A Hierarchical Approach to Effective Test Generation for VHDL Behavioral Models

by

Sanat R. Rao

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

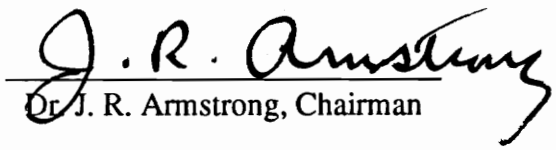
in partial fulfillment of the requirements for the degree of

Master of Science

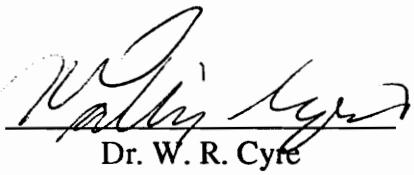
in

Electrical Engineering

APPROVED:



Dr. J. R. Armstrong, Chairman



Dr. W. R. Cyre

Dr. F. G. Gray

May 1993

Blacksburg, Virginia

A Hierarchical Approach to Effective Test Generation for VHDL Behavioral Models

by

Sanat R. Rao

Dr. James R. Armstrong, Chairman

Electrical Engineering

(ABSTRACT)

This thesis describes the development of the Hierarchical Behavioral Test Generator (HBTG) for the testing of VHDL behavioral models. HBTG uses the Process Model Graph of the VHDL behavioral model as the base for test generation. Test sets for individual processes of the model are precomputed and stored in the design library. Using this information, HBTG hierarchically constructs a test sequence that tests the functionality of the model. The test sequence generated by HBTG is used for simulation of the model. Various features present in HBTG and the implementation of the algorithm are discussed. The idea of an effective test sequence for a VHDL behavioral model is proposed. A system is presented to evaluate the quality of the test sequence generated by the algorithm. Test sequences and coverage results are given for several models. Some suggestions for future improvements of the tools are made. The HBTG forms part of a complete CAD system for rapid development and testing of VHDL behavioral models.

Acknowledgments

I wish to express my sincere thanks to my advisor Dr. James R. Armstrong for his constant support and encouragement through the conception of this work. His patience, friendly guidance and caring have been invaluable during the course of this thesis. The opportunities provided by him have richly enriched my knowledge. I would like to thank Dr. Walling R. Cyre and Dr. F. Gail Gray for serving as members of my committee. Many thanks are also due to Dr. Sallie M. Henry for her enthusiasm and helpful suggestions.

It has been a pleasure working with all the members of the two research groups. Special thanks are due to S. Shankaranarayanan for his wholehearted assistance in times of difficulty. The exemplary work as well as refreshing company of Philip Wright, Chang Cho, Gunjeet Baweja and Meenakshi Manek, to name but a few others, made research a very enjoyable experience.

I gratefully acknowledge the funding support provided in part by the National Science Foundation.

I would like to dedicate this work to my parents who have bravely stood behind me through all my difficult times. Their unreserved love and support are the most important reasons for all my achievements. My friends have provided me with my most enjoyable moments. I specially wish to thank Chaitanya Rajguru who showed me that God helps those who help others. It is to all my friends that I convey my deepest gratitude.

Table of Contents

List of Illustrations

List of Tables

Chapter 1. Introduction

1.1 Motivation

The complexity of integrated circuits has been increasing at a rapid rate over the past few years. Modern VLSI technology provides the ability to produce chips composed of over a million transistors [1]. For example, the Intel 486DX consists of 1.2 million transistors. In addition, chips consisting of up to a hundred million transistors are expected over the next few years. The design of such complex chips requires extensive use of computer tools. Computer simulation is one of the most powerful tools for design verification and test generation. A crucial factor in simulation is the capability to accurately describe the behavior of the actual circuit. Computer languages such C and PASCAL are inadequate for this purpose, since they do not have the ability to model concurrency in the execution of the logic blocks. Furthermore, these languages cannot represent accurate timing relations between logic blocks. This has led to the development of special languages called Hardware Description Languages (HDLs).

Simulation and design verification using the VHSIC hardware description language (VHDL) is becoming extremely popular in industry. The main reason for this is that

VHDL has a rich set of constructs for the accurate modeling of digital systems, and is an IEEE standardized hardware description language [2]. Using VHDL, designs can be described in top-down or bottom-up fashion through varying levels of abstraction. Once the VHDL behavioral model of a circuit is developed, test patterns need to be generated and applied for simulation of the model. By observing the simulation output, the functionality of the model can be verified. The correct functioning of the VHDL behavioral model is a direct reflection on the correctness of the original design.

Traditionally, test development for behavioral models has relied on a design engineer or model developer to manually construct tests. This is a very time-consuming and labor-intensive task. Morever, such an approach yields test sets that satisfy no formal definition of completeness. This is because the tests written reflect the writer's view about what the model should do. As a result, basic functions are frequently left untested. Several research efforts [15-18] have concentrated on test generation for VHDL models.

A preliminary software tool, the Hierarchical Behavioral Test Generator (HBTG), was developed [21] at Virginia Tech for the testing of VHDL behavioral models. HBTG accepts the process model graph of the VHDL behavioral model as input. Precomputed tests for each process of the PMG are stored in the design library. Using this information, the HBTG algorithm generates a test sequence for the entire entity. This algorithm is discussed in chapter-3 in more detail.

The above test generation algorithm was developed under many assumptions and had severe limitations that completely precluded its effective usage for testing of VHDL models. Some of the limitations are discussed below:

1. The algorithm did not permit the existence of *multiple-fanout condition,* i.e., the condition in which the output signal of a process is the input to more than one process. The algorithm assumed that all circuits are fanout-free. This limitation prevented any real system from being tested, since most real systems invariably contain multiple-fanout structure.

2. The algorithm assumed that all processes had a single output. i.e., it did not permit the existence of a *multiple-output condition.* Hence, processes having more than one output port were not allowed. This further restricted the kinds of models that could be tested, since several common primitives like decoders, adders, etc., have more than one output port.

3. The algorithm was not designed to incorporate sequential primitives in the test generation process. Only combinational primitive processes were allowed.

4. The ability to permit *feedback* within the VHDL models was not present in the algorithm, i.e., a signal could not be fed back from an output port to any input port in the model.

5. No technique was present to evaluate the quality of the tests generated by the HBTG algorithm. As a result, it was not possible to determine the effectiveness of the generated test sequence. Hence, a system to evaluate the quality of the test pattern was required.

6. At the end of the test generation process, there was no information provided to the user about the different functions that had been tested. Hence, the user was not aware of the capability of the test-sequence constructed by the algorithm. In order to prevent any misunderstanding, it was necessary to provide an indication about the test-sequence.

7. The program was written at a very basic stage. Crucial test generation procedures were not properly developed. The structure of the code was inefficient. Important subroutines for selecting tests and assigning values to signals during the activation, propagation and justification processes were very elementary or did not exist at all.

7. There were several bugs and hitches in the software that caused frequent program crashes and faults. Problems such as improper memory allocation for the data structures and incorrect handling of pointers caused the program to crash more often than not.

The above major limitations prevented the use of the HBTG algorithm in testing any real-time systems. In fact, only four - five models were ever completely tested. Very few primitive modules were used, and the ones used were predominantly gate-level primitives rather than high-level processes.

1.2 Contributions

The Hierarchical Behavioral Test Generator has been completely re-written in Sun-C and is now running on a SUN SPARCstation II. Only certain basic test generation procedures and the core data structure have been carried over from the previous algorithm. The current HBTG algorithm can run on any system on which the Modeler's Assistant is installed. A new program has been written to interface to the Modeler's Assistant in order to receive the PMG information. The current HBTG algorithm is versatile in the sense that models of reasonable size can be tested. Basic procedures for activation, justification and propagation have been changed and the algorithm enhanced to incorporate several new features.

The sensitive path construction and the test generation process have been optimized by taking multiple-fanout structure and processes with multiple-output structure into consideration. The presence of these features is essential in any system where models of any reasonable complexity are to be tested.

The current version of HBTG allows sequential primitives to be present in a model. Test generation is performed by first driving the primitives into an initial state by a combination of user-specified information and automatic test generation techniques. The characteristics of the precomputed tests stored in the design library have been examined, since they play an important role in this process.

An important feature of HBTG is its ability to generate tests for models with feedback. The presence of a feedback signal causes unknown initial states in a model and

must be initialized before the test generation process can begin. Another useful feature of HBTG is its ability to generate tests for models with resolved ports. Currently, ports of type TSL and TSL_VECTOR are supported, and provisions have been made to incorporate other types in the future.

A complete system has been developed to evaluate the quality of tests generated by the HBTG algorithm. The proposed system utilizes the coverage environment of the simulator to determine the quality of the generated test pattern. The presence of this feature is very important in any software testing system. Several results have been presented to illustrate this feature.

The Hierarchical Behavioral Test Generator consists of approximately 4000 lines of C code. Various macros have been defined in order to facilitate the manipulation of data-structures used in the program. Various bugs present in the previous algorithm have been removed. The C files have been appropriately documented and a programmer's manual has been developed, as is essential in a project of this magnitude.

1.3 Contents

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

Chapter 3, "Previous Development" explains the development of the Modeler's Assistant and the previous HBTG algorithm.

Chapter 4, "New Developments", briefly discusses the various new features incorporated in the current HBTG algorithm.

Chapter 5, "The Enhanced Hierarchical Test Generation Algorithm", describes the current hierarchical test generation algorithm with the new features. The extraction of information from the Modeler's Assistant, construction of sensitive paths, storage of precomputed tests and the entire test generation process are discussed in detail.

Chapter 6, "Evaluation of Test Quality", presents a system to evaluate the effectiveness of the tests generated by the HBTG algorithm. Several coverage results are provided.

Chapter 7, "Results", discusses the test sequence generated by HBTG for various models and the coverage results obtained from the test vectors.

Chapter 8, "Proposed Future Development", discusses possible future improvements to the hierarchical behavioral test generator.

Chapter 9, "Conclusion".

Appendix, "Programmer's manual for HBTG and its Environment" provides an overview of the various data structures and format of files used by HBTG along with an explanation of the compilation procedures.

Chapter 2. Background and Literature Review

2.1 Modeling with VHDL

The VHSIC hardware description language (VHDL) was standardized by the IEEE in 1987 [2]. VHDL has an effective set of constructs for chip-level modeling [3]. It can model concurrent execution of logic modules and can represent timing relations between modules accurately.

2.1.1 Structural and Behavioral Models

There are two kinds of VHDL models :

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

Behavioral model: in which the model is described in terms of the input/output response of the circuit.

A 2-TO-1 Multiplexer with enable is chosen to illustrate the VHDL model development for digital circuits. Figure-1 shows the 2-TO-1 multiplexer.

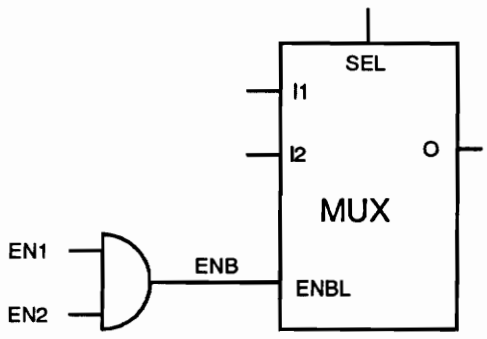


Figure-1: 2-TO-1 Multiplexer with Enable

```
entity TWO_TO_ONE_MUX is
   port(I1, I0, EN1, EN2, SEL: in BIT;
      O : out BIT);
end TWO_TO_ONE_MUX;

architecture STRUCTURAL of TWO_TO_ONE_MUX is
component MUX
   port (I1, I0, SEL, ENBL : in BIT; O: out BIT);
end component;
component AND2
   port (IN1, IN2: in BIT, OP: out BIT);
end component;
signal ENB: BIT;

begin
  COMP_1: AND2
            port map (EN1, EN2, ENB);
  COMP_2: MUX
            port map (I1, I0, SEL, ENB);
end STRUCTURAL;
```

Figure-2: VHDL Structural Model for the 2-TO-1-MUX with Enable

When the enable lines EN1 and EN2 are asserted to logic 1, the MUX gets enabled and depending on the select input SEL, it passes either I0 or I1 to the output O. The VHDL structural model of the 2-TO-1 mux with enable is shown in Figure-2. As we can see, the entity is described as an interconnection of two primitives: AND and MUX.

As opposed to the structural model, the VHDL behavioral model describes the operation of the circuit in terms of its I/O response without resorting to lower-level representations. A VHDL behavioral model for the 2-TO-1 MUX is shown in Figure-3.

```
entity TWO_TO_ONE_MUX is
   port(I1, I0, EN1, EN2, SEL: in BIT;
      O : out BIT);
end TWO_TO_ONE_MUX;

architecture BEHAVIOR of TWO_TO_ONE_MUX is
begin
  process(EN1, EN2, SEL)
  begin
          if ((EN1 and EN2) = '1') then
             case SEL is
                   when '0' => O <= I0;
                   when '1' => O <= I1;
             end case;
          end if;
  end process;
end BEHAVIOR;
```

Figure-3: VHDL Behavioral Model for the 2-TO-1 MUX with Enable

One advantage of the behavioral model is that it can be developed at different levels of abstraction. The model in Figure-3 is an algorithmic model consisting of one process. A model at a different level of abstraction can be developed by partitioning the

circuit either structurally or functionally. A behavioral model for the 2-TO-1 MUX at a different level of abstraction is shown in Figure-4. Each of the submodules in this model is a process. As we go down the abstraction hierarchy, the number of processes in the VHDL behavioral model increases and the model provides greater circuit detail.

```
entity TWO_TO_ONE_MUX is
   port(I1, I0, EN1, EN2, SEL: in BIT;
      O : out BIT);
end TWO_TO_ONE_MUX;

architecture BEHAVIORAL of TWO_TO_ONE_MUX is
signal ENB: BIT;
begin
  AND: process (EN1, EN2)
  begin
          ENB <= EN1 and EN2;
  end process AND;

  MUX: process (ENB, SEL)
  begin
          if (ENB = '1') then
            case SEL is
                  when '0' => O <= I0;
                  when '1' => O <= I1;
            end case;
          end if;
  end process MUX;

end BEHAVIORAL;
```

Figure-4: VHDL Behavioral Model for the 2-TO-1 MUX at a different level of abstraction

2.1.2 The Graphical Representation of the VHDL Behavioral Model

The graphical representation of a system is used to illustrate the relationships between the various elements of the system. The ***Process Model Graph*** is the graphical representation of the VHDL behavioral model. Figure-5 shows an example process model graph. As can be seen, the process model graph (PMG) is a directed graph with the processes in the model as the nodes of the graph and the edges between the nodes representing the signals between processes. The arrows indicate the direction of flow of signals. Thus, the PMG represents a partitioning of the VHDL model in the behavioral domain. Figure-6 shows the PMG of the 2-TO-1 MUX.

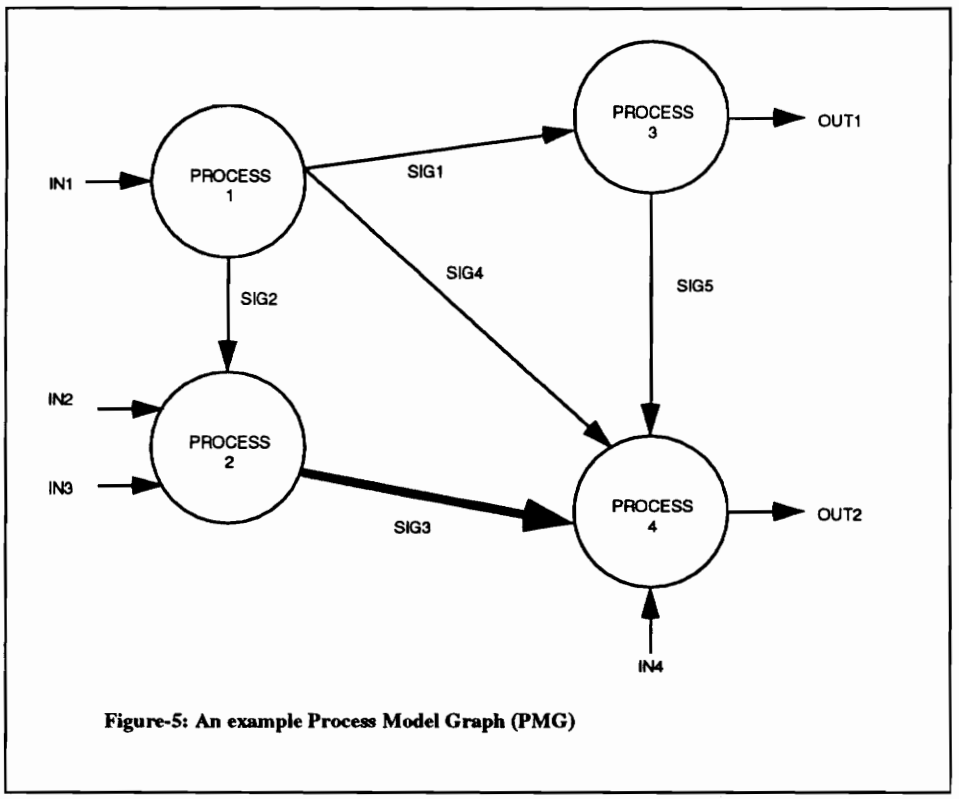


Figure-5: An example Process Model Graph (PMG)

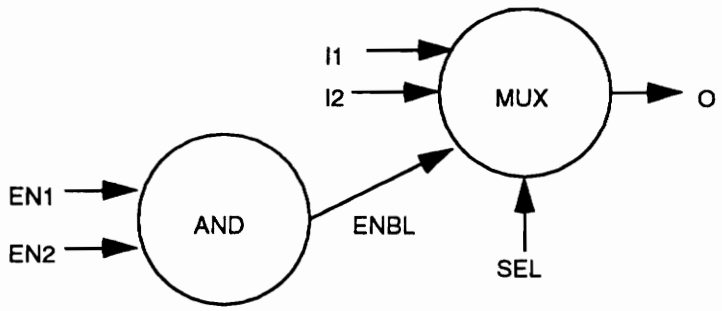


Figure-6: Process Model Graph of the 2-TO-1 MUX

2.2 Test Generation for Digital Systems

Testing is the process of exercising a digital system and analyzing the resulting output to determine whether the system is behaving correctly [4]. ***Test Generation*** is the process of determining the set of test patterns required to test a digital system. Test generation can be classified into two basic categories:

(i) *Fault-Oriented Test Generation* in which tests are generated to detect specific faults in the model.

(ii) *Function-Oriented Test Generation* in which tests are generated which, when applied to the model, show whether the model performs a specific function or not.

2.2.1 Gate Level Test Generation

Traditional test generation algorithms generate tests using gate-level structural descriptions where the circuit is treated as an interconnection of primitive gate components like the AND gate, OR gate, XOR gate, etc. The signals in the circuit are treated as single-bit buses. A simple fault model like the *stuck-at fault* (s-a-v, $v \in \{0,1\}$) is used to assume logic lines in the gate-level circuits are stuck at logic 1 or logic 0, and test patterns are generated to detect the fault. The test-vector generated to detect a fault must perform two basic operations:

(i) *Fault Sensitization*: The test-vector must generate a value at the fault site that is opposite to the faulty value at the site.

(ii) *Fault Propagation*: The test-vector must be able to propagate the fault to a primary output by assigning appropriate values to internal signals. The values assigned during creation of this propagation path must then be justified back towards the primary inputs.

Several algorithms have been developed for the gate-level test generation process. The D-Algorithm proposed by Roth [5] in 1966 was the one of the first algorithms for generating tests for gate-level combinational circuits. The D-Algorithm uses primitive d-cubes of faults to sensitize the fault at the fault site. The symbol D represents a good value of logic 1 and a faulty value of logic 0, and the symbol $\overline{D}$ represents a good value of logic 0 and a faulty value of logic 1. During the test generation (TG) process, the fault

is sensitized by generating a D or a $\overline{D}$ at the fault site depending on the stuck-at fault. The D or $\overline{D}$ is then propagated to the primary output by assigning appropriate values and justifying these values to primary inputs.

