(D_OUT)


Sensitive path 2:
sp[2][0]=20(C2)
sp[2][1]=18(OUP)
sp[2][2]=12(CLR)
sp[2][3]=10(D_OUT)

Sensitive path 3:
sp[3][0]=21(C1)
sp[3][1]=18(OUP)
sp[3][2]=12(CLR)
```

sp[3][3]=10(D_OUT)

No. of Spath = 4

A complete test sequence for model eg7:

frame	*sig(0)*	*sig(1)*	*sig(2)*	*sig(3)*	*sig(4)*	*sig(5)*	*sig(6)*	*sig(7)*	*sig(8)*	*sig(9)*	*sig(10)*
0	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*
2	*X*	*X*	*X*	*X*	*X*	*F*	*1*	*F*	*X*	*X*	*X*
1	*R*	*F*	*D2*	*1*	*F*	*0*	*1*	*0*	*D2*	*D2*	*D3*
3	*1*	*0*	*D2*	*1*	*F*	*0*	*1*	*0*	*D2*	*D2*	*D3*
4	*1*	*0*	*D1*	*1*	*1*	*R*	*R*	*1*	*D2*	*D2*	*D3*
5	*1*	*0*	*D1*	*1*	*1*	*R*	*1*	*R*	*D2*	*D2*	*D3*

sig(0) = NO
sig(1) = NI
sig(2) = D_OUT
sig(3) = LD
sig(4) = CLK
sig(5) = OUP
sig(6) = C2
sig(7) = C1
sig(8) = OP
sig(9) = IN2
sig(10) = IN1

PortAct(7) = 1
PortAct(12) = 2 PortAct(15) = 1
PortAct(20) = 1 PortAct(21) = 2
PortAct(29) = 1

Figure 101. Test Results generated by HBTG for MUX-REG

```
use work.VHDLCAD.all, work.all;

entity EG7_TEST_BENCH is
end EG7_TEST_BENCH;

architecture BEHAVIOR of EG7_TEST_BENCH is
signal NI, LD, CLK, C2, C1: BIT;
signal D_OUT, IN2, IN1: BIT_VECTOR(7 downto 0);

component EG7_A
port( NI: in BIT; D_OUT: out BIT_VECTOR(7 downto 0); LD: in BIT; CLK: in BIT; C2: in
BIT; C1: in BIT; IN2: in BIT_VECTOR(7 downto 0); IN1: in BIT_VECTOR(7 downto 0));
end component;

for all: EG7_A use entity work.EG7(BEHAVIORAL);

begin

R1: EG7_A
  port map(NI, D_OUT, LD, CLK, C2, C1, IN2, IN1);

 process
  begin


  NI <= transport '0' after 1 ns;
  LD <= transport '0' after 1 ns;
  CLK <= transport '0' after 1 ns;
  C2 <= transport '0' after 1 ns;
  C1 <= transport '0' after 1 ns;
  IN2 <= transport "00000000" after 1 ns;
  IN1 <= transport "00000000" after 1 ns;

  NI <= transport '0' after 4 ns;
  LD <= transport '0' after 4 ns;
  CLK <= transport '0' after 4 ns;
  C2 <= transport '0' after 4 ns;
  C1 <= transport '0' after 4 ns;
  IN2 <= transport "00000000" after 4 ns;
  IN1 <= transport "00000000" after 4 ns;

  NI <= transport '0' after 6 ns;
  LD <= transport '0' after 6 ns;
  CLK <= transport '0' after 6 ns;
```

```
C2 <= transport '1' after 6 ns;
C1 <= transport '1' after 6 ns;
IN2 <= transport "00000000" after 6 ns;
IN1 <= transport "00000000" after 6 ns;

NI <= transport '0' after 9 ns;
LD <= transport '0' after 9 ns;
CLK <= transport '0' after 9 ns;
C2 <= transport '1' after 9 ns;
C1 <= transport '0' after 9 ns;
IN2 <= transport "00000000" after 9 ns;
IN1 <= transport "00000000" after 9 ns;

NI <= transport '1' after 11 ns;
LD <= transport '1' after 11 ns;
CLK <= transport '1' after 11 ns;
C2 <= transport '1' after 11 ns;
C1 <= transport '0' after 11 ns;
IN2 <= transport "11110000" after 11 ns;
IN1 <= transport "00001111" after 11 ns;

NI <= transport '0' after 14 ns;
LD <= transport '1' after 14 ns;
CLK <= transport '0' after 14 ns;
C2 <= transport '1' after 14 ns;
C1 <= transport '0' after 14 ns;
IN2 <= transport "11110000" after 14 ns;
IN1 <= transport "00001111" after 14 ns;

NI <= transport '0' after 16 ns;
LD <= transport '1' after 16 ns;
CLK <= transport '1' after 16 ns;
C2 <= transport '1' after 16 ns;
C1 <= transport '0' after 16 ns;
IN2 <= transport "11110000" after 16 ns;
IN1 <= transport "00001111" after 16 ns;