PODEM (Path-Oriented Decision Making) was proposed by Goel [6] in 1981. The algorithm uses the symbols D and $\overline{D}$ for fault sensitization. In the test generation process, values are first 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 values of the internal signals due to the primary inputs are determined. Knowledge of the implied logic values at internal signals can avoid conflict in logic value assignment of a signal when continuing propagation of D or $\overline{D}$ towards a primary output. Hence, the number of backtracking steps needed is greatly reduced as compared to the D-algorithm.

FAN (Fanout-Oriented Test Generation) was proposed by Fujiwara and Shimono [7] in 1983. FAN also uses the stuck-at fault model and the symbols D and $\overline{D}$ for fault sensitization. The FAN algorithm improves upon PODEM by stopping the backtracking process at internal lines rather than at the primary inputs. Thus, the number of backtracking steps can be reduced and test generation time shortened.

The above gate-level test generation algorithms provide a systematic method to effectively generate tests for MSI and LSI circuits. They are widely used in industry.

2.2.2 High Level Test Generation

The gate-level schemes described in the previous section are suitable for relatively small circuits. However, the increasing complexity of modern VLSI digital circuits has made test generation at the gate-level extremely difficult. The test generation problem at the gate-level has been shown to be NP complete [8]. Hence, gate-level test generation becomes more and more time-consuming and expensive as the complexity of the circuit increases. Morever, 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 gained prominence in the domain of VLSI testing. These techniques view the circuit with lesser structural detail as compared to gate-level methods.

The major goal of these high-level techniques is to reduce test generation complexity and computational time by the use of high-level primitives in the test generation process. i.e., the model of the circuit under test consists of high-level modules, rather than a complex interconnection of gate-level primitives. Also, the number of signals in the circuit is much smaller than that in the corresponding gate-level representation. Since the circuit has fewer primitive components, fewer module evaluations have to be performed during test generation [9]. A circuit consisting of over two hundred primitive gates can be represented as a circuit of only six components and a small number of buses by using a high-level representation [12]. This reduced complexity provides a great advantage over the traditional gate-level techniques. In addition, these high-level techniques can exploit any modularity involved in the design process itself, particularly in

cases where CAD tools are used for the design of the circuit. This section illustrates various high-level approaches to test generation.

Kunda, Narain, Abraham and Rathi [10] propose an approach to speed up the test generation process by making use of high-level primitives. The circuit to be tested is modeled as a data-flow graph with the high-level primitives as the nodes of the graph. Primitives such as AND, NAND, XOR, BUFF, JOIN, NEGATE, MULTIPLY, EQUAL, SPLIT, etc., are supported. Dependency-directed backtracking mechanism is used to reduce the number of backtrackings needed. According to experimental results for five circuits, this approach provides a significant speed-up and reduction in memory requirement for circuits composed of high-level primitives.

Somenzi et al [11] present a set of algorithms for high-level test generation. These techniques are similar to the D-Algorithm. The system to be tested consists of combinational macros interconnected by single-bit lines. Each macro performs a well-defined function based on functional partitioning. Tests are first generated locally as the involved macro is considered to be isolated. The algorithm regards the macros as the primitive elements instead of the boolean gates. The tests are then expressed in terms of primary inputs and outputs using a topological strategy and algebraic method for propagation of signals through macros.

Murray and Hayes [12] propose a hierarchical approach to test generation based on high-level primitives. Their view is similar to that of [10] except that no specific fault model is adopted. Test information for the modules is stored in the design library along with the design information for the modules. The tests are stimulus/response pairs for the

modules. There are two kinds of tests: 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. The algorithm is similar in nature to the D-Algorithm. Experimental results measuring the number of module evaluations indicate that this approach provides a great improvement over the traditional gate-level test generation.

Kruger [13] proposes a tool for hierarchical test generation. This test generator is part of a computer hardware design system (MIMOLA). For primitive and behavioral models, predefined *test-packages* similar to those in [12] are stored in the library and provide stimuli and response. If no input stimuli are present, the test generator applies default patterns and computes the responses. Test packages are generated and stored for all modules at one level, and are then used in the next level of hierarchy.

Sarfert, Markgraf, Schulz and Trischler [14] present the extension of an existing ATG system for hierarchical test pattern generation for combinational circuits. High-level primitives supported include the decoder, encoder, adder, etc. In order to perform ATG for faults inside a high-level primitive (HLP), the primitive is expanded dynamically to its gate-level representation. By expanding only one high-level primitive at a time, this method takes advantage of the use of HLPs in the ATG process. Experimental results indicate that the algorithm performs significantly better as compared to a gate-level ATG in terms of CPU time and memory requirements.

2.2.3 Test Generation from Hardware Description Languages

Test generation from hardware description languages is a special case of the high-level test generation explained in the previous section. Most high-level test generation schemes use their own specific circuit representations to describe the circuit during the test generation process. Describing the model in a standardized hardware description language saves time as well as provides compatibility between different approaches. A hardware description language is an excellent means of accurately modeling and simulating digital circuits. HDL models are excellent sources of information for test generation at the functional level. In this section, several approaches to test generation from hardware description languages are reviewed.

Levendel and Menon [15] use a generalization of the D-algorithm to generate tests for circuits containing modules described in a hardware description language. Both procedural (sequential) and non-procedural (concurrent) HDL representations are considered during the test generation process. In case of procedural descriptions, path functions representing successive HDL statements are used for D-propagation. For non-procedural descriptions, the path function is not required. The derivation of the D-propagation cubes and the path functions for large HDL functions is very complicated and may reduce test generation efficiency.

The FunTestIC test generation method (FUNctional TEST pattern generation for Integrated Circuits) proposed by Hummer, Veit and Topfer [16] derives functional tests for modules and systems described in VHDL In this approach, the circuit model is represented in structures of a Control Flow Graph (CFG), Data Flow Graph (DFG), and

Sequence Graph (SG). The CFG is similar to a flow chart representation of the circuit model. The DFG represents the possible data flows between the variables of the models. The SG provides information about the sequence in which the data has to be passed in the CFG. Input/Output paths are derived from the description and activated, and the circuit is exercised with characteristic data.

A Behavioral Test Generator [17] has been developed at Virginia Tech using artificial intelligence techniques. The BTG operates directly on the VHDL description statements. High-level fault models such as the Stuck-Then, Stuck-Else, etc., are first developed to represent faults in the VHDL behavioral model. The algorithm, which is a modified version of the D-algorithm, then generates tests to detect the faults.

Cho and Armstrong [18] 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.

Chapter 3: Previous Development

CAD tools for rapid development and testing of VHDL behavioral models have been the focus of research at Virginia Tech over the last few years. One major product that has evolved out of this research is the Modeler's Assistant which is a tool for VHDL behavioral modeling. The other major tool that has been developed is the Hierarchical Behavioral Test Generator, which is the focus of this thesis.

3.1 The Modeler's Assistant

The Modeler's Assistant [19, 20] is a graphical CAD tool for the development of VHDL behavioral models. This tool accepts the Process Model Graph of the VHDL behavioral model, created interactively by the user, as input. The user also enters the functionality of each process in the process model graph. The Modeler's Assistant then uses this information to construct the entire VHDL behavioral model. It thus provides a mixed textual/graphical design capture to reduce the effort required in the development of a VHDL behavioral model.

Using the Modeler's Assistant, the user inputs the PMG of the VHDL model. Each process in the PMG is first defined and saved. The PMG is then constructed by loading (adding) the processes and then drawing the signals between them. Frequently used processes are stored as parametrized primitives in the library and used in the design process. This significantly reduces the time required to develop behavioral models [22].

Figure-7 shows an example Process Model Graph created on the Modeler's Assistant. In the figure, the large circles represent the processes. The smaller circles on the periphery of the process represent process ports[1]. Sensitive ports are depicted by shading the area inside the port. Small squares inside the processes indicate constants and variables. One or more processes together form the PMG. Figure-8 shows the VHDL "shell" produced by the Modeler's Assistant for the PMG in Figure-7. The VHDL model thus developed can be analyzed by any analyzer installed in the system.

Figure-9 shows the system block diagram of the Modeler's Assistant. The important components of the system are two databases, one containing the information to generate the graphics and the other containing information to produce the VHDL code. The PMG database contains the details of the geometry of the process model graph in terms of the coordinates of the various graphical constructs like modules, ports, signals, variables, etc. Information about modes of a port (in, out, inout), sensitivity, types of signals, etc., is also included. The process functionality database contains the text of the functionality of all the processes in the PMG. The HBTG algorithm described in this thesis extracts information from the PMG database and uses it in the test generation process.

[1] The term 'port' refers to a process input of output.

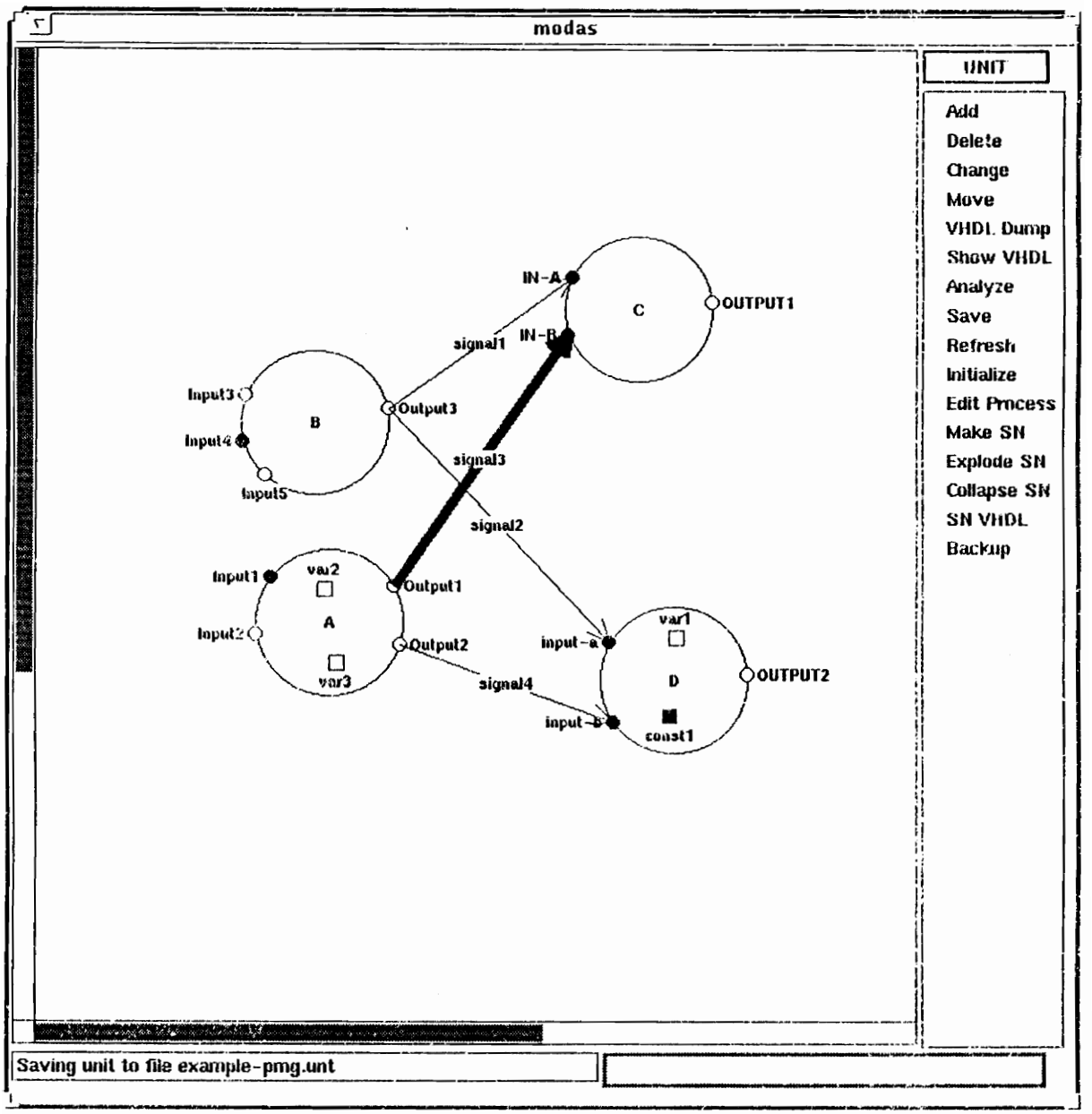


Figure-7: Example PMG created using the Modeler's Assistant

```
use WORK.VHDLCAD.all, WORK.USER_TYPES.all;
-- ********************************************************
entity example-pmg is
  port (OUTPUT2: out BIT;
    OUTPUT1: out BIT_VECTOR(0 to 7);
    Input5: in BIT;
    Input4: in BIT;
    Input3: in BIT;
    Input2: in BIT_VECTOR(0 to 7);
    Input1: in BIT
    );
end example-pmg;
-- ********************************************************

architecture BEHAVIORAL of example-pmg is

  signal signal2: BIT;
  signal signal4: BIT;
  signal signal3: BIT_VECTOR(0 to 7);
  signal signal1: BIT;
begin


  -- ---------------------------------------------------
  -- Process Name: D
  -- ---------------------------------------------------
  D_2: process (signal2,signal4)
     variable var1: BIT;
     constant const1: INTEGER;
  begin

     User-Specified VHDL

  end process D_2;
```

```
  -- ---------------------------------------------------
  -- Process Name: C
  -- ---------------------------------------------------
  C_11: process (signal3,signal1)
  begin

     User-Specified VHDL

  end process C_11;


  -- ---------------------------------------------------
  -- Process Name: B
  -- ---------------------------------------------------
  B_16: process (Input4)
  begin

     User-Specified VHDL

  end process B_16;


  -- ---------------------------------------------------
  -- Process Name: A
  -- ---------------------------------------------------
  A_22: process (Input1)
    variable var2: BIT;
    variable var3: BIT;
  begin

     User-Specified VHDL

  end process A_22;

end BEHAVIORAL;
```

Figure-8: VHDL shell produced by the Modeler's Assistant for the example PMG

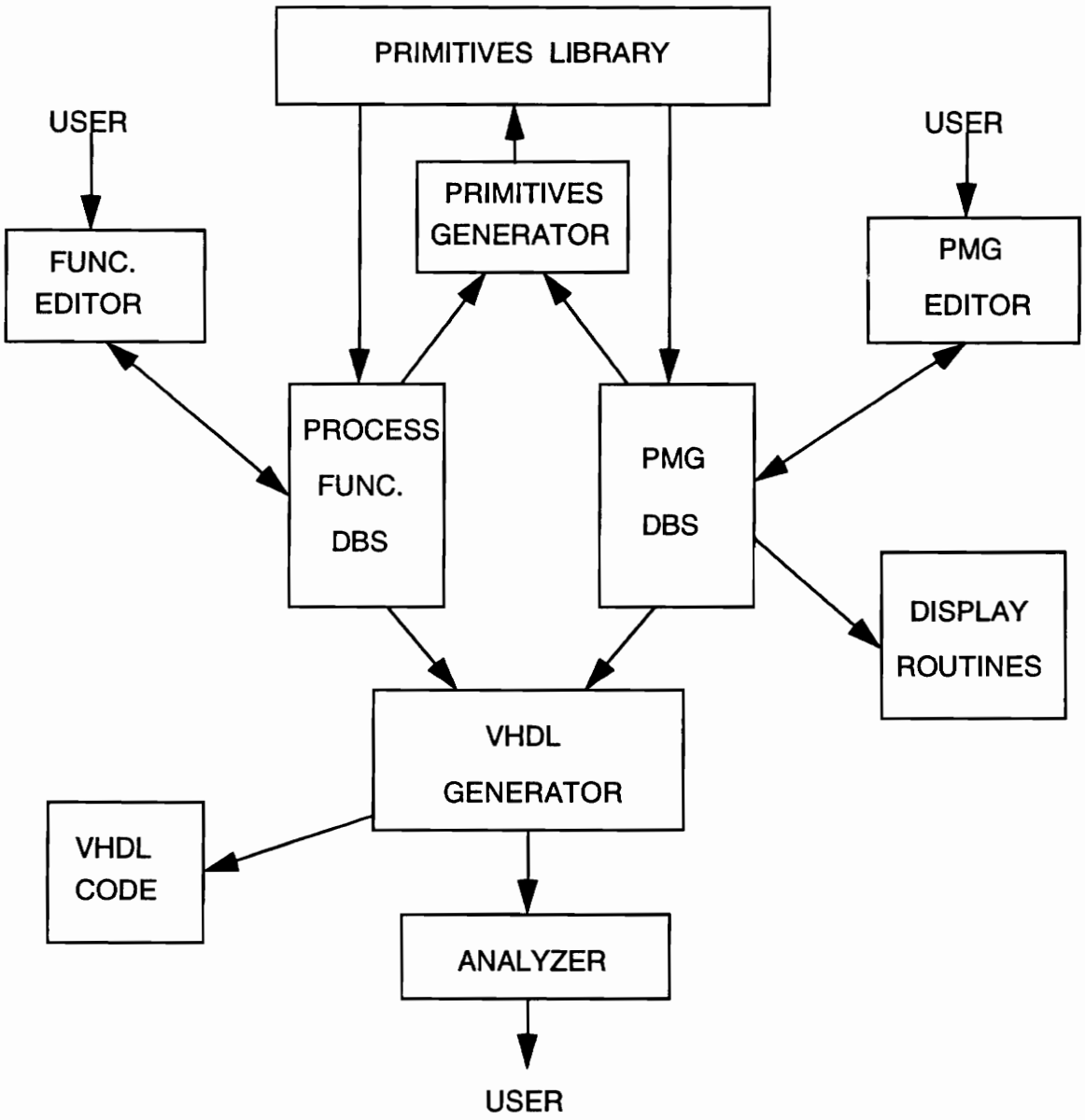


Figure-9: System Block Diagram of the Modeler's Assistant

3.2 The Hierarchical Behavioral Test Generator

The Hierarchical Behavioral Test Generator (HBTG) is a program developed for the automatic test generation for VHDL behavioral models. The process model graph of the VHDL model to be tested is created using the Modeler's Assistant described in the previous section. For each process in the process model graph, tests are precomputed and stored in the design library. Using these tests, the HBTG algorithm constructs a test sequence for the entire VHDL model automatically. The tests are used for simulation of the model.

3.2.1 Test Generation Approach

A hierarchical approach to test generation is used. We know that storage of primitive processes in the design library accelerates the model development process [23]. This is also true to a large extent in the test generation process. Test information for individual modules of the circuit is precomputed and stored in the design library, similar to the approach in [12]. The test sequence for the whole entity is then constructed from these primitive tests. The precomputed tests need to provide sufficient information about the functionality of the processes in the PMG.

In many traditional test generation techniques, specific fault models are used to represent faults in the circuit. For example, the s-a-v ($v \in \{0,1\}$) fault model is commonly

used to represent logical faults in gate level test generation as explained in chapter 2. In a higher level approach like the BTG [17], fault models like Stuck-Then, Stuck-Else, Assignment Control, etc., are used.

The HBTG does not adopt any specific fault models. Instead of generating a test sequence to detect a fault, the algorithm constructs a test sequence that exercises the model. i.e., when the test pattern is applied to the model for simulation, it will test various functions of the model.

3.2.2 Construction of Sensitive Paths

The HBTG follows the traditional test sequence of activation - propagation - justification. The term *activation* refers to the generation of an event on a port of the Process Model Graph. This creates a rise or a fall on a single-bit bus or assigns different bit-vectors to a multi-bit bus. The signal resulting from activation of a port has to be *propagated* to a primary output (PO) of the PMG to be observed. The internal values assigned in this process need to be *justified* towards the primary inputs (PIs) of the PMG.

In order to perform signal propagation and justification, sensitive paths are constructed through the process model graph.

Definition: A ***Sensitive Path*** is a directed path that starts at a sensitive primary input port (PI) and ends at a primary output port (PO) with the intermediate ports along the path consisting of as many sensitive ports as possible.