NI <= transport '0' after 19 ns;
LD <= transport '1' after 19 ns;
CLK <= transport '0' after 19 ns;
C2 <= transport '1' after 19 ns;
C1 <= transport '0' after 19 ns;
IN2 <= transport "11110000" after 19 ns;
IN1 <= transport "00001111" after 19 ns;

NI <= transport '0' after 21 ns;
```

```
    LD <= transport '1' after 21 ns;
    CLK <= transport '1' after 21 ns;
    C2 <= transport '0' after 21 ns;
    C1 <= transport '1' after 21 ns;
    IN2 <= transport "11110000" after 21 ns;
    IN1 <= transport "00001111" after 21 ns;

    NI <= transport '0' after 24 ns;
    LD <= transport '1' after 24 ns;
    CLK <= transport '1' after 24 ns;
    C2 <= transport '1' after 24 ns;
    C1 <= transport '1' after 24 ns;
    IN2 <= transport "11110000" after 24 ns;
    IN1 <= transport "00001111" after 24 ns;

    NI <= transport '0' after 26 ns;
    LD <= transport '1' after 26 ns;
    CLK <= transport '1' after 26 ns;
    C2 <= transport '1' after 26 ns;
    C1 <= transport '0' after 26 ns;
    IN2 <= transport "11110000" after 26 ns;
    IN1 <= transport "00001111" after 26 ns;

    NI <= transport '0' after 29 ns;
    LD <= transport '1' after 29 ns;
    CLK <= transport '1' after 29 ns;
    C2 <= transport '1' after 29 ns;
    C1 <= transport '1' after 29 ns;
    IN2 <= transport "11110000" after 29 ns;
    IN1 <= transport "00001111" after 29 ns;

    wait;

  end process;
end BEHAVIOR;
```

Figure 102. Test Bench generated by TBG for MUX-REG

Model: BUF_LATCH

```
use WORK.VHDLCAD.all, WORK.USER_TYPES.all;
-- ************************************************************
entity Buf_latch is
  port (DATA: in BIT_VECTOR(0 to 7);
    CLK: in BIT;
    DO: out BIT_VECTOR(0 to 7);
    NDS2: in BIT;
    DS1: in BIT);
end Buf_latch;
-- ************************************************************

architecture BEHAVIORAL of Buf_latch is

  signal Buf: BIT_VECTOR(0 to 7);
  signal ENBLD: BIT;
begin

  -- ------------------------------------------------------
  -- Process Name: Ltch
  -- ------------------------------------------------------

  Ltch_4: process (DATA,CLK)
  begin
    if CLK='1' then
      Buf <= Buf;
    else
      Buf <= DATA;
    end if;


  end process Ltch_4;


  -- ------------------------------------------------------
  -- Process Name: Buff
  -- ------------------------------------------------------

  Buff_9: process (ENBLD,Buf)
  begin
```

```
    if (ENBLD='1') then
       DO <= Buf;
    else
       DO <= "11111111";
    end if;


 end process Buff_9;


 -- ------------------------------------------------------
 -- Process Name: Enable
 -- ------------------------------------------------------

 Enable_14: process (NDS2,DS1)
 begin
    ENBLD <= DS1 and not NDS2;


 end process Enable_14;


end BEHAVIORAL;
```

Figure 103. VHDL source for a PMG BUF_LATCH

```
Enter name of the unit:
Unit name is Buf_latch
No. of signals = 7

Sensitive path 0:
sp[0][0]=8(DATA)
sp[0][1]=5(OP)
sp[0][2]=15(BUF)
sp[0][3]=12(DO)

Sensitive path 1:
sp[1][0]=9(CLK)
sp[1][1]=5(OP)
sp[1][2]=15(BUF)
sp[1][3]=12(DO)
```

Sensitive path 2:
sp[2][0]=20(NDS2)
sp[2][1]=18(ENBLD)
sp[2][2]=14(ENBLD)
sp[2][3]=12(DO)

Sensitive path 3:
sp[3][0]=21(DS1)
sp[3][1]=18(ENBLD)
sp[3][2]=14(ENBLD)
sp[3][3]=12(DO)

No. of Spath = 4

A complete test sequence for model Buf_latch:

frame	*sig(0)*	*sig(1)*	*sig(2)*	*sig(3)*	*sig(4)*	*sig(5)*	*sig(6)*
0	*D2*	*D2*	*F*	*X*	*X*	*X*	*X*
1	*D2*	*D2*	*0*	*X*	*X*	*X*	*X*
3	*D2*	*D2*	*0*	*D2*	*R*	*F*	*1*
2	*D3*	*D3*	*0*	*D3*	*1*	*0*	*1*
4	*D3*	*D2*	*R*	*D3*	*1*	*0*	*1*
5	*D2*	*D2*	*0*	*D4*	*F*	*R*	*1*
6	*D2*	*D2*	*0*	*D2*	*R*	*0*	*R*

sig(0) = OP
sig(1) = DATA
sig(2) = CLK
sig(3) = DO
sig(4) = ENBLD
sig(5) = NDS2
sig(6) = DS1

PortAct(8) = 1 PortAct(9) = 1
PortAct(14) = 2 PortAct(15) = 2
PortAct(20) = 1 PortAct(21) = 2

Figure 104. Test Results generated by HBTG for BUF_LATCH