The activation of a port is done along a sensitive path. When the activation of a sensitive path is finished, the result of activation has been propagated to a primary output. The justification process then begins to justify the pre-assigned signal values assigned during propagation towards the primary inputs.

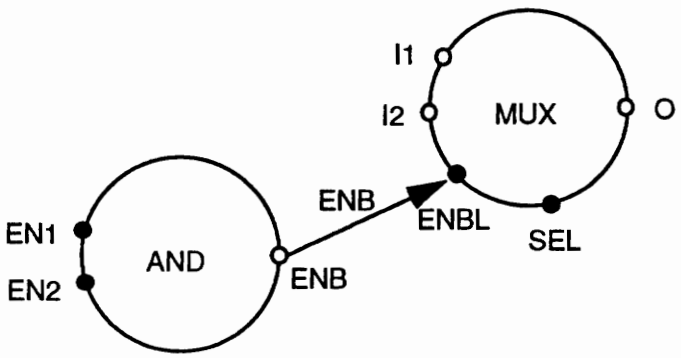


Figure-10: PMG of the 2-TO-1 MUX

Figure-10 shows the PMG of the 2-TO-1 MUX with sensitive and non-sensitive ports appropriately represented. The Sensitive paths constructed by the algorithm through the PMG are shown in Figure-11. As can be seen, three sensitive paths have been constructed through the PMG.

```
No. of signals = 7
path 0:
sp[0][0]=7(SEL)
sp[0][1]=5(O)

path 1:
sp[1][0]=15(EN2)
sp[1][1]=13(ENB)
sp[1][2]=8(ENBL)
sp[1][3]=5(O)

path 2:
sp[2][0]=16(EN1)
sp[2][1]=13(ENB)
sp[2][2]=8(ENBL)
sp[2][3]=5(O)

No. of Spath = 3
```

Figure-11: Sensitive Path Construction for the 2-TO-1 MUX

3.2.3 Test generation Process

The HBTG was written in ANSI-C for an APOLLO DN 3500 workstation. In order to use HBTG, the process model graph of the VHDL model is first created using the Modeler's Assistant. Sensitive paths are constructed through the PMG as discussed in the previous section. Precomputed tests for each process of the PMG are assumed to be present in the design library. Once the sensitive paths have been constructed, the test generation process begins. During the test generation process, the sensitive paths are selected and activated one by one.

The basic test generation process is as follows:

Step 1: Select a sensitive path not yet activated.

Step 2: Activate the first port along the path, i.e., generate an event (change in signal value) on the first port on the sensitive path.

Step 3: Propagate the resulting signal value forwards till the primary output on the path is reached.

Step 4: Justify the assigned signal values backwards until the primary inputs are reached.

Step 5: Perform implication for modules with known inputs and unknown outputs.

Step 6: If all the sensitive paths have not yet been activated, go to step 1.

When the activation of all the sensitive paths is finished, the test generation process ends and the test-sequence for the model is obtained.

Table-1 on page-31 shows the test sequence generated by HBTG for the 2-TO-1 MUX in Figure-10.

Table-1: Test Sequence for the 2-TO-1 MUX

frame	O	SEL	I2	I1	ENB	EN2	EN1
0	X	X	X	X	X	X	X
2	X	X	X	X	R	1	R
1	F	F	1	0	1	1	1
3	R	0	0	1	R	R	1
4	R	0	0	1	R	1	R

In the above table, the signals of the PMG are on the horizontal axis and the vertical axis represents the time-frames for test generation. The signal values are in symbolic form. The meanings of the symbols are as follows:

'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' : Unknown value or don't care.

'Z' : High Impedance.

'D_i ': Any chosen value in either a single-bit bus or a multi-bit bus. The values can be D1, D2, D3, D4, etc.

The test-sequence for the 2-TO-1 MUX in table-1 is converted into a test-bench and is used for simulation of the model. The test sequence is applied in lexical order.

Chapter 4. New Developments

As discussed in chapter 1, the previous test generation algorithm developed was a very elementary one. Models with more than three processes could not be tested, i.e., the algorithm could do only one step of propagation and justification. The previous test generation algorithm has been completely rewritten and several new ideas and techniques have been incorporated into the current HBTG algorithm. This chapter is devoted to the discussion of these new features. The enhanced algorithm is discussed in the next chapter.

4.1 Test Generation for Models with Multiple Fanout

Multiple fanout is the condition in which the output signal from a process is input to more than one process. i.e. given a process model graph G as a triple {P(G), S(G), Ψ_G} where Ψ_G is an incidence function that associates every signal in the set {S(G)} with two processes in the set {P(G)}, there is at least one signal $S_i \in$ {S(G)} for which

$$\Psi_G(S_i) = (P_i, P_j) \quad \text{and}$$
$$\Psi_G(S_i) = (P_i, P_k)$$

where i, j and k are assumed distinct without loss of generality and $P_i, P_j, P_k \in$ {P(G)}.

The problem with multiple-fanout structure is that there is more than one path for an event to be propagated to the output. If the test generation process does not consider fanout, propagation always takes place along the same path, irrespective of the existence of multiple paths. Hence, certain processes will not be executed at all. Since our aim is to exercise as many operations of the model as possible, it is necessary to optimize the test generation for multiple fanout structure.

In order to generate tests for models with a fanout structure, the sensitive path construction first needs to be optimized through the fanout structure(s). This is done by a procedure called *Select_Destn_Port*. The pseudo-code for this procedure is shown in Figure-12.

```
procedure Select_Destn_Port
begin
   if more than one signal is leaving port, then
         call  nu_destn_ports;
         if a destn port has not yet been activated, then
                  choose that port as the next port along sensitive path;
         else
                  choose the least activated of all
                           destination ports as next port along sensitive path;
         end if;
   else
         choose the destination port of
                  the signal as next port along sensitive path;
   end if;
end procedure;
```

Figure-12: Procedure *Select_Destn_Port*

When a multiple fanout port is encountered, the above algorithm looks for a destination port that is not on any other path as yet. If such a port does not exist, it

chooses that port which is on the least number of other sensitive paths as the next port on the path. The function *nu_destn_ports* returns the total number of destination ports, and hence the total number of signals leaving a fanout port.

During the test generation process, events are propagated along the sensitive path by means of a procedure *Select_Fanout_Destination* that selects tests for that module which contains the destination port.

Figure-13 shows an example process model graph with a multiple fanout structure.

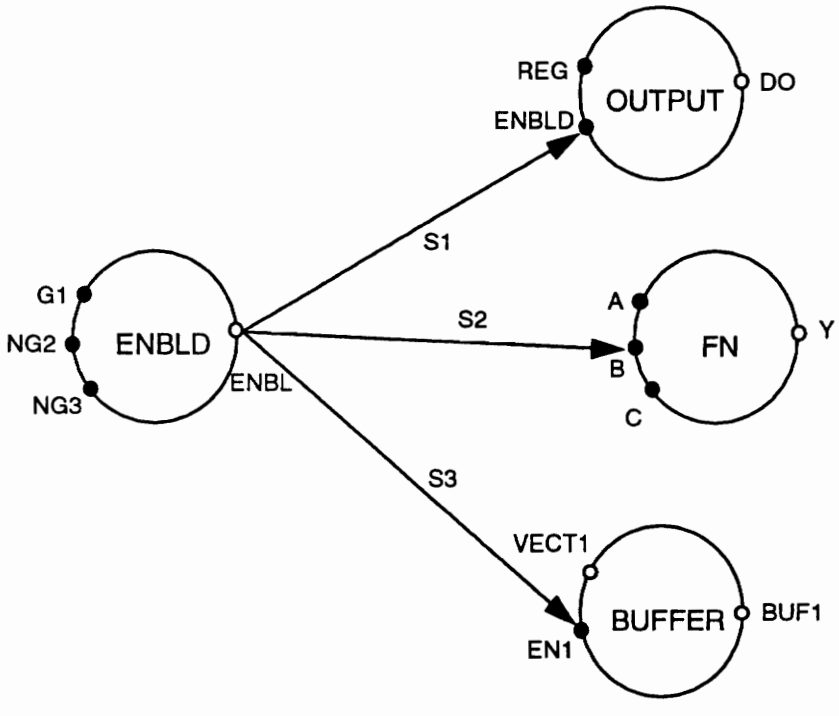


Figure-13: Example PMG showing Multiple Fanout Condition

The sensitive paths constructed through the PMG of Figure-13 is shown in Figure-14 below. As we can see, the paths are constructed through as many fanout branches as possible. The test sequence generated by HBTG for the model is shown in table-2.

```
No. of signals = 11

path 0:
sp[0][0]=14(C)
sp[0][1]=11(Y)

path 1:
sp[1][0]=16(A)
sp[1][1]=11(Y)

path 2:
sp[2][0]=22(REG)
sp[2][1]=19(DO)

path 3:
sp[3][0]=28(NG3)
sp[3][1]=25(ENBL)
sp[3][2]=7(EN1)
sp[3][3]=5(BUF1)

path 4:
sp[4][0]=29(NG2)
sp[4][1]=25(ENBL)
sp[4][2]=15(B)
sp[4][3]=11(Y)

path 5:
sp[5][0]=30(G1)
sp[5][1]=25(ENBL)
sp[5][2]=21(ENBLD)
sp[5][3]=19(DO)

No. of Spath = 6
```

Figure-14: Sensitive Path Construction for the fanout model

Table-2: Test-Sequence for the Fanout Model

frame	BUF1	VECT1	Y	C	A	DO	REG	ENBL	NG3	NG2	G1
0	X	X	X	X	X	X	X	X	X	X	X
2	X	X	X	X	X	X	X	F	0	0	F
1	X	X	R	R	0	X	X	0	0	0	0
4	X	X	1	1	0	X	X	R	1	1	R
3	X	X	R	0	R	X	X	1	1	1	1
5	X	X	1	0	1	D1	D1	1	1	1	1
6	D1	D1	1	0	1	D1	D1	R	F	0	1
7	D1	D1	R	0	1	D1	D1	R	0	F	1
8	D1	D1	1	0	1	D1	D1	R	1	1	R

4.2 Test generation for Models with Multiple-Outputs

If one or more processes in a VHDL model has more than one output port, the model has a multiple-output condition. Several common primitives have more than one output port. For example, an adder has a SUM output and a CARRY output. Similarly, a 2-TO-4 decoder will have four outputs. Hence, it is crucial to consider the existence of more than one output in the test generation process.

The problem with processes having multiple outputs is that there is more than one output port through which an event can be propagated. Ideally, all the output ports of a model should be on some sensitive path, particularly those which are not the primary outputs of a process.

In order to account for multiple outputs during the test generation process, we first need to examine each module during the sensitive path construction process to determine

if there is more than one output port. This is done by a function called *Only_One_Output.* A procedure called *Choose_Output_Port* then selects the port that is on the least number of sensitive paths as the next port on the current sensitive path. The psuedo-code is shown in Figure-15.

```
procedure Choose_Output_Port
begin
    select module;
    assign  temp_port as the first port in the linked list of ports for the module;
    choose next port in the linked list of ports for the module;
    while  all ports of the module have not been scanned, do
        If port is an output port or an inout port
                If port is on fewer sensitive paths than temp_port
                                                assign next port to temp_port;
                end if;
        end if;
    end while;
    assign temp_port as the next port on the sensitive path;
end procedure;
```

Figure-15: Procedure *Choose_Output_Port*

During the test generation process, the *Assign_Value* procedures assign values to all output ports of the process under test. However, only for that port which is on the sensitive path being activated is the event considered and propagated to the primary output. An example PMG with a multiple output condition is shown in Figure-16. The decoder DECD has two output ports DEC1 and DEC0. Figure-17 shows the sensitive paths constructed for the model. As we can see, sensitive paths have been constructed through both the output ports of the decoder.

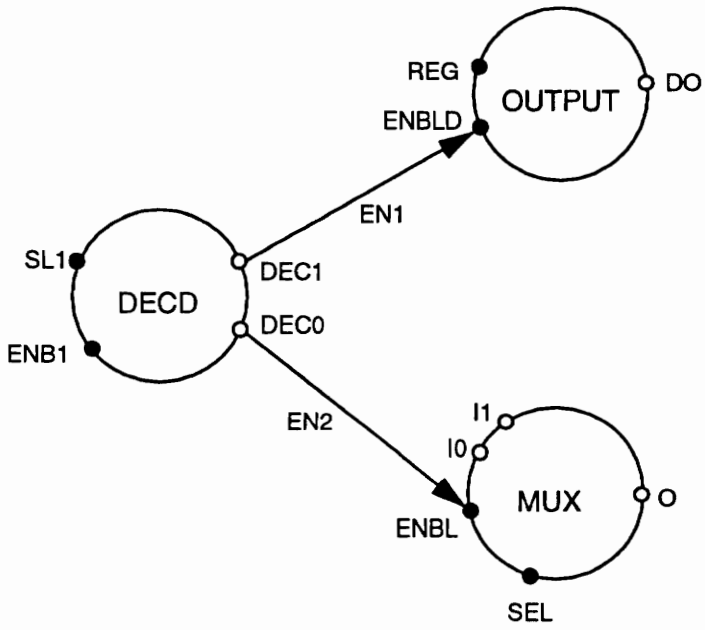


Figure-16: Example PMG with multiple-output condition

No. of signals = 10

path 0:
sp[0][0]=7(SEL)
sp[0][1]=5(O)

path 1:
sp[1][0]=16(REG)
sp[1][1]=13(DO)

path 2:
sp[2][0]=22(ENB1)
sp[2][1]=19(DEC1)
sp[2][2]=15(ENBLD)
sp[2][3]=13(DO)

path 3:
sp[3][0]=23(SL1)
sp[3][1]=21(DEC0)
sp[3][2]=8(ENBL)
sp[3][3]=5(O)

No. of Spath = 4

Figure-17: Sensitive Path Construction for the PMG with Multiple Outputs

The test sequence generated by HBTG for the model is shown in table-3 below.

Table-3: Test Sequence for the model with multiple outputs

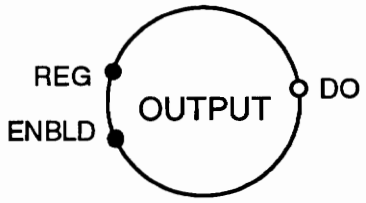
frame	O	SEL	I2	I1	DO	REG	DEC1	DEC0	ENB1	SL1
0	X	X	X	X	X	X	X	X	X	X
2	X	X	X	X	X	X	F	R	1	F
1	F	F	1	0	X	X	0	1	1	0
4	0	0	1	0	X	X	R	F	1	R
3	0	0	1	0	D1	D1	1	0	1	1
5	0	0	1	0	D1	D1	R	Z	R	1
6	R	0	0	1	D1	D1	F	R	1	F

4.3 Improved Activation Scheme

During the test generation process, the HBTG algorithm activates the sensitive paths by selecting tests from the predefined primitive tests. During the activation of a sensitive path, a test is selected that generates an event on the first port on the path. As a result, the process is executed and the resulting signal is propagated to the output.

The previous algorithm had a serious limitation in the sense that it could not activate ports of type BIT_VECTOR, MVL_VECTOR, TSL, etc. i.e., the algorithm could activate only ports of type BIT. The current HBTG algorithm uses a procedure *Select_tst_set* which selects appropriate values to activate sensitive primary input ports. This procedure makes use of functions *Is_Event_Tst* and functions *Is_Event_Tst_Act* to select tests which activate ports of type BIT and TSL by a 'R' of a 'F' on that port, and ports of type BIT_VECTOR, MVL_VECTOR by assigning a 'D1', 'D2', 'D3' or 'D4' on

that port. Morever, in case of a port of type BIT_VECTOR, the selection process chooses a value different from the currently assigned value of that port.



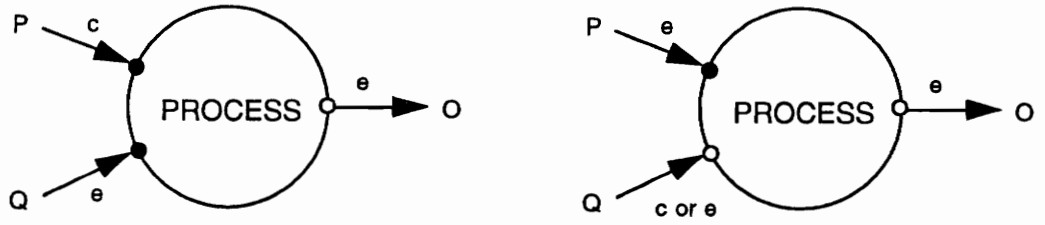
For example, if the sensitive port REG in the process OUTPUT shown above has already been assigned a value 'D1' in a previous time-frame, the HBTG algorithm will choose a value different from 'D1' (i.e., D2, D3, etc.) for activation of the port.

Furthermore, the HBTG algorithm selects a test which has an event both on the sensitive input port as well as the output port of the process. This ensures that the next process on the path will also be executed, and is hence a better test for the model. A constant value at the output is chosen by default, only when a test with an event on that port is not found.

4.4 New Propagation Scheme

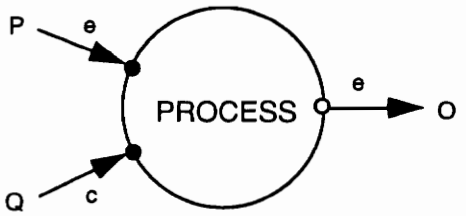
In the test generation process, the activation of a sensitive path leads to the generation of a signal value which needs to be propagated to the primary output on the path. The previous algorithm could propagate only events through sensitive ports. The

new propagation scheme allows the propagation of constant signal values and propagation of signal values through non-sensitive ports.



Case-A: Propagation of an event on sensitive port Q

Case-B: Propagation of an event on non-sensitive port Q



Case-C: Propagation of an constant on sensitive port Q

Figure-18: New Propagation Scheme

The new scheme is implemented in a procedure called *Propagate()*. In the new scheme, if the signal to be propagated is an event and the port is a sensitive port, then a constant value is assigned to the other ports of the module. This is shown in case-A of Figure-18.

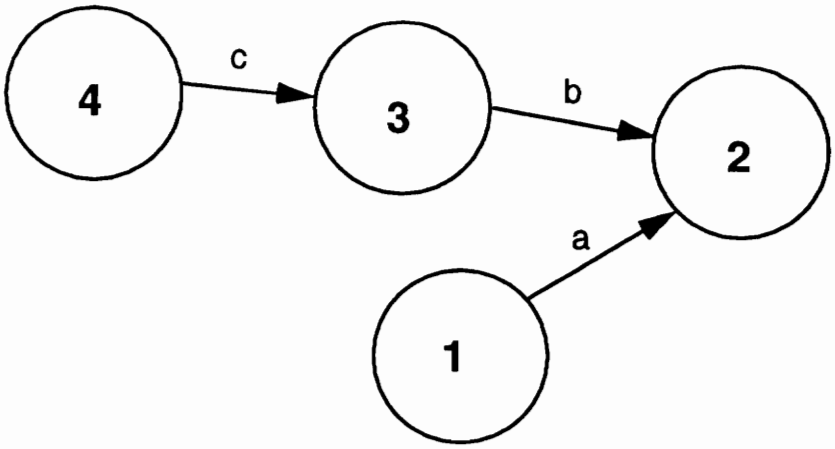
If the signal to be propagated is a value on a non-sensitive port, then it is necessary to consider a sensitive port of the module and select a test that has the required signal value on the port to be propagated and an event on the selected sensitive port. This allows the required process to be executed and the value can be propagated to the output port. This is illustrated in case-B of the Figure-18.

If a constant signal is to be propagated on a sensitive port as shown in case-C, a test is selected that assigns the constant to the sensitive port and assigns an event to some other sensitive port of the model.

4.5 New Justification Scheme

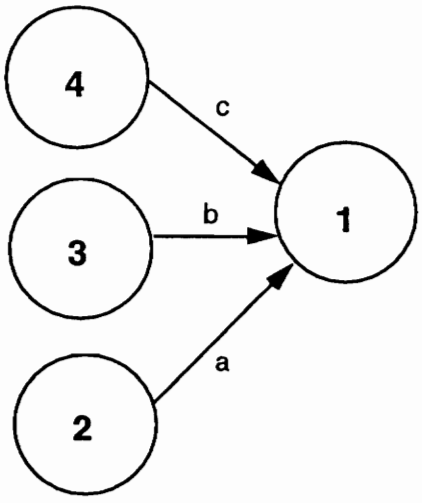
During propagation, certain internal signals are assigned values which need to be justified to the primary input ports. The justification scheme has been totally changed. The new procedure *Justify()* incorporates the following features:

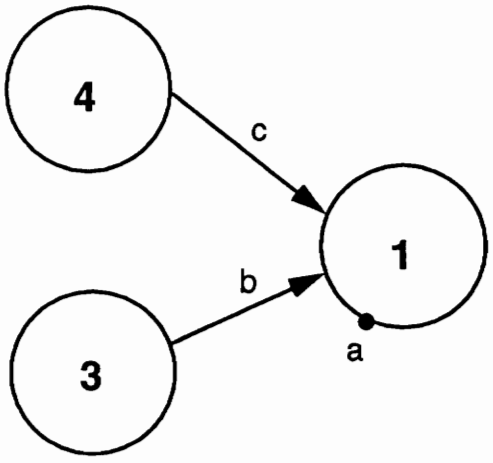
(1) The justification scheme allows more than one step of justification to take place, i.e., it allows models based on the structure shown below to be tested. In the example below, to propagate 'a' to the output, 'b' is assigned a constant value. This needs to be justified to process 3. As a result, 'c' is then assigned a value which further needs to be justified to process 4.



(2) The scheme allows more than one signal to be justified in the same time-frame. For example, models based on the structures shown below. In order to propagate 'a' to the output, both 'b' and 'c' are assigned constant values which need to be justified to processes 3 and 4 respectively.

(3) Combinations of 2 and 3 above.



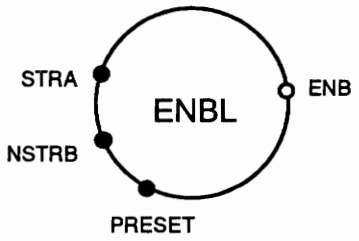


4.6 The Philosophy of the Primitive Tests

In our hierarchical test generation approach, primitive tests for each process of the process model graph are assumed to be precomputed and stored in the design library. These precomputed test sets are based on the functionality of the processes. They contain both the set of input patterns to the process and the output response expected for a correctly functioning module. These tests are then used to assemble the test for the whole entity. Thus, the basic primitive test philosophy can be stated as follows:

Given the set of precomputed tests containing a set of input test sequences and the corresponding expected output responses for each process in the PMG, the test sequence for the entire model can be hierarchically constructed from these primitive test sets.

Figure-19 shows the test data file for the ENBL module. The first line indicates the order in which the ports are stored in the linked-list of ports for the module. This order can be determined by means of the ***extract.c*** program explained in chapter 5. The order of the tests for the ports is assumed to be the same as the order of the ports in the linked-list. The second line is an *initialization frame* where the user can specify if an output needs to be initialized or not. In the above case, the output ENB of the module will be preset to logic '1' during the test generation. This is discussed in the next section in greater detail.




```
                    portorder: ENB PRESET NSTRB STRA
Initialization frame => 1
                        1
                        R 0 0 R
                        1
                        F 0 0 F
                        1
                        R 0 F 1
                        1
                        F 0 R 1
                        1
                        R R 0 0
                        1
                        F F 0 0
                        2
                        R 0 0 R
                        1 0 0 1
                        2
                        F 0 R 1
                        0 0 1 1
```

Figure-19: Test Data File for the module ENBL

The test sequences are given starting from the third line of the data file. Each test set is preceded by a number indicating the number of time-frames needed to assign that test. Symbolic values are used to represent the various tests. The meaning of the symbols are as follows:

'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' on a single-bit bus.

'F' : A transition from '1' to '0' on a single-bit bus.

'X' : Unknown value or don't care.

'Z' : High Impedance.

'D_i ': Any chosen value in either a single-bit bus or a multi-bit bus. The values can be D1, D2, D3, D4, etc.

'P' : a value which is the same as the value of the signal in the previous time-frame.

'C_i' : Values used to represent control signals, like the output of a counter. The values can be C1, C2, C3, C4, etc.

Each test data file needs to satisfy certain requirements. They are:

[1] In case of processes whose outputs need to be initialized, the output value needs to be specified in the initialization frame. If no initialization is required, this frame is left blank.

[2] For each sensitive port of the process, there must be a test that generates an event on that port. The test should also generate an event on the output port of the process.

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

[4] For an internal signal whose destination port is a sensitive input port, it is required to have a test that generates an event on the signal as well as a test that generates a constant value on the signal. However, for an internal signal whose destination port is not a sensitive port, only a test that generates a constant value on the signal is required. Hence, a test that generates an event on the output port of the process as well as a test that generates a constant value on the output port should be present, since the information about internal signals is not available for a process in isolation.

[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.

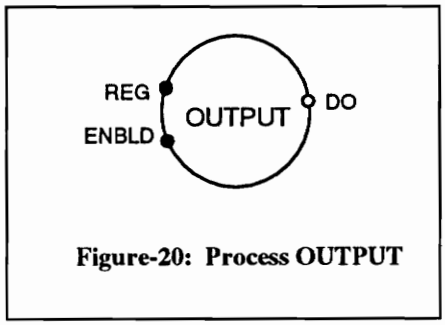


Figure-20: Process OUTPUT

The tests that have events on the sensitive signals guarantee that there is at least one test to activate each of the sensitive ports. Also, the requirement that in any time frame, only one input signal should have an event in the test data file, is important. For example, in the process OUTPUT in Figure-20, there are two sensitive input ports REG and ENBLD. In order to check the sensitivity of port ENBLD, a test with an event on ENBLD is needed. In addition, the test should have a constant signal value on port REG.

Otherwise, even if the process OUTPUT is executed, we cannot tell whether it is due to the event on port ENBLD or the event on port REG. Hence, the following test is selected to propagate the event on signal ENBLD to the output DO.

D0	ENBLD	REG
D1	R	D1

4.7 Incorporation of Sequential Primitives

A sequential primitive is one in whose output depends both on the present set of inputs to the primitive and its previous state. A general sequential primitive is shown in Figure-21. The use of purely gate-level structural detail makes the test generation problem for sequential circuits extremely complex and the test sequence very large. However, the use of high-level functional models reduces the complexity to a large extent.

In case of circuits containing sequential primitives, the test generation problem is similar to that of combinational circuits in that a set of test vectors has to be generated to activate a sensitive path and propagate the result to the output. However, in case of models containing sequential primitives, the sequential primitive first has to be placed in an initial state, and this state is used when propagating a value to the primary output [25]. This is because a particular set of inputs may cause the output of the process to change to a value which depends on the previous output. For example, in case of a counter, on the active edge of the clock, the output is incremented by one over its previous value.

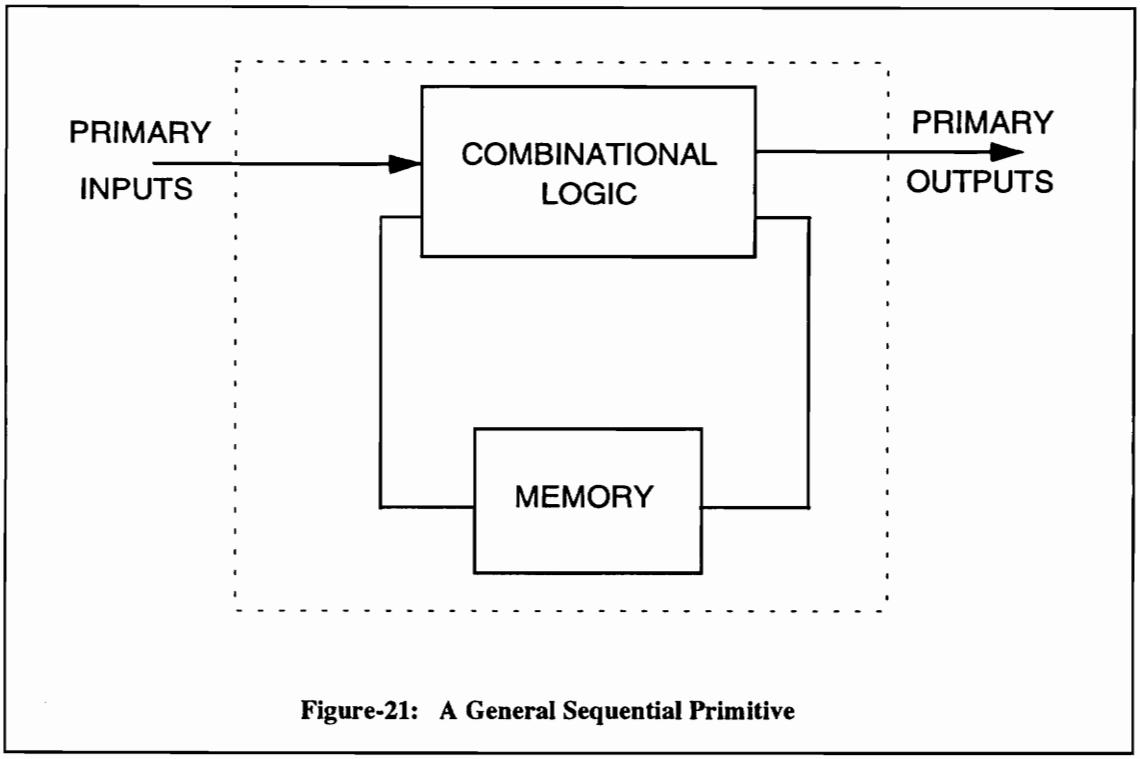


Figure-21: A General Sequential Primitive

In order to incorporate sequential primitives in the HBTG test generation process, a combination of automatic analytical techniques and user-interaction is adopted. Such a bilateral approach [26, 28] is commonly applied to provide the user with some steering capability. A provision is made for the user to specify the initialization value for each module in the process model graph. As explained in section 4.6, an *initialization frame* is created in the precomputed test data file stored in the library for each module. If this frame is left blank, no initialization is done for that module. If a specific value is indicated in the frame, the HBTG algorithm then proceeds to initialize the appropriate signal to that value before starting the actual test generation process. The initialization process is performed by the procedure *Initialize_Module.*

In case of the processes to be initialized, we assume the following:

(1) Synchronous primitives have a CLOCK port.

(2) Asynchronous controls such as PRESET and RESET are available wherever appropriate.

A macro called *PortNameValue* has been defined which assigns unique integers to special ports. The ports recognized are RESET or RST, PRESET, CLOCK or CLK and CLEAR or CLR. This macro is crucial in referencing the ports during the test generation process.

In order to initialize single-bit values, the user specifies a '1' or a '0' in the initialization frame. For example, in the test-data file for the module ENBLD in Figure-19, there is a '1' in the initialization frame. The output port is then set to '1' when the test generation process starts. This is accomplished by the procedure *Initialize_Single_Bit()*.

During the test generation process, initially all the signals are assumed to be in unknown state. When a process is encountered whose output needs to be initialized, the initialization process starts in time-frame 0. For example, in order to initialize port ENB of the process ENBL, the preset port is used to set the output to 1 as shown below:

frame	ENB	PRESET
0	R	R
1	1	0

In order to initialize BIT_VECTOR outputs in case of modules such as latches, registers, etc., the user specifies the initial output value (D1, D2, D3, etc.). For example, in order to set the output of a latch module to 'D2', the following process is performed by the procedure *Initialize_Vector* :

frame	REG	CLK	DATA
0	D2	F	D2
1	D2	0	D2

In case of circuits where control signals are present, symbolic values C_i (C1, C2, C3, etc.) have been defined as explained in section 4.5. A hierarchy is assumed to be present among the control signals where C1 can be clocked to C2, C2 can be clocked to C3, etc. i.e. C1 differs in one bit from C2, C2 differs in one bit from C3 and so on. For example, the assignments could be:

C1: 00 C3: 10
C2: 01 C4: 11

Any such set of assignments could be made, as long as they are consistent with the definition of the control signals.

In order to initialize a counter, we follow a typical approach [26, 27] in which the counter is first reset to a initial value and clocked to the required value specified in the initialization frame. For example, the initial state is assumed to be 'C1'. The following process is followed to initialize the counter to 'C3':

frame	COUNT	CLK	RESET
0	C1	X	R
1	C1	R	0
2	C2	F	0
3	C2	R	0
4	C3	F	0

For a model containing sequential primitives, once the test sequence has been generated for one time-frame, this information is used as the initial state information for the next time-frame. In order to allow this feature to be accessed within the precomputed test-data files, a symbolic value 'P' has been defined. Hence, whenever a process has a 'P' at the output, it remains at the same value as the previous time-frame. This is a crucial factor in allowing the test generation process to reference previous state information.

4.8 Test Generation through Resolved Ports

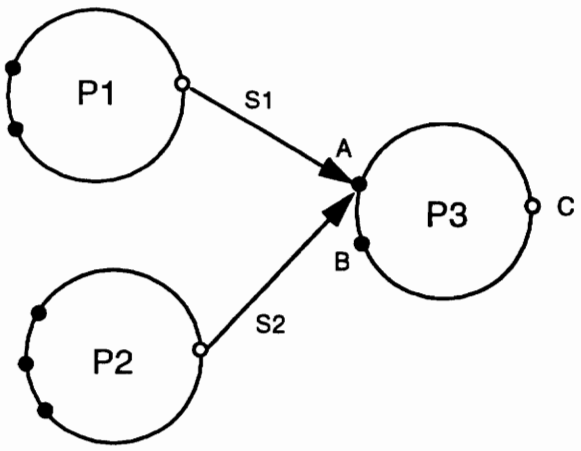
We know that in VHDL, a signal may contain multiple containers called *drivers*, and the value of the signal is a function of all the drivers [3]. The value of the signal is computed by a *resolution function.*

A simple and commonly used resolved type is the type TSL defined as:

```
type MVL is ('X','0','1','Z');
type MVL_VECTOR is array (INTEGER RANGE <> ) of MVL;
function BUSFUNC(INPUT: MVL_VECTOR) return MVL;
subtype TSL is BUSFUNC MVL;
type TSL_VECTOR is array (INTEGER RANGE <> ) of TSL;


function BUSFUNC(INPUT: MVL_VECTOR) return MVL is
  variable RES_VAL: MVL:='Z';
  begin
    for I in INPUT'RANGE loop
      if INPUT(I) /= 'Z' then
        RES_VAL := INPUT(I);
      end if;
    end loop;
    return RES_VAL;
  end BUSFUNC;
```

In order to perform test generation through resolved ports, it is necessary to recognize the existence of these ports in the model. Currently, the resolved types TSL and TSL_VECTOR are supported. In the figure below, the port A is a resolved port. Both the signals S1 and S2 make assignments to port A. Sensitive paths are constructed through port A from both processes P1 and P2. If an event occurs on S2, the event is propagated through port A to the output. A similar process occurs for an event on S1.



If a constant value is assigned to port A and needs to be justified, it is justified to the process which occurs earlier in the linked-list of processes. A similar procedure is followed if the port is of type TSL_VECTOR.

4.9 Test Generation for Models with Feedback

The presence of feedback is common in digital systems. In a VHDL model, the presence of a feedback signal causes unknown initial states in the model. Hence, it is necessary to initialize the feedback signal. Figure-22 shows feedback in a general VHDL behavioral model represented by its process model graph.

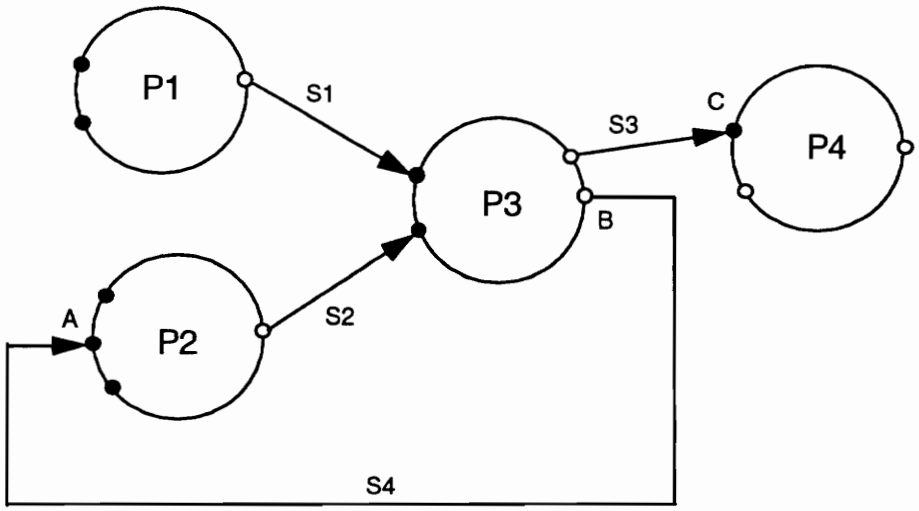


Figure-22A: A General VHDL Model with Feedback

In Figure-22, the signal S4 is fed back from port B to port A. Hence, it is necessary to initialize the signal S4 before the test generation process starts. This problem occurs due to the presence of sequential primitives in which the output depends on the initial state of the primitive.

The Modeler's Assistant recognizes the existence of feedback signals by means of an 'external' property where the signals are tagged as external to the model. In order to generate tests for models with feedback, a two-step approach is followed. A macro called *PortExt(port)* has been defined for each port of the PMG. This macro assigns a 1 to the source port of a feedback signal and a '2' to the destination port of the feedback signal. This macro allows us to reference primary inputs and outputs on the feedback path. The construction of sensitive paths is performed making use of the above macro. When a sensitive path reaches a primary output port that is the source of a feedback signal, the construction of the path stops at that point. This prevents the sensitive path construction from going into an infinite loop.

Once the sensitive paths have been constructed, the test generation process begins as the second step. During the TG process, the feedback signals are first selected and the user prompted for an initial value. The initial value is then reflected in the very first time frame, i.e. frame 0 of the test sequence. During the test generation process, the sequential primitives will use the initialized value as the previous state. The signal value assigned to the source of a feedback signal (output port) in one frame is used as the initial state for the destination of the feedback signal (input port) in the next time frame.

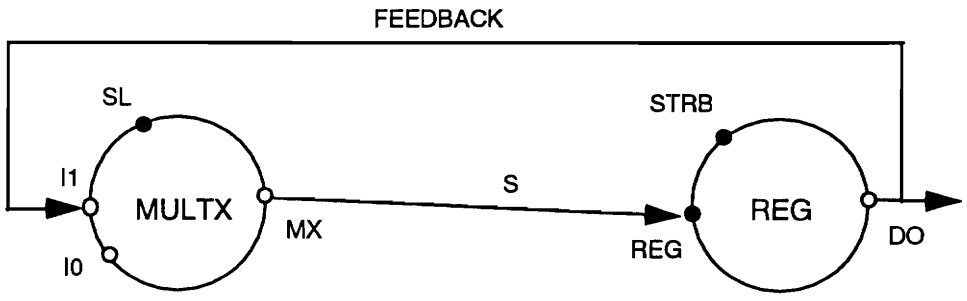


Figure-22B: Register Model with Feedback

In the above register model with feedback, the output DO is fedback to the input I1 of the multiplexer. Table-4 below shows the test sequence generated by HBTG for the above model. There are two sensitive paths:

SL => MX => REG => DO and

STRB => DO

In time frame 0, the user is prompted to enter an initial value, say '1'. Then, in time-frame 1, in order to activate the port SL, a test is chosen with an event on SL and MX, and having a value of '1' on the port I1. A similar process is followed for subsequent time-frames, where the tests chosen for the MULTX module reflect the value of the feedback signal (port DO) in the previous time-frame.