```
use work.VHDLCAD.all, work.all;

entity BUF_LATCH_TEST_BENCH is
end BUF_LATCH_TEST_BENCH;

architecture BEHAVIOR of BUF_LATCH_TEST_BENCH is
signal DATA, DO: BIT_VECTOR(0 to 7);
signal CLK, NDS2, DS1: BIT;

component BUF_LATCH_A
port( DATA: in BIT_VECTOR(0 to 7); CLK: in BIT; DO: out BIT_VECTOR(0 to 7); NDS2: in
BIT; DS1: in BIT);
end component;

for all: BUF_LATCH_A use entity work.BUF_LATCH(BEHAVIORAL);

begin

R1: BUF_LATCH_A
  port map(DATA, CLK, DO, NDS2, DS1);

 process
  begin


  DATA <= transport "10011001" after 1 ns;
  CLK <= transport '1' after 1 ns;
  NDS2 <= transport '0' after 1 ns;
  DS1 <= transport '0' after 1 ns;

  DATA <= transport "10011001" after 4 ns;
  CLK <= transport '0' after 4 ns;
  NDS2 <= transport '0' after 4 ns;
  DS1 <= transport '0' after 4 ns;

  DATA <= transport "10011001" after 6 ns;
  CLK <= transport '0' after 6 ns;
  NDS2 <= transport '0' after 6 ns;
  DS1 <= transport '0' after 6 ns;

  DATA <= transport "10011001" after 9 ns;
  CLK <= transport '0' after 9 ns;
  NDS2 <= transport '0' after 9 ns;
  DS1 <= transport '0' after 9 ns;
```

```
DATA <= transport "10011001" after 11 ns;
CLK <= transport '0' after 11 ns;
NDS2 <= transport '1' after 11 ns;
DS1 <= transport '1' after 11 ns;

DATA <= transport "10011001" after 14 ns;
CLK <= transport '0' after 14 ns;
NDS2 <= transport '0' after 14 ns;
DS1 <= transport '1' after 14 ns;

DATA <= transport "10011111" after 16 ns;
CLK <= transport '0' after 16 ns;
NDS2 <= transport '0' after 16 ns;
DS1 <= transport '1' after 16 ns;

DATA <= transport "10011111" after 19 ns;
CLK <= transport '0' after 19 ns;
NDS2 <= transport '0' after 19 ns;
DS1 <= transport '1' after 19 ns;

DATA <= transport "11110000" after 21 ns;
CLK <= transport '0' after 21 ns;
NDS2 <= transport '0' after 21 ns;
DS1 <= transport '1' after 21 ns;

DATA <= transport "11110000" after 24 ns;
CLK <= transport '1' after 24 ns;
NDS2 <= transport '0' after 24 ns;
DS1 <= transport '1' after 24 ns;

DATA <= transport "11110000" after 26 ns;
CLK <= transport '0' after 26 ns;
NDS2 <= transport '0' after 26 ns;
DS1 <= transport '1' after 26 ns;

DATA <= transport "11110000" after 29 ns;
CLK <= transport '0' after 29 ns;
NDS2 <= transport '1' after 29 ns;
DS1 <= transport '1' after 29 ns;

DATA <= transport "11110000" after 31 ns;
CLK <= transport '0' after 31 ns;
NDS2 <= transport '0' after 31 ns;
DS1 <= transport '0' after 31 ns;
```

```
    DATA <= transport "11110000" after 34 ns;
    CLK <= transport '0' after 34 ns;
    NDS2 <= transport '0' after 34 ns;
    DS1 <= transport '1' after 34 ns;
    wait;
  end process;
end BEHAVIOR;
```

Figure 105. Test Bench generated by TBG for BUF_LATCH

Vita

Shekhar Kapoor was born on May 26, 1969 in Bombay, India. He graduated from Saint Francis School, Amritsar, India, in May 1985, and did his High School from D. A. V. College, Amritsar, in June 1987. He entered the Punjab Engineering College, Chandigarh, India, in July 1987, and graduated with a Bachelor of Engineering degree in Electronics and Electrical Communications *(Honors)* in July 1991. He attended graduate school at the Virginia Polytechnic Institute and State University from January 1992 to March 1994, receiving a Master of Science degree in Electrical Engineering in March 1994.

Shekhar is a member of the IEEE and his technical interests include digital design, VLSI circuit design, modeling with VHDL, test generation and computer architecture.