Table-4: Test Sequence for Register Model with Feedback

frame	MX	SL	I0	I1	DO	STRB
0	X	X	X	1	1	X
1	R	R	0	1	R	1
3	R	0	0	R	R	1
2	1	0	0	1	R	R

4.10 Information to the User

The main function of the HBTG algorithm is to exercise the VHDL behavioral model thoroughly. i.e. the test sequence should exercise as many functions of the VHDL model as possible. Once the test sequence has been constructed by HBTG, it is necessary that some information be provided to the user about the various functions that are exercised by the test sequence. This is because the test sequence is in the form of test-vectors which cannot be directly interpreted by the user.

The information is provided in two different steps. A macro called *PortAct* is defined for each port of the process model graph. This macro is used to reference activation information about each port. Initially, the *PortAct* for each port of the PMG is set to zero. In the test generation process, everytime an event is assigned to a sensitive port, the *PortAct* of that port is incremented by one. At the end of the test generation process, the *PortAct* of each port indicates how many times that port has been activated. This provides an indication of the number of times the corresponding process has been executed and is a representation of the capability of the generated test sequence.

Another system used to provide information to the user is the coverage environment of the VHDL system simulator. This system is discussed in detail in chapter 6. The results indicate the number of times each statement in the VHDL behavioral model is executed during simulation by applying the test patterns generated by HBTG.

Chapter 5. The Enhanced Hierarchical Test Generation Algorithm

5.1 Programming Environment

The previous algorithm was developed in ANSI-C on an APOLLO DN 3500 workstation. The current HBTG algorithm has been developed in Sun-C and is running on a Sun SPARCstation II. Since the Sun SPARC II does not directly support ANSI-C, a portion of the previous code had to be ported over to Sun-C.

The previous algorithm received information from the old version 2 of the Modeler's Assistant [19]. The Modeler's Assistant has since then been modified [20]. One of the features of the current HBTG algorithm is its ability to interface with the new version 3 of the Modeler's Assistant. The version 3 stores information in the PMG database in a different manner. Hence, a program had to be written to extract information from the PMG database of the Modeler's Assistant version 3.

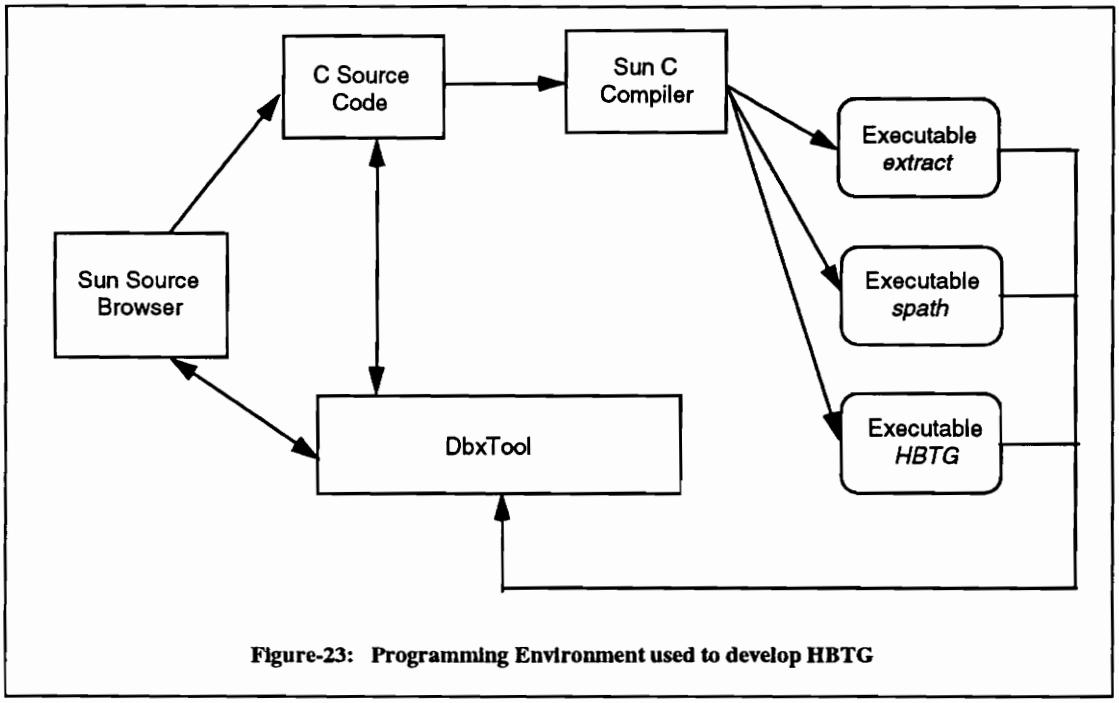


Figure-23: Programming Environment used to develop HBTG

Figure-23 shows the programming environment used in the development of the HBTG algorithm. Several tools were used to aid the software development effort. The Sun Source Browser [24] is a utility that allows the user to traverse and edit large pieces of code. For example, the user can reference all occurrences of a particular variable or pointer by simply performing a *query* on the item. This tool proved invaluable in maintaining the code which consists of 9 files and over 4000 lines of code.

The DbxTool was applied to debug the source code. This tool provides the basic debugging functions such as *stepping* through the code one line at a time and tracing the values of variables after each step. In case of a program crash, the DbxTool can also locate the line that caused the crash.

There are three executable files: **extract**, which extracts information from the database of the Modeler's Assistant (source file: extract.c), **spath** which constructs sensitive paths through the PMG (source file: spath.c) and **HBTG** which is the final test generation program (source file: testgen.c). The above programs along with the additional files are explained in the programmers manual in the appendix.

5.2 Test Generation Methodology

5.2.1 Assumptions

Certain assumptions have been made during the development of HBTG. They are listed below:

- The model to be tested is developed using the Modeler's Assistant. Hence, it is a single entity, single architecture VHDL behavioral model and all the constructs inside the architectural body are processes.

- Primitive tests for each process of the model are assumed to be available in the design library.

- The model has no reconvergent fanout structure.

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

5.2.2 Usage of the PMG of the VHDL Model

The graphical representation of a VHDL model, the process model graph, provides all the information about the model such as sensitivity of the ports, modes of each port, interconnection between the processes, etc. The only information that the PMG does not convey is about the functionality of each process in the PMG.

A process model graph G of a VHDL behavioral model is a directed-graph consisting of a non-empty set of processes $\{P(G)\}$ and a set of internal signals $\{S(G)\}$ such that each signal of G is associated with a pair of processes of G. Thus, the PMG is a directed-graph representation of *behavior* rather than a structural description of the model. This ability to clearly describe behavioral interconnections enables the PMG to be a direct representation of the division of functionality within the VHDL behavioral model. i.e. there is a *hierarchy* inherent in a VHDL behavioral model described by its process model graph. The test generation technique described in this thesis exploits this hierarchy. Hence, the process model graph of the VHDL behavioral model is used as the base for test generation.

5.2.3 Notion of *Effectiveness* of a Test Sequence

In our approach to test generation, no specific fault models are adopted. The main aim of HBTG is to construct a test sequence that exercises the VHDL model thoroughly.

We wish to generate an efficient test pattern to test different operations of the model. In order for the test to be *efficient*, the test sequence must be as short as possible and should check as many operations of the model as possible.

According to the semantics of a VHDL behavioral model, a process inside the architectural body of the behavioral model will be executed only if there is an *event* (change in signal value) on at least one of its sensitive input ports [3]. Hence, a test that generates events on the sensitive ports of a model is better than one that does not for the purpose of exercising the model. One major criterion used by the HBTG algorithm is: *The test sequence should activate as many sensitive ports of the model as possible.* As explained in section 3.2.2, "activate a port" in this context refers to the creation of an event on that port which generates a rise or fall in the single-bit bus or assigns different bit-vectors to a multi-bit bus.

Definition: An ***effective test sequence*** for a VHDL behavioral model is a test sequence that activates all the sensitive input ports of the model at least once.

Thus, if all the sensitive ports of the PMG of the VHDL model are activated at least once, we say that the test sequence is an effective test sequence. The above notion of effectiveness of a generated test sequence is a major factor in evaluating test quality. This is described in chapter 6 in greater detail.

5.2.4 Interface to the Modeler's Assistant

The process model graph of the VHDL model is developed interactively using the Modeler's Assistant. As explained in section 3.1, the Modeler's Assistant has a PMG database that stores information about the geometry of the PMG in terms of the various PMG elements such like processes, ports, signals, etc.

The different constructs of the PMG are termed as *nodes*. Hence, a node could be a unit, process, port, signal, constant, variable or generic. Currently, the number of nodes in a process model graph can be anything upto 800. Each node of the PMG is stored in a data structure ***a-node*** defined as

```
typedef struct
{
    char name[32];
    int type;
    int ptr[6];
    rect R;
} a_node;
```

The field ***name*** is used to store the name of the node. For example, if the node is a port called ENBLD, this field stores the string "ENBLD". The field ***type*** indicates the type of node i.e. whether the node is a port, module, signal, etc. The field ***ptr[6]*** is an array, each element of which points to a different related item in the PMG like the mode of a port, the next port in the linked list of ports, etc. The field ***rect*** is itself another data

structure which stores information about the coordinates of the display screen. The test generation algorithm HBTG does not make use of this field.

The PMG database of the Modeler's Assistant is actually a linked list of nodes. Figure-24 shows the linked list for the PMG created using the Modeler's Assistant in Figure-7 in chapter 3. A more general linked-list is shown in the appendix.

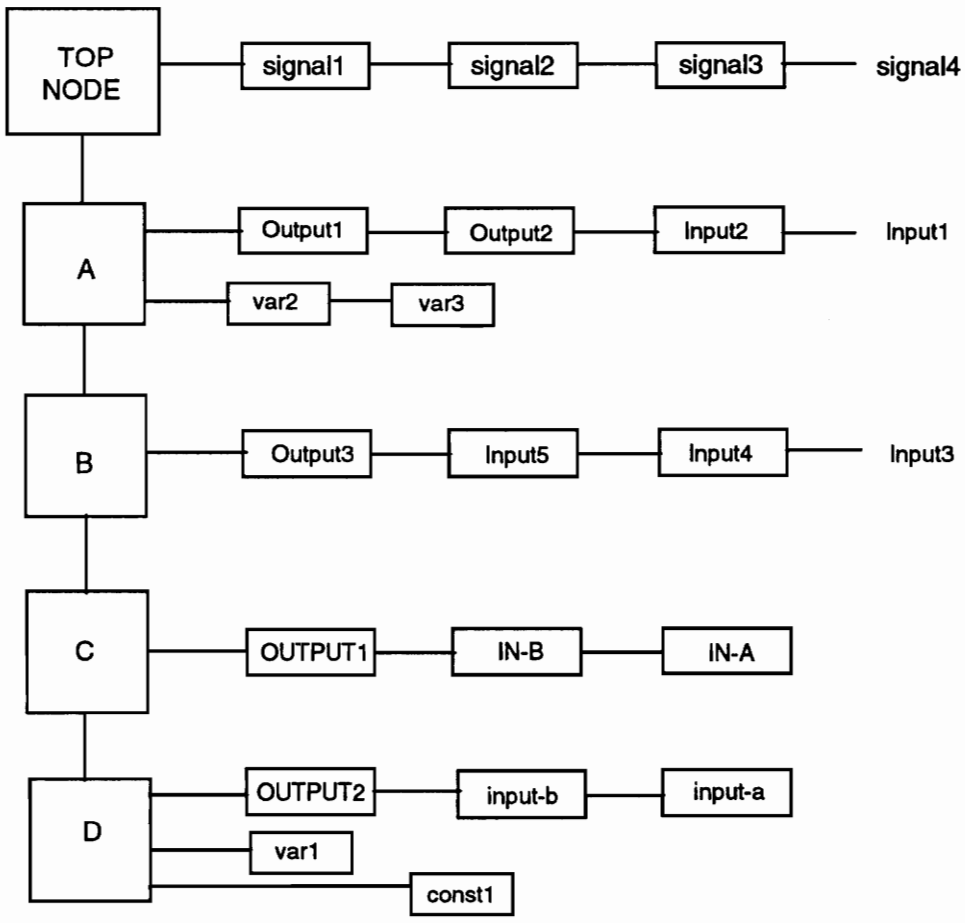


Figure-24: Linked-List Structure used to store information about example-pmg

A program called ***his.c*** is used to store information into the database described above.

The HBTG algorithm extracts information from the PMG database and uses it in the TG process. In order to extract information from the database, a program called ***extract.c*** has been written. This program uses various macros defined in the file ***macrosm.h*** in order to retrieve the information from the PMG database. Since each node of the PMG is stored in a structure of ***a_node***, the macros can obtain information by accessing the fields of the structure. Typical macros include *UnitName(unit)* which accesses the name of the top unit, *TopSignal(unit)* which points to the top signal in the linked list of signals and *NextPort(port)* which references the next port in the linked list of ports.

Figure-25 shows the PMG of an 8-bit latch created using the Modeler's Assistant. Figure-26 illustrates the VHDL code produced by the Modeler's Assistant for the model in Figure-25. The program ***extract.c*** that extracts information from the PMG database is shown in Figure-27. Figure-28 indicates the information retrieved from the database by the extract program.

In Figure-28, the modules in the top unit file and the ports in each module are shown along with their respective node numbers. The term *PortMode* indicates the mode of a port; ports with mode 2 are input ports, those with mode 4 are output ports and those with mode 0 are inout ports. The parameter *Sense* provides sensitivity information about the ports. If a port is sensitive, this parameter has a value of one. Else, it has a value of zero.

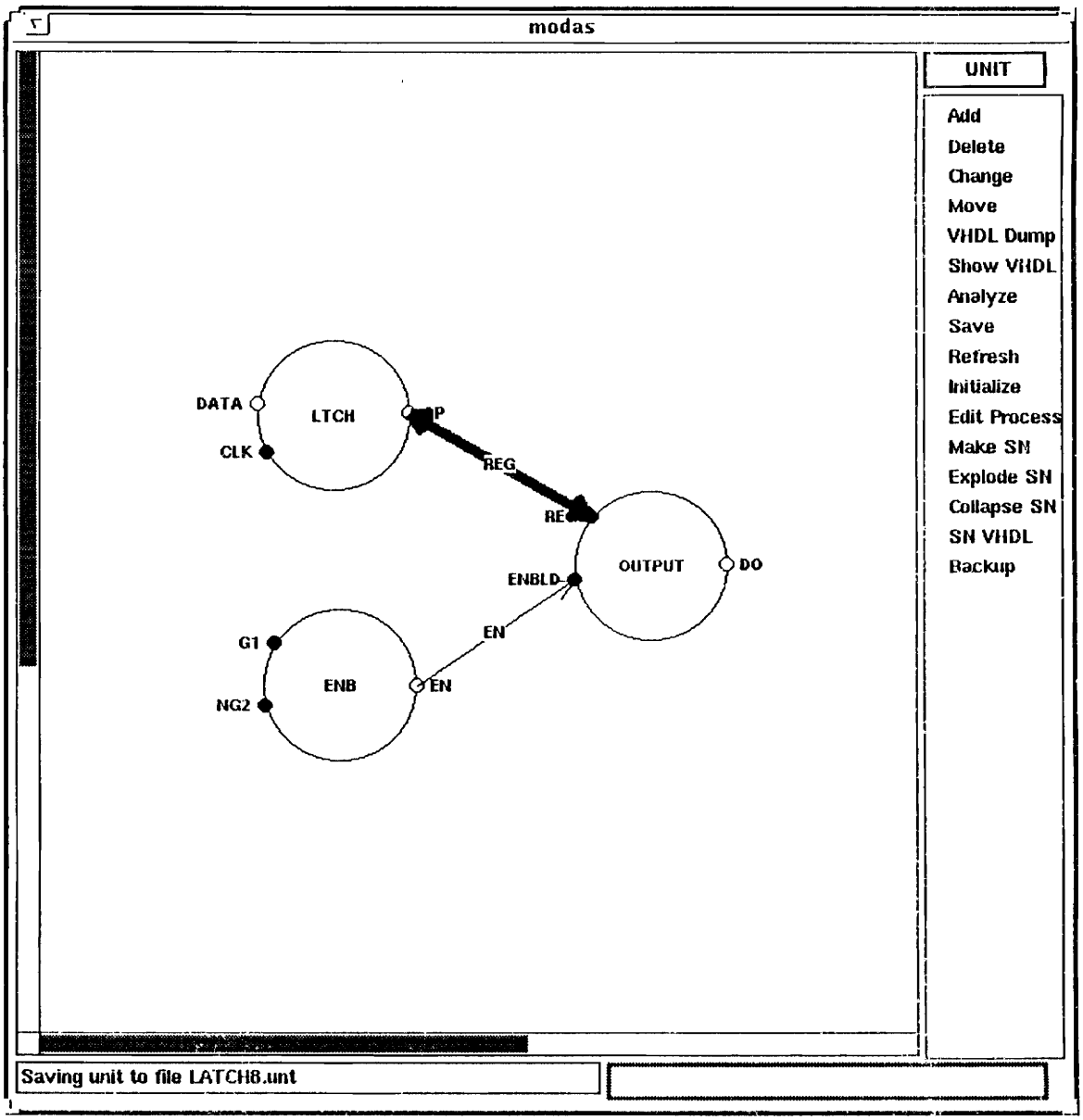


Figure-25: PMG of 8-bit Latch

```
use WORK.VHDLCAD.all, WORK.USER_TYPES.all;
-- ************************************************************
entity LATCH8 is
 port (DO: out MVL_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 Name: OUTPUT
 -- ------------------------------------------------------------
 OUTPUT_4: process (EN,REG)
 begin
   if (EN = '1') then
          DO <= BV_to_MVL(REG);
   else
          DO <= "ZZZZZZZZ";
   end if;
 end process OUTPUT_4;

 -- ------------------------------------------------------------
 -- Process Name: ENB
 -- ------------------------------------------------------------
 ENB_9: process (NG2,G1)
 begin
   EN <= G1 AND NOT NG2;
 end process ENB_9;

 -- ------------------------------------------------------------
 -- Process Name: LTCH
 -- ------------------------------------------------------------
 LTCH_14: process (CLK)
 begin
   if (CLK = '0') then
          REG <= DATA;
   else
          REG <= REG;
   end if;
 end process LTCH_14;

end BEHAVIORAL;
```

Figure-26: VHDL code produced by Modeler's Assistant for 8-bit Latch

```
/*****************************************************************************
  This file is used to extract information from the PMG database.
  The compile command is:
                        cc his.c extract.c -o extract
 *****************************************************************************/
#include <stdio.h>
#include "macrosm.h"
#include "vhdlm.h"
#include "externs.h"

a_node Node[MAX_NODES];
int FreeNodeCount, FreeList, top_node;

main()
{
   int top_node, next_mod, next_port, sigptr; num_port = 0, num_mod = 0;
   char unit_name[20];

   init_nodes();
   printf("Program to extract information from PMG database\n");
   printf("\n");
   printf("Please enter name of unit:\n");
   scanf("%s", unit_name);
   top_node = load_unit(unit_name);

   next_mod = TopOfModule(top_node);
   while (next_mod != __UnDefNode)
   {
     num_mod++;
     printf("ModuleName = %s\n",ModuleName(next_mod));
     printf("Module node number = %i\n", next_mod);
     next_port = NextPort(TopPort(next_mod));
     while (next_port !=__UnDefNode){
         num_port++;
        printf ("PortName= %s, Node number= %i, PortMode = %x, Sense = %x;\n",
              PortName(next_port),next_port,  PortMode(next_port), PortSense(next_port));
        sigptr = TopSignal(top_node);
        while (sigptr != __UnDefNode)
        {
           if (AbsPortLoc(SignalSrc(sigptr)) == next_port){
                              printf("PortSignalNode = %i\n", sigptr);
            }
          sigptr = NextSignal(sigptr);
        }
       next_port=NextPort(next_port);
     }
     NumPort(next_mod) = num_port;
     printf("Number of ports in module %s = %i\n\n", ModuleName(next_mod), NumPort(next_mod));
     num_port = 0;
```

--(continued on next page)---

```
    next_mod = NextModRef(next_mod);
  }
    NumMod(top_node) = num_mod;
    printf("Number of modules in the unit = %i\n", NumMod(top_node));
}
```

Figure-27: Program to extract information from the PMG database

```
Program to extract information from PMG database

Please enter name of unit:
LATCH8

ModuleName = OUTPUT
Module node number = 2
PortName= DO, Node number= 5, PortMode = 4, Sense = 0;
PortName= ENBLD, Node number= 7, PortMode = 2, Sense = 1;
PortName= REG, Node number= 8, PortMode = 2,  Sense = 1;
Number of ports in module OUTPUT = 3

ModuleName = ENB
Module node number = 9
PortName= EN, Node number= 11, PortMode = 4, Sense = 0;
PortSignalNode = 22
PortName= NG2, Node number= 14, PortMode = 2, Sense = 1;
PortName= G1, Node number= 15, PortMode = 2, Sense = 1;
Number of ports in module ENB = 3

ModuleName = LTCH
Module node number = 16
PortName= OP, Node number= 18, PortMode = 0, Sense = 0;
PortSignalNode = 3
PortName= CLK, Node number= 20, PortMode = 2,  Sense = 1;
PortName= DATA, Node number= 21, PortMode = 2,  Sense = 0;
Number of ports in module LTCH = 3

Number of modules in the unit = 3
```

Figure-28: Information extracted from the database by *extract.c*

5.3 Improved Sensitive Path Construction

During the test generation process, tests are selected from the primitive test sets that activate a sensitive port of a process. This causes the process to be executed and a result assigned to an output port of the process. The result must now be propagated to a primary output port of the model to be observed. In order to perform the above activation, sensitive paths are constructed through the process model graph as explained in section 3.2. We have defined a sensitive path as a directed path that starts at a sensitive primary input port and ends at a primary output port of the process with the intermediate ports along the path consisting of as many sensitive ports as possible. The creation of these sensitive paths is a dominant activity in the test generation process.

Cho and Armstrong [18] state that two processes in a VHDL model may be either **connected** or **not connected**. Two connected processes may further be strongly connected or weakly connected as defined below:

$(I_S)_{P_j}$: the set of signals which are inputs to process P_j and included in its sensitivity list.

$(I_{NS})_{P_j}$: the set of signals which are inputs to process P_j but are not included in its sensitivity list.

$(O)_{P_i}$: the set of signals that are outputs from P_i.

Definition: Two processes P_i and P_j are **connected** if $(O)_{P_i}$ and the union of $(I_S)_{P_j}$ and $(I_{NS})_{P_j}$ have a common element.

i.e. $(O)_{P_i} \cap \{(I_S)_{P_j} \cup (I_{NS})_{P_j}\} \notin \Phi$, where Φ represents the null set.

Definition: Two processes P_i and P_j are **strongly connected** if $(O)_{P_i}$ and $(I_S)_{P_j}$ have a common element.

i.e. $(O)_{P_i} \cap (I_S)_{P_j} \notin \Phi$

Definition: Two processes P_i and P_j are **weakly connected** if $(O)_{P_i}$ and $(I_{NS})_{P_j}$ have a common element, but $(O)_{P_i}$ and $(I_S)_{P_j}$ do not have a common element.

i.e. $(O)_{P_i} \cap (I_{NS})_{P_j} \notin \Phi$

$(O)_{P_i} \cap (I_S)_{P_j} \in \Phi$

Thus, we can see that a sensitive path constructed through the PMG goes through a series of connected processes. In an ideal case, any two processes are strongly connected. When two connected processes are strongly connected, the test sequence for that path will be fairly short.

5.3.1 The Algorithm for Sensitive Path Construction

The improved algorithm for sensitive path construction through the process model graph is shown in Figure-29. The construction of the sensitive paths begins when a sensitive primary input port is found, and ends at a primary output port of the model. One important criterion is:

For the set of sensitive paths $\{SP_i, i = 0, 1, 2 .....n\}$ constructed for a VHDL behavioral model, the first port on each sensitive path SP_i is always unique. i.e. the first port on a sensitive path x is always different from the first port on any other sensitive path y where $(x,y \in \{SP_i\})$.

```
program Construct_Sensitive_Path
begin
    while there is a sensitive input port not selected, do
        select the port as first port of the sensitive path;
        Repeat
            If process has only one output, then
                choose that output as next port along path;
                Select_Destn_Port;
          else
                choose that output port that is
                        on the least number of sensitive paths;
                Select_Destn_Port;
          end if;
        Until primary output is reached;
   end while;
end program;


procedure Select_Destn_Port
begin
     if more than one signal is leaving port, then
        choose the least activated of all
                destination ports as next port along sensitive path;
    else
        choose the destination port of
                the signal as next port along sensitive path;
    end if;
end procedure;
```

Figure-29 : Algorithm for Sensitive Path Construction

During the construction of a sensitive path *sp[i]*, a sensitive input port that is not the first port on any other path, is selected as the first port on the path *sp[i]*. If the process containing this port has only one output port, then that output port is selected as the next port on the path *sp[i]*. If there is more than one output, the next port on the path is selected based on the *choose_output_port* algorithm explained in section 4.2. This algorithm causes that output port which is on the least number of sensitive paths, to be selected as the next port on the path *sp[i]*. The procedure *Select_Destn_Port* is then invoked in order to select a destination port in case of multiple fanout. This port is then assigned as the next port on the path *sp[i]*. This process is continued till a primary output port of the model is reached. Once the primary output is reaches, the construction of path *sp[i]* is completed.

5.3.2 Implementation

The implementation of the above sensitive path algorithm is done by a program called ***spath.c***. This program uses the information extracted from the PMG database to perform the sensitive path construction. Various macros defined in the file ***macrosm.h*** are used in the sensitive path construction process. The macros *NumMod(unit)* and *NumPort(module)* represent the number of processes in the unit and the number of ports in a process (module) respectively. The macro *PortOrder(port)* assigns a value to each port of the PMG. Each primary input port and primary output port are assigned an unique port order. All ports connected to the same signal are assigned the same port order.

The above macros play a crucial role in the algorithm to optimize the sensitive path construction according to modules with multiple output and multiple fanout condition.

Figure-30 shows the sensitive paths constructed by the program ***spath.c*** for the PMG of the 8-bit latch in Figure-25. The paths are represented in two dimensional form. The first dimension indicates the sensitive path number. The order of the port on the sensitive path is represented by the second dimension. Thus, sp[3][4] is the third port on sensitive path 2.

```
Enter name of the unit:
LATCH8

No. of signals = 6

path 0:
sp[0][0]=14(NG2)
sp[0][1]=11(EN)
sp[0][2]=7(ENBLD)
sp[0][3]=5(DO)

path 1:
sp[1][0]=15(G1)
sp[1][1]=11(EN)
sp[1][2]=7(ENBLD)
sp[1][3]=5(DO)

path 2:
sp[2][0]=20(CLK)
sp[2][1]=18(OP)
sp[2][2]=8(REG)
sp[2][3]=5(DO)

No. of Spath = 3
```

Figure-30: Sensitive Paths constructed for the 8-bit latch

5.4 Creation of the Primitive Test Database

The precomputed tests for each module of the process model graph are stored as *.tst* files in the design library. Once the sensitive paths have been constructed, a program called ***input.c*** reads the test data files and stores them in a primitive test database.

```
portorder: OP CLK DATA
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-31: Test Data File for Process LTCH

Figure-31 above shows the test data file for the module LTCH of the 8-bit latch model in Figure-25. An array *Init[MAX_MOD][81]* is used to store the initialization information for each module of the process model graph. The test sets within the data file are stored in a linked-list of structures of *struct test* defined as:

```
struct test{
        int tstorder;
        int nu_frame;
        char tst_buffer[MAX_NU_FRAME][MAX_LINE_CHAR];
        struct test *ptr_to_next;
}
```

For each test-set, the field **tstorder** indicates the order of the test-set in the linked list starting from 0; **nu_frame** indicates the number of time-frames needed for the test; the array **tst_buffer** is a string used to store the test-set itself; the pointer **ptr_to_next** points to the next test in the linked-list of tests. Figure-32 shows the linked-list database created for the module LTCH.

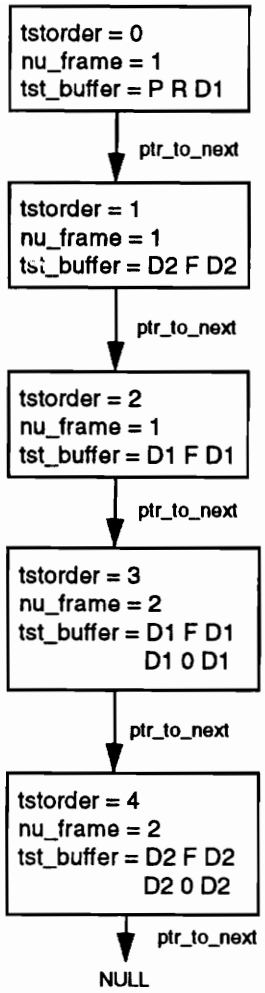


Figure-32: Linked-List Database of Test Sets for process LTCH

5.5 The Test Generation Process

Once the sensitive paths have been constructed through the PMG and the precomputed tests have been stored in the linked list database, the test generation process starts. This section discusses the test generation process.

5.5.1 Algorithm HBTG

The HBTG algorithm interfaces with the Modeler's Assistant to receive the process model graph. The basic psuedo-code for the algorithm is listed in Figure-33.

```
program HBTG

begin
   do Construct_Sensitive_Path;
   do Input_Tst;
   while there is a process not yet initialized, do
                call Initialize_Module;
   end while;
   while there is a sensitive path in the PMG not activated, do
         Select the path;
         Decide put;
         Activate the first port along the path;
         Increase the portact of the port;
         while Primary output of the path is not reached, do
               Decide put;
               If the front signal on the path is an event, then
                  If the signal is a sensitive signal, then
                       do propagation of the event;
                       Increase the portact of the port;
                  else
                       do propagation of the non-sensitive signal;
                       Increase the portact of the port;
                  ------( Continued on next page )--------
```

```
                end if;
                do implication;
            else
                if the front signal on the path depends on the previous frame, then
                        Decide the value of the signal in the current frame;
                        do propagation of the signal value;
                        Increase portact of the port on which an event has been assigned;
                        do implication;
                else
                        do propagation of a constant signal value;
                        Increase portact of the port on which an event has been assigned;
                        do implication;
                end if;
            end if;
        end while;
        while a process with known output (Os) and unknown input exists, do
            Select a sensitive path that traverses Os if possible;
            while the primary input on the path is not reached, do
                decide put;
                do signal justification towards primary input;
                do implication;
                Increase the portact of the port;
            end while;
        end while;
    end while;

end program HBTG;



program Input_Tst
begin
    while a module exists for which tests have not been stored in the database, do
    begin
        Store the Initialization frame into an array;
        Set the order of the test-sets to null;
        while there is a test-set not yet stored, do
        begin
            Save the nember of frames needed for the test-set;
            Store the test-set into the data-structure;
            Increment the order of test-sets in the linked-list;
            Set pointer of previous test-set to point to current test-set;
        end while;
    end while;
end program Input_Tst;
```

Figure-33: Algorithm HBTG

In the HBTG algorithm, sensitive paths are constructed through the PMG by the *Construct_Sensitive_Path* program explained in Figure-29. Primitive tests for each process are stored in the database by the program *Input_Tst* listed in the above figure. The processes are then initialized to the value specified in the precomputed test data files by the procedure *Initialize_Module.* In the test generation process, sensitive paths are selected and *activated* one by one until all the paths are activated. The activation of a path starts with activating the first port along the path. The signal value is then *propagated* to the primary output (PO) on the path, thereby activating the other sensitive ports along the path. Once the PO is reached, *justification* begins to justify the internal signal values specified during the forward activation process towards the primary inputs. *Implication* is performed to calculate the output values of a process when its inputs are specified. The test generation process is illustrated in Figure-34. Some terms used in the HBTG algorithm are defined below:

Definition: A ***front signal*** is the signal on the path selected that has been assigned a value most recently.

Definition: The ***put*** is the process under test. In the activation process, the ***put*** is the process in which the first port of the sensitive path is present. During propagation, the ***put*** is the process on the path that has the front signal as one of its input ports. During justification, the **put** is the process on the path selected that has the signal to be justified at its output port.

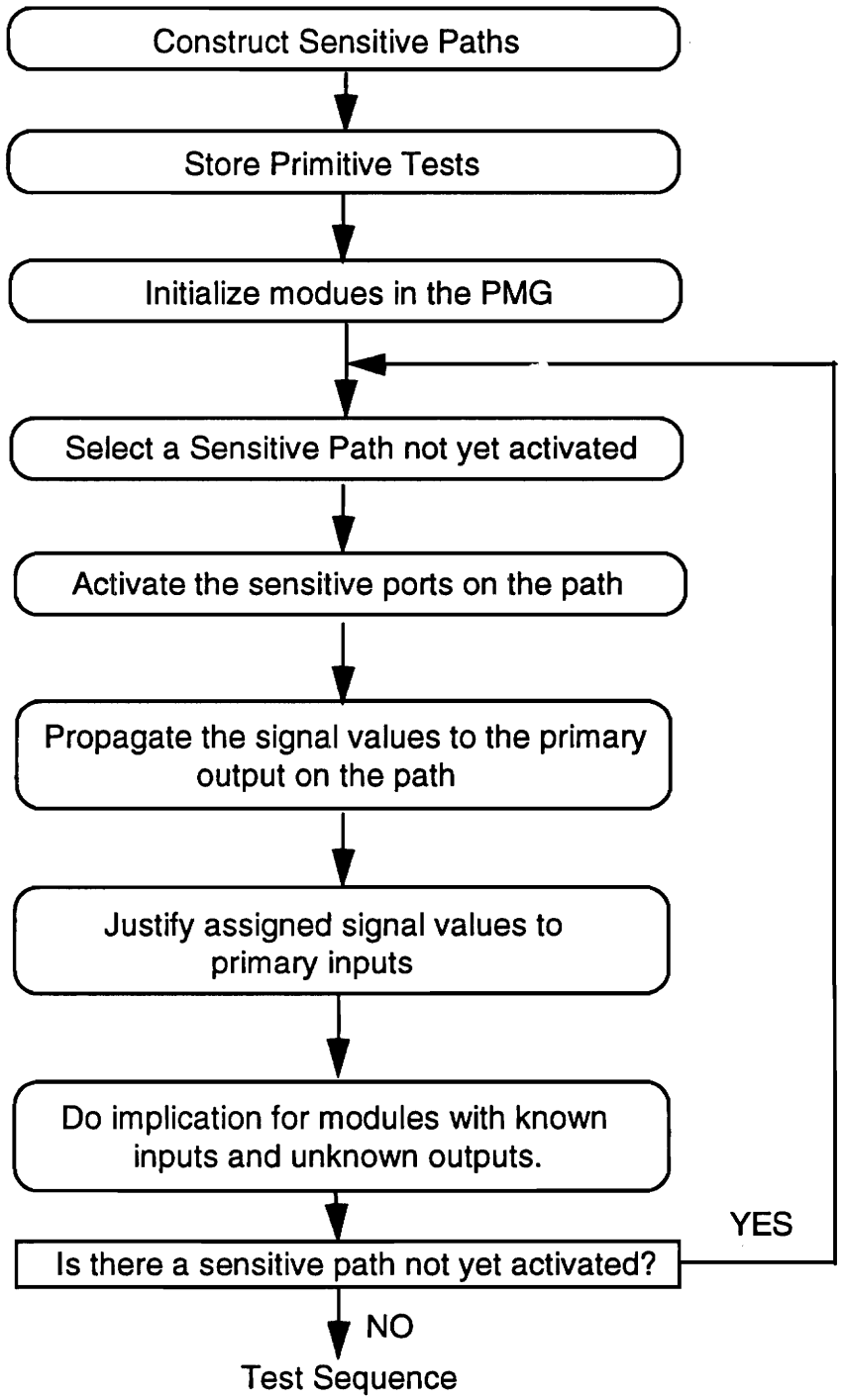


Figure-34: The HBTG Test Generation Process

Definition: The ***portact*** of a port is the number of times the test sequence has generated an event on the port. It is meaningful for sensitive input ports only.

During the test generation process, the HBTG algorithm selects appropriate test sequences from the primitive test files and instantiates them during the test generation process. For the different operations during the test generation process, different tests are selected. The selection of the tests are based on certain rules discussed below:

- 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 (the output port of the ***put***) 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 the 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 ***put***.

- 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.

During the test generation process, the procedure ***Select_Tst_Set*** selects appropriate tests for the various operations based on the above principles. Once the tests have been selected, the ***Assign_Value*** procedures assign values to the various signals. The final test sequence is a double linked-list of time-frames. For each time-frame, the signals are stored in a data-structure of struct *time_frame* defined as:

```
struct time_frame{
        int frameorder;
        struct time_frame *ptr_to_next;
        struct time_frame *ptr_to_last;
        int signal_value[100];
}frame;
```

In the above data structure, the field **frameorder** indicates the number of the current time frame; the pointer **ptr_to_next** points to the next structure in the double linked-list; the pointer **ptr_to_last** points to the previous time frame; the array **signal_value** is used to store the entire test sequence for a particular time frame.

A ***time_frame*** is the period of time required to complete the execution of all processes in a sensitive path caused by the activation of the path. It is assumed that a time frame is sufficiently long for the result of an activation to be propagated to a primary output. Once the test sequence has been generated for one time-frame, this information is

used as the initial state information for the next time-frame. Every time a sensitive path is activated, a new time-frame is created. Similarly, in the justification process, when more than one frame is needed for the justification, new frames are created and inserted before the current time frame.

5.5.2 A Test Generation Example

This section discusses the construction of a test sequence for the 8-bit latch of Figure-25. The PMG is first inputted through the Modeler's Assistant. Next, sensitive paths are constructed through the PMG. There are three sensitive paths:

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

In the final test sequence, the signal values are on the horizontal axis and the vertical axis shows the time-frames. Initially, all the signal values are "unknown" or "don't care".

frame	DO	EN	NG2	G1	OP	CLK	DATA
0	X	X	X	X	X	X	X

The first step in the test generation process is to initialize the modules in the PMG. According to the initialization frame in the precomputed test data file for the module LTCH in the PMG, the output of the process needs to be initialized to a value 'D2'. This is done in the first two time-frames, i.e. frames 0 and 1 as shown below.

frame	DO	EN	NG2	G1	OP	CLK	DATA
0	X	X	X	X	D2	F	D2
1	X	X	X	X	D2	0	D2

Once the initialization has been completed, the actual test generation process begins. The first step in this process is the activation of sensitive path 0. Activation of path 0 starts with the activation of the first port on the path i.e. port NG2. From the precomputed test file for module ENB, a test with an event on NG2 is selected, which is:

EN	NG2	G1
R	F	1

The test selected assigns an event 'R' to signal EN. This event now has to be propagated to the primary output on the path 0. i.e. DO. Hence, the following test is selected for the process OUTPUT:

DO	EN	REG
D1	R	D1

This test assigns a value 'D1' to the primary output DO. The justification process now begins to justify the signal value 'D1' on the internal signal REG. For this process, the following two time-frame test is selected from the precomputed test file for module LTCH:

OP	CLK	DATA
D1	F	D1
D1	0	D1

A two time-frame test is needed to guarantee that a constant value is assigned to signal REG (output port OP) in frame 2 in the fable shown below. Hence, the complete test sequence for activation of path 0 is :

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

As can be seen, the justification process inserts a time frame (frame 3) between frame 2 and frame 1. The frame numbers indicate the order of selection.

The activation of paths 1 and 2 proceeds in a similar manner to the activation of path 0. To activate port G1 of path 1, the following test is selected:

EN	NG2	G1
F	0	F

The signal values after activation of port G1 are:

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	X	F	0	F	D1	0	D1

In order to propagate the event on EN, the following test is chosen:

DO	EN	REG
Z	R	X

The signal values after propagation and implication are:

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

The final step is to activate path 2. This is done by activating port CLK. The test selected is:

OP	CLK	DATA
P	R	D1

The 'P' indicates that the output of the process LTCH depends on its previous value. Since each time-frame uses its previous time-frame as a initial reference, the port OP is assigned the same value 'D1' as in the previous time-frame. In order to propagate this value to the output, the following test is selected:

DO	EN	REG
D1	1	D1

Hence, the signal values after activation of port CLK and propagation of the value to the output DO are:

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
5	D1	1	X	X	D1	R	D1

In order to justify the signal on EN to process ENABLE, the following test is selected:

EN	NG2	G1
R	0	R
1	0	1

Hence, the signal values after the first step of justification are:

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
5	D1	1	0	1	D1	R	D1

Signal values after the second step of justification are:

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	X	R	1	R	D1	0	D1
5	D1	1	1	1	D1	R	D1

During implication for module OUTPUT, the following test is selected:

DO	EN	REG
D1	1	D1

Hence, the final test sequence generated by HBTG for the 8-bit latch model is as shown in table-4 below:

Table-5: Test Sequence for 8-bit Latch

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.

Figure-35 shows the PMG of the model CNTR constructed using the Modeler's Assistant. Figure-36 shows the sensitive paths constructed through the PMG. There are

five sensitive paths through the PMG. Table-5 shows the test sequence generated by HBTG for the model CNTR. In the table, time-frames 0 to 6 are used to initialize the counter CNT to the value 'C4' specified in its test data file. In frame 0, the output port COUNT is reset to 'C1' and is then clocked to the required value 'C4' over the next 6 frames. Time frames 8 and 7 are used to activate sensitive path 0. Paths 1 and 2 are activated in frames 9 and 10 respectively. Frames 12 and 11 represent the activation of path 3. The last sensitive path, i.e., sensitive path 4, is activated in frame 13.

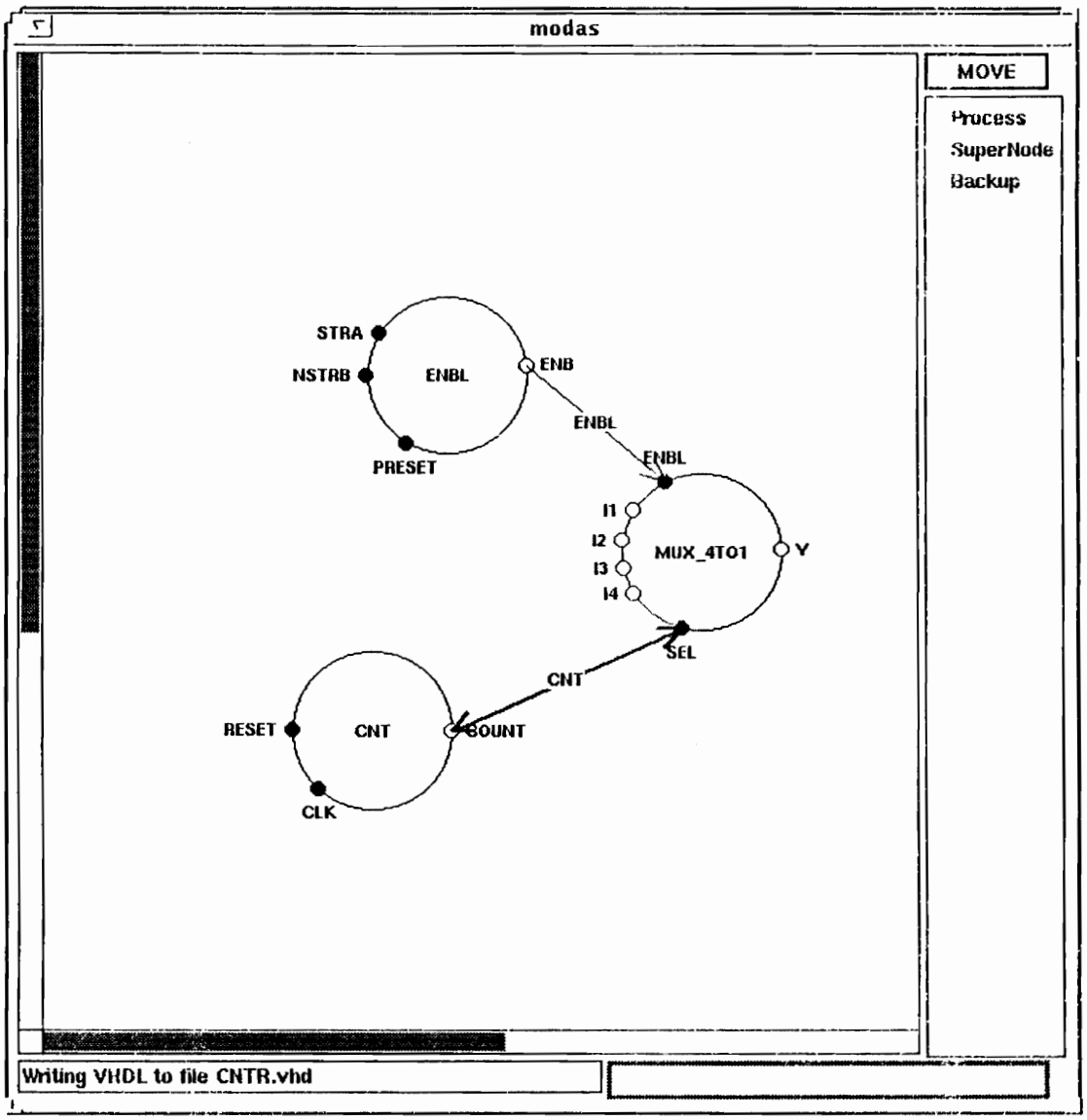


Figure-35: PMG of the model CNTR

Enter name of the unit:
CNTR

No. of signals = 11

path 0:
sp[0][0]=7(PRESET)
sp[0][1]=5(ENB)
sp[0][2]=14(ENBL)
sp[0][3]=12(Y)

path 1:
sp[1][0]=8(NSTRB)
sp[1][1]=5(ENB)
sp[1][2]=14(ENBL)
sp[1][3]=12(Y)

path 2:
sp[2][0]=9(STRA)
sp[2][1]=5(ENB)
sp[2][2]=14(ENBL)
sp[2][3]=12(Y)

path 3:
sp[3][0]=24(CLK)
sp[3][1]=22(COUNT)
sp[3][2]=15(SEL)
sp[3][3]=12(Y)

path 4:
sp[4][0]=25(RESET)
sp[4][1]=22(COUNT)
sp[4][2]=15(SEL)
sp[4][3]=12(Y)

No. of Spath = 5

Figure-36: Sensitive Path Construction for the model CNTR.

Table-6: Test Sequence for model CNTR

frame	ENB	PRESET	NSTRB	STRA	Y	I4	I3	I2	I1	COUNT	CLK	RESET
0	R	R	X	X	X	X	X	X	X	C1	X	R
1	1	0	X	X	X	X	X	X	X	C1	R	0
2	1	0	X	X	X	X	X	X	X	C2	F	0
3	1	0	X	X	X	X	X	X	X	C2	R	0
4	1	0	X	X	X	X	X	X	X	C3	F	0
5	1	0	X	X	X	X	X	X	X	C3	R	0
6	1	0	X	X	X	X	X	X	X	C4	F	0
8	1	0	X	X	X	X	X	X	X	C1	F	0
7	R	R	0	0	R	0	0	0	1	C1	0	0
9	F	0	R	1	F	1	1	1	1	C1	0	0
10	R	0	0	R	R	0	0	0	1	C1	0	0
12	R	0	0	R	R	0	0	0	1	C1	0	0
11	1	0	0	1	F	1	1	0	1	C2	F	0
13	R	0	0	1	R	0	0	0	1	C1	0	R

5.6 Databases used by HBTG

Figure-37 illustrates the various databases present in the HBTG environment. As we can see, there are three main databases. The first is the PMG database in the Modeler's Assistant which supplies information about the PMG. The second database is the primitive test database used to store the precomputed tests for each process in the PMG. The third database is the test-sequence database which stores the final test-sequence generated by HBTG.

The PMG database and the primitive test database are linked lists whereas the test sequence database is a double-linked list of data-structures.

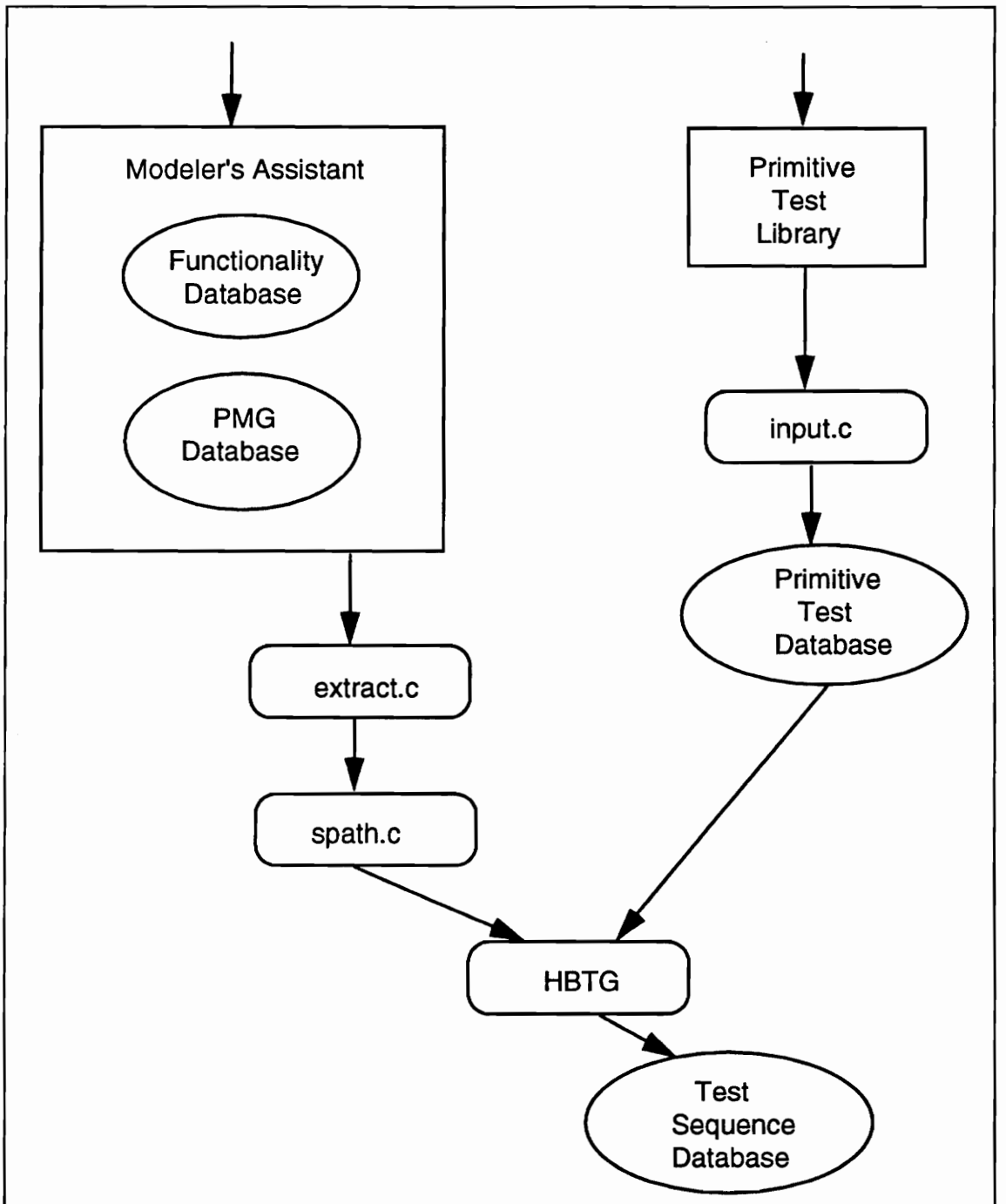


Figure-37: Databases in the HBTG Environment

Chapter 6. Evaluation of Test Quality

Once the test sequence for the VHDL behavioral model has been generated by HBTG, it is necessary to evaluate the quality of the generated test sequence. i.e. We wish to determine the *effectiveness* of the generated test pattern. One of the most important considerations in software testing is the construction of effective tests. This chapter discusses a scheme to evaluate the quality of the test sequence generated by the HBTG algorithm.

6.1 Software Testing Principles

This section illustrates certain important principles in software testing that have been formulated over the years. Many of these ideas have been incorporated into the HBTG algorithm.

6.1.1 Criteria for Testing

The construction of a test sequence basically follows a simple concept: select the model to test or the attribute to exercise; generate the test patterns (cases) that exercise the required attribute of function; simulate the model by applying the test pattern and observe the results in comparison to a standard or expected output. However, in the context of software testing, it is impossible [29] to have a complete test. This is because the total number of test cases for even the simplest model is extremely large. Hence, it is not possible to test all cases in practice. As a result, the testing must be based on some criteria. The goal is to find a subset of all possible tests that exercises a large number of functions of the model. The quality of tests generated depends strongly on the criteria on which the test generation process is based. As explained in the previous chapter, the criterion used by the HBTG algorithm is to activate as many sensitive ports of the VHDL behavioral model as possible.

6.1.2 Basic Methodologies

Several methodologies have been laid out in the domain of software engineering. Traditionally, some of these methods are adopted as the ground rules on which testing is based.

Path Testing is the name given to the family of testing techniques in which a set of test paths is constructed through the program or model. Exhaustive path testing, i.e. the execution of all possible paths through the model, is impractical if not impossible, due to

the enormous number of paths through even trivial models. However, by selecting a set of test paths properly, some measure of thoroughness can be achieved in the test generation process [30]. This idea of path testing is fundamental to many testing techniques.

Functional Testing, from the software testing point of view, refers to those techniques which construct test cases based on the functional description of the model. i.e., the tests generated are independent of the actual structure of code within the model itself. The entire model is viewed as a 'black box' and tests are typically constructed based on functional input/output specifications.

Structural Testing refers to the techniques which generate test patterns based on the actual structure of the model or the functionality within the model. An important factor in structural testing is the idea of *coverage*.

The term ***statement coverage*** refers to the property by which every statement in the model is executed at least once during simulation of the model. According to Myers [29], executing every statement is a necessary, though not sufficient, condition for test generation. Statement Coverage has been established as a minimum requirement in the IEEE unit test standard [30]. Thus, executing every statement in the model is an direct indication of the quality of the test sequence.

A stronger criterion to test generation than statement coverage is ***decision coverage*** which requires every decision within the module to have a true or false decision, in addition to every statement being executed. However, the test sequence to obtain decision coverage is invariably much longer than that for statement coverage.

6.2 System to Evaluate HBTG Test Quality

The HBTG algorithm incorporates several of the ideas explained in the previous section. In this section, a system is presented to evaluate the quality of the test pattern generated by HBTG.

A combination of path testing and statement coverage is used to determine test quality. The major criterion used by the HBTG algorithm is to activate as many sensitive ports of the model as possible, and the test generation process based on this criterion proceeds along the sensitive *paths* through the PMG. The test generation process is a kind of *top-down testing*, in which the TG starts at the topmost node (unit) and proceeds through each module (process). Within each process, various ports (sub-ordinate nodes) are activated and this process makes use of information within the ports such as sensitivity, etc.

An ***effective test sequence*** for a VHDL behavioral model has been defined as one that generates an event on every sensitive port of the model at least once. This notion of effectiveness is used to evaluate the test sequence. As explained, it is impossible to have a a 'complete' test. An effective test is one that reduces this incompleteness as much as possible. An effective test sequence for a VHDL model must satisfy the following conditions:

(1) It must test as many different functions of the model as possible.

(2) It must be 'efficient' in the sense that it must be as short as possible.

The test sequence generated by HBTG is converted into a test-bench file for simulation of the model. The rules to convert the test sequence into a test-bench are shown below. These rules are consistent with the criterion on which the test generation is based. During simulation, the test sequence is applied in lexical order.

- A time period of 5 ns is chosen for each time-frame. Since all the signal assignment statements within the model have delta delays, this value is sufficient to distinguish between the occurrence of different events. Thus, the time period for frame 0 is (1 ns to 5 ns), the frame 1 is assigned values in the period (6 ns to 10 ns), and so on. All the signals in one time-frame are assigned values during the corresponding time period.

- In each time period, those signals in the test sequence which are primary inputs (PIs) are assigned values.

- The various PIs are assigned values at times 1 ns, 4 ns, 6 ns, 9 ns, 11 ns, 14 ns, etc. This means that signal assignments are made at times (5*T + 1) ns and (5*T + 4) ns where $T \in \{0, 1, 2, 3.....\}$ is the lexical order of the time frame starting from 0.

- A constant value such as a BIT, BIT_VECTOR, TSL, etc. is assigned at time (5*T + 1) ns. A BIT_VECTOR signal can be assigned any bit_vector value of appropriate size. However, signals assigned 'D1' must be assigned a different value from those assigned 'D2', and so on. Thus, a signal 'REG' may be assigned a value"00001111" at time 21 ns in time-frame 4.

- In order to assign a 'F' to a signal, if the value of the signal is not a '1' in the previous time-frame, a '1' is assigned at time (5*T +1) ns and a '0' is assigned at time (5*T + 4) ns. Thus, in order to assign a 'F' to a signal ENBLD in time frame 3, a '1' is assigned to ENBLD at 16 ns and a '0' is assigned at 19 ns.

  To assign a 'R' to a signal, if the value of the signal is not '0' is the previous time - frame, the same procedure above is repeated with the 1s and 0s interchanged.

- In order to assign a 'F' to a signal, if the value of the signal is a '1' in the previous time frame, only a '0' is assigned to the signal at time (5*T + 4) ns.

- In case of control signals C1, C2, C3, etc., the signals are assigned values at time (5*T + 1) ns. The values assigned to the control signals must follow a definite order, where each value differs from its preceding value by one bit. Hence, for control signals C1, C2, C3, C4, a set of assignments could be:

C1: 10	C3: 00
C2: 11	C4: 01

- Since the last event in a time-frame occurs at time (5*T + 4) ns, the simulation results should be observed at that time.

For example, the test-bench file for the 8-bit Latch model using the test sequence in table-4 is shown in Figure-38.

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

entity LATCH8_TEST is
end LATCH8_TEST;

architecture TEST of LATCH8_TEST is
signal NDS2, DS1, CLK: BIT;
signal DATA: BIT_VECTOR(0 to 7);
signal DO:  MVL_VECTOR(0 to 7);

component LATCH8
 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: LATCH8 use entity work.LATCH8(BEHAVIORAL);

begin

R1:  LATCH8
    port map(DO, NDS2, DS1, CLK, DATA);

  process
  begin
          CLK <= transport '1' after 1 ns;
          CLK <= transport '0' after 4 ns;
          DATA <= transport "11110000" after 1 ns;

          CLK <= transport '1' after 11 ns;
          CLK <= transport '0' after 14 ns;
          DATA <= transport "00110011" after 11 ns;

          DS1 <= transport '1' after 16 ns;
          NDS2 <= transport '1' after 16 ns;
          NDS2 <= transport '0' after 19 ns;

          DS1 <= transport '0' after 24 ns;

          DS1 <= transport '0' after 26 ns;
          DS1 <= transport '1' after 29 ns;

          CLK <= transport '0' after 31 ns;
          CLK <= transport '1' after 34 ns;
      wait;

    end process;
end TEST;
```

Figure-38: Test-Bench for the 8-bit Latch.

The property of statement coverage is used to evaluate test quality. The test-bench file is used to simulate the model using the Synopsys VHDL system simulator [31]. 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. In order to incorporate the coverage environment, the following process is followed:

% vhdlan filename.vhd

% vhdlan test-bench.vhd

% vhdlsim test-bench

 # include .con file (optional)

 # coverage filename.vhd

 # run

 # quit

% coverage filename.cov

The % above refers to the general unix prompt while the # refers to the simulator prompt. Simulation results for the 8-bit Latch are shown in Figure-39. The coverage results for the 8-bit latch are shown in Figure-40. The number in the second column indicates the number of times the corresponding statement has been executed during simulation. As we can see, all the statements have been executed several times when the model is simulated using the test sequence generated by HBTG.

```
/
0 NS
  SMON4:   ACTIVE /LATCH8_TEST/DO (value = "ZZZZZZZZ")
1 NS
  SMON3:   ACTIVE /LATCH8_TEST/DATA (value = X"F0")
  SMON2:   ACTIVE /LATCH8_TEST/CLK (value = '1')
4 NS
  SMON2:   ACTIVE /LATCH8_TEST/CLK (value = '0')
  SMON4:   ACTIVE /LATCH8_TEST/DO (value = "ZZZZZZZZ")
11 NS
  SMON3:   ACTIVE /LATCH8_TEST/DATA (value = X"33")
  SMON2:   ACTIVE /LATCH8_TEST/CLK (value = '1')
14 NS
  SMON2:   ACTIVE /LATCH8_TEST/CLK (value = '0')
  SMON4:   ACTIVE /LATCH8_TEST/DO (value = "ZZZZZZZZ")
16 NS
   SMON:   ACTIVE /LATCH8_TEST/NG2 (value = '1')
  SMON1:   ACTIVE /LATCH8_TEST/G1 (value = '1')
19 NS
   SMON:   ACTIVE /LATCH8_TEST/NG2 (value = '0')
  SMON4:   ACTIVE /LATCH8_TEST/DO (value = "00110011")
24 NS
  SMON1:   ACTIVE /LATCH8_TEST/G1 (value = '0')
  SMON4:   ACTIVE /LATCH8_TEST/DO (value = "ZZZZZZZZ")
26 NS
  SMON1:   ACTIVE /LATCH8_TEST/G1 (value = '0')
29 NS
  SMON1:   ACTIVE /LATCH8_TEST/G1 (value = '1')
  SMON4:   ACTIVE /LATCH8_TEST/DO (value = "00110011")
31 NS
  SMON2:   ACTIVE /LATCH8_TEST/CLK (value = '0')
34 NS
  SMON2:   ACTIVE /LATCH8_TEST/CLK (value = '1')
```

Figure-39: Simulation Results for the 8-Bit Latch

```
        Copyright (c) 1990-1992 by Synopsys, Inc.
                    ALL RIGHTS RESERVED
        This program is proprietary and confidential information
        of  Synopsys, Inc. and may be used and disclosed only as
        authorized in a  license agreement  controlling such use
        and disclosure.

data date: Wed Apr  7 05:53:35 1993
simulation time: 34
coverage data: LATCH8.cov
VHDL source:   LATCH8.vhd

 Line     Count          Text
-------------------------------------------------------
   1
   2            use WORK.VHDLCAD.all, WORK.USER_TYPES.all;
   3            -- ***********************************************************
   4            entity LATCH8 is
   5             port (DO: out MVL_VECTOR(0 to 7);
   6               NG2: in BIT;
   7               G1: in BIT;
   8               CLK: in BIT;
   9               DATA: in BIT_VECTOR(0 to 7));
  10            end LATCH8;
  11            -- ***********************************************************
  12
  13            architecture BEHAVIORAL of LATCH8 is
  14
  15             signal EN: BIT;
  16             signal REG: BIT_VECTOR(0 to 7);
  17            begin
  18
  19              -- ---------------------------------------------------
  20              -- Process Name: OUTPUT
  21              -- ---------------------------------------------------
  22
  23              OUTPUT_4: process (EN,REG)
  24              begin
  25      6          if (EN = '1') then
  26      2                  DO <= BV_to_MVL(REG);
  27      4          else
  28      4                  DO <= "ZZZZZZZZ";
  29      6          end if;
  30
  31
  32              end process OUTPUT_4;
  33
  34
```

```
35              -- ----------------------------------------------------
36              -- Process Name: ENB
37              -- ----------------------------------------------------
38
39              ENB_9: process (NG2,G1)
40              begin
41      5           EN <= G1 AND NOT NG2;
42
43
44              end process ENB_9;
45
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-40: Coverage Results for the 8-bit Latch

In the same manner as above, the test sequence for the model CNTR in table-5 is converted into a test-bench file and used for simulation of the model. The coverage results obtained are shown in Figure-41.

```
        Copyright (c) 1990-1992 by Synopsys, Inc.
                    ALL RIGHTS RESERVED
        This program is proprietary and confidential information
        of  Synopsys, Inc. and may be used and disclosed only as
        authorized in a  license agreement  controlling such use
        and disclosure.

data date: Wed Apr  7 07:41:05 1993
simulation time: 69
coverage data: CNTR.cov
VHDL source:   CNTR.vhd

 Line      Count              Text
-------------------------------------------------------
   1
   2              use WORK.VHDLCAD.all, WORK.USER_TYPES.all;
   3              -- ***********************************************************
   4              entity CNTR is
   5                port (PRESET: in BIT;
   6                  NSTRB: in BIT;
   7                  STRA: in BIT;
   8                  Y: out BIT;
   9                  I4: in BIT;
  10                   I3: in BIT;
  11                   I2: in BIT;
  12                   I1: in BIT;
  13                   CLK: in BIT;
  14                   RESET: in BIT);
  15              end CNTR;
  16              -- ***********************************************************
  17
  18              architecture BEHAVIORAL of CNTR is
  19
  20                signal ENBL: BIT;
  21                signal CNT: BIT_VECTOR(0 to 1);
  22              begin
  23
  24                -- -------------------------------------------------
  25                -- Process Name: ENBL
  26                -- -------------------------------------------------
  27
  28                ENBL_4: process (PRESET,NSTRB,STRA)
  29                begin
  30       9          if (PRESET = '1') then
  31       2                ENBL <= '1';
  32       7          else
  33       7                ENBL <= STRA and not NSTRB;
  34       9          end if;
  35
```

```
36              end process ENBL_4;
37
38
39              -- ---------------------------------------------------
40              -- Process Name: MUX_4TO1
41              -- ---------------------------------------------------
42
43              MUX_4TO1_10: process (ENBL,CNT)
44              begin
45      16          if (ENBL = '1') then
46      11              case CNT is
47       6                      when "00" => Y <= i1;
48      10                      when "01" => I2;
49       4                      when "10" => I3;
50       2                      when "11" => I4;
51                      end case;
52      16          end if;
53
54
55              end process MUX_4TO1_10;
56
57
58              -- ---------------------------------------------------
59              -- Process Name: CNT
60              -- ---------------------------------------------------
61
62              CNT_19: process (CLK,RESET)
63              begin
64      14          if (RESET = '1') then
65       2                  CNT <= "00";
66      12          else
67      12            if (CLK = '0') then
68       7                  CNT <= INC(CNT);
69      12            end if;
70      14          end if;
71
72
73              end process CNT_19;
74
75            end BEHAVIORAL;
```

Figure-41: Coverage Results for model CNTR

Chapter 7. Results

The Hierarchical Behavioral Test Generator (HBTG) and the scheme to evaluate test quality have been discussed in the previous chapters. This chapter discusses the results obtained for various models.

The Process Model Graph of the models are created using the Modeler's Assistant. The primitive tests for each process of the PMG are precomputed and stored as *.tst* files in the library. After extraction of information from the database, sensitive paths are constructed through the PMG. The precomputed tests are then stored into the linked-list database. Once this is done, HBTG then generates the test sequence for the model. The test sequence is converted into a test-bench file based on the rules laid out in the previous chapter. The Synopsys VHDL system simulator is then used to estimate statement coverage obtained by simulating the model with the given test sequence.

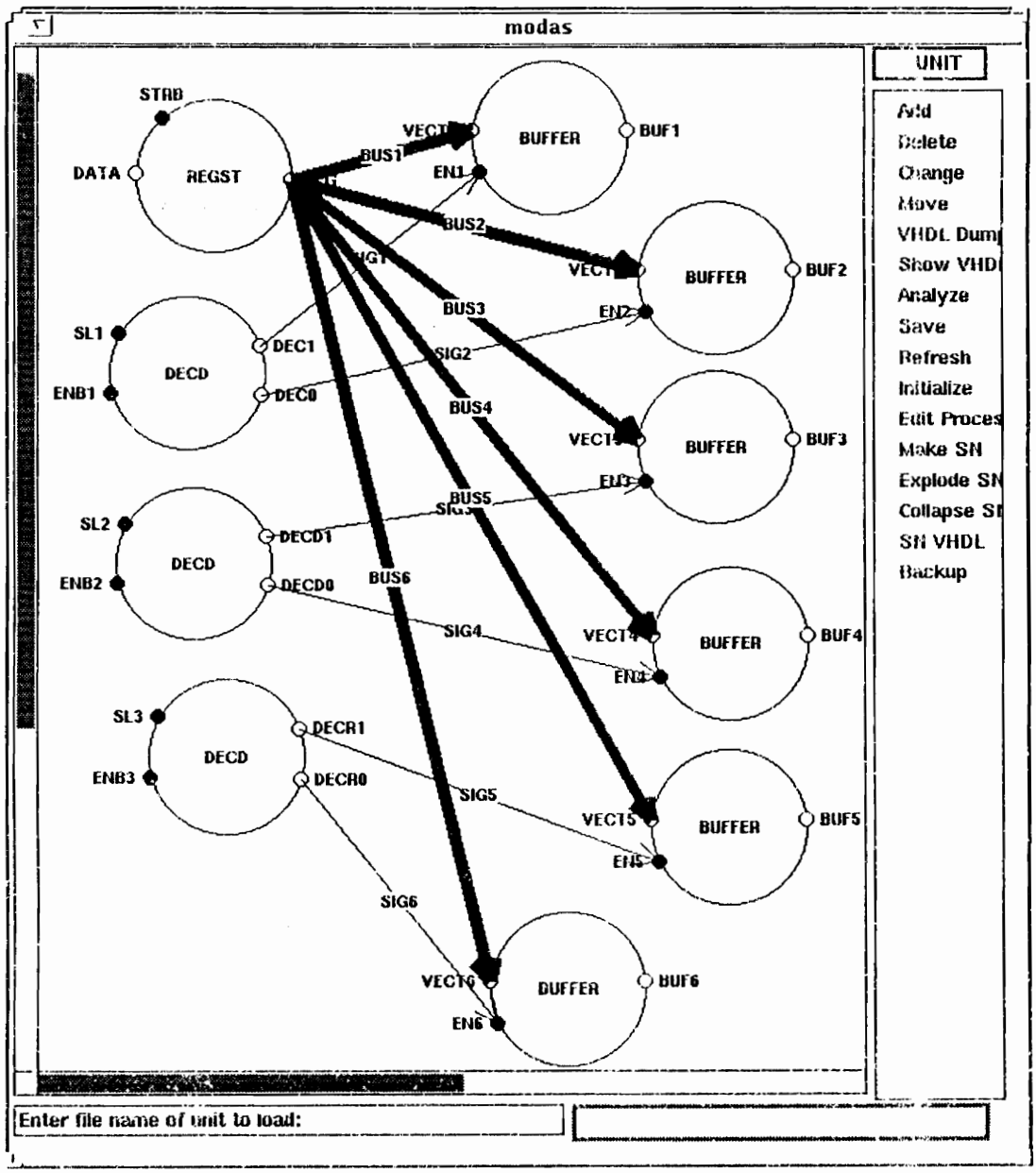


Figure-42: Model MSD (Part of a Microprocessor Development Board System)

```
Unit name is msd3
No. of signals = 21

Sensitive path 0:
sp[0][0]=8(ENB3)
sp[0][1]=5(DECR1)
sp[0][2]=34(EN5)
sp[0][3]=32(BUF5)

Sensitive path 1:
sp[1][0]=9(SL3)
sp[1][1]=7(DECR0)
sp[1][2]=28(EN6)
sp[1][3]=26(BUF6)

Sensitive path 2:
sp[2][0]=15(ENB2)
sp[2][1]=12(DECD1)
sp[2][2]=46(EN3)
sp[2][3]=44(BUF3)

Sensitive path 3:
sp[3][0]=16(SL2)
sp[3][1]=14(DECD0)
sp[3][2]=40(EN4)
sp[3][3]=38(BUF4)

Sensitive path 4:
sp[4][0]=22(ENB1)
sp[4][1]=19(DEC1)
sp[4][2]=58(EN1)
sp[4][3]=56(BUF1)

Sensitive path 5:
sp[5][0]=23(SL1)
sp[5][1]=21(DEC0)
sp[5][2]=52(EN2)
sp[5][3]=50(BUF2)

Sensitive path 6:
sp[6][0]=64(STRB)
sp[6][1]=62(REG)
sp[6][2]=29(VECT6)
sp[6][3]=26(BUF6)

No. of Spath = 7
```

Figure-43: Sensitive Path Construction for model MSD

Table-7: Test-Sequence for model MSD

frame	DECR0	DECR1	ENB3	SL3	DECD1	DECD0	ENB2	SL2	DEC1	DEC0	ENB1	SL1	BUF6	BUF5	BUF4	BUF3	BUF2	BUF1	REG	STRB	DATA
0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
2	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	D1	R	D1
1	R	Z	R	1	X	X	X	X	X	X	X	X	X	D1	X	X	X	X	D1	1	D1
3	F	R	1	F	X	X	X	X	X	X	X	X	D1	D1	X	X	X	X	D1	1	D1
4	0	1	1	0	R	Z	R	1	X	X	X	X	D1	D1	X	D1	X	X	D1	1	D1
5	0	1	1	0	F	R	1	F	X	X	X	X	D1	D1	D1	D1	X	X	D1	1	D1
6	0	1	1	0	0	1	1	0	R	Z	R	1	D1	D1	D1	D1	X	D1	D1	1	D1
7	0	1	1	0	0	1	1	0	F	R	1	F	D1	D1	D1	D1	D1	D1	D1	1	D1
8	F	R	1	0	F	R	1	0	F	R	1	0	D2	D1	D2	D1	D2	D1	D2	R	D2

```
                Copyright (c) 1990-1992 by Synopsys, Inc.
                        ALL RIGHTS RESERVED
        This program is proprietary and confidential information
        of  Synopsys, Inc. and may be used and disclosed only as
        authorized in a  license agreement  controlling such use
        and disclosure.

data date: Sat Apr 10 23:30:49 1993
simulation time: 39
coverage data: msd.cov
VHDL source:   msd.vhd

 Line      Count              Text
-------------------------------------------------------
   1
   2              use WORK.VHDLCAD.all, WORK.USER_TYPES.all;
   3              -- ***********************************************************
   4              entity msd is
   5                port (ENB3: in BIT;
   6                  SL3: in BIT;
   7                  ENB2: in BIT;
   8                  SL2: in BIT;
   9                  ENB1: in BIT;
  10                   SL1: in BIT;
  11                   BUF6: out MVL_VECTOR(0 to 7);
  12                   BUF5: out MVL_VECTOR(0 to 7);
  13                   BUF4: out MVL_VECTOR(0 to 7);
  14                   BUF3: out MVL_VECTOR(0 to 7);
  15                   BUF2: out MVL_VECTOR(0 to 7);
  16                   BUF1: out MVL_VECTOR(0 to 7);
  17                   STRB: in BIT;
  18                   DATA: in BIT_VECTOR(0 to 7));
  19              end msd;
  20              -- ***********************************************************
  21
  22              architecture BEHAVIORAL of msd is
  23
  24                signal SIG5: BIT;
  25                signal SIG6: BIT;
  26                signal SIG3: BIT;
  27                signal SIG4: BIT;
  28                signal SIG1: BIT;
  29                signal SIG2: BIT;
  30                signal BUS6: BIT_VECTOR(0 to 7);
  31                signal BUS5: BIT_VECTOR(0 to 7);
  32                signal BUS4: BIT_VECTOR(0 to 7);
  33                signal BUS3: BIT_VECTOR(0 to 7);
  34                signal BUS2: BIT_VECTOR(0 to 7);
  35                signal BUS1: BIT_VECTOR(0 to 7);
  36              begin
  37
  38                -- ---------------------------------------------------
  39                -- Process Name: DECD
  40                -- ---------------------------------------------------
```

```
41
42              DECD_4: process (ENB3,SL3)
43              begin
44      4           if ENB3 = '1' then
45      2               case (SL3) is
46      2                   when '0' => SIG6 <= '1';
47      2                   when '1' => SIG5 <= '1';
48                    end case;
49      4           end if;
50
51
52              end process DECD_4;
53
54
55              -- ---------------------------------------------------
56              -- Process Name: DECD
57              -- ---------------------------------------------------
58
59              DECD_10: process (ENB2,SL2)
60              begin
61      4           if ENB2 = '1' then
62      2               case (SL2) is
63      2                   when '0' => SIG4 <= '1';
64      2                   when '1' => SIG3 <= '1';
65                    end case;
66      4           end if;
67
68
69              end process DECD_10;
70
71
72              -- ---------------------------------------------------
73              -- Process Name: DECD
74              -- ---------------------------------------------------
75
76              DECD_16: process (ENB1,SL1)
77              begin
78      4           if ENB1 = '1' then
79      2               case (SL1) is
80      2                   when '0' => SIG2 <= '1';
81      2                   when '1' => SIG1 <= '1';
82                    end case;
83      4           end if;
84
85
86              end process DECD_16;
87
88
89              -- ---------------------------------------------------
90              -- Process Name: BUFFER
91              -- ---------------------------------------------------
92
93              BUFFER_22: process (SIG6)
94              begin
95      2           if SIG6 = '1' then
```

```
 96    1              BUF6 <= BV_to_MVL(BUS6);
 97    1          else
 98    1              BUF6 <= "ZZZZZZZZ";
 99    2          end if;
100
101
102            end process BUFFER_22;
103
104
105            -- ---------------------------------------------------
106            -- Process Name: BUFFER
107            -- ---------------------------------------------------
108
109            BUFFER_27: process (SIG5)
110            begin
111    2           if SIG5 = '1' then
112    1               BUF5 <= BV_to_MVL(BUS5);
113    1           else
114    1               BUF5 <= "ZZZZZZZZ";
115    2           end if;
116
117
118            end process BUFFER_27;
119
120
121            -- ---------------------------------------------------
122            -- Process Name: BUFFER
123            -- ---------------------------------------------------
124
125            BUFFER_32: process (SIG4)
126            begin
127    2           if SIG4 = '1' then
128    1               BUF4 <= BV_to_MVL(BUS4);
129    1           else
130    1               BUF4 <= "ZZZZZZZZ";
131    2           end if;
132
133
134            end process BUFFER_32;
135
136
137            -- ---------------------------------------------------
138            -- Process Name: BUFFER
139            -- ---------------------------------------------------
140
141            BUFFER_37: process (SIG3)
142            begin
143    2           if SIG3 = '1' then
144    1               BUF3 <= BV_to_MVL(BUS3);
145    1           else
146    1               BUF3 <= "ZZZZZZZZ";
147    2           end if;
148
149
150            end process BUFFER_37;
```

```
151
152
153             -- ------------------------------------------------------
154             -- Process Name: BUFFER
155             -- ------------------------------------------------------
156
157             BUFFER_42: process (SIG2)
158             begin
159     2           if SIG2 = '1' then
160     1                   BUF2 <= BV_to_MVL(BUS2);
161     1           else
162     1                   BUF2 <= "ZZZZZZZZ";
163     2           end if;
164
165
166             end process BUFFER_42;
167
168
169             -- ------------------------------------------------------
170             -- Process Name: BUFFER
171             -- ------------------------------------------------------
172
173             BUFFER_47: process (SIG1)
174             begin
175     2           if SIG1 = '1' then
176     1                   BUF1 <= BV_to_MVL(BUS1);
177     1           else
178     1                   BUF1 <= "ZZZZZZZZ";
179     2           end if;
180
181
182             end process BUFFER_47;
183
184
185             -- ------------------------------------------------------
186             -- Process Name: REGST
187             -- ------------------------------------------------------
188
189             REGST_52: process (STRB)
190             begin
191     4           if (STRB = '1') then
192     2                       BUS1  <= DATA;
193     4           end if;
194
195             end process REGST_52;
196
197
198         end BEHAVIORAL;
```

Figure-44: Coverage Results for model MSD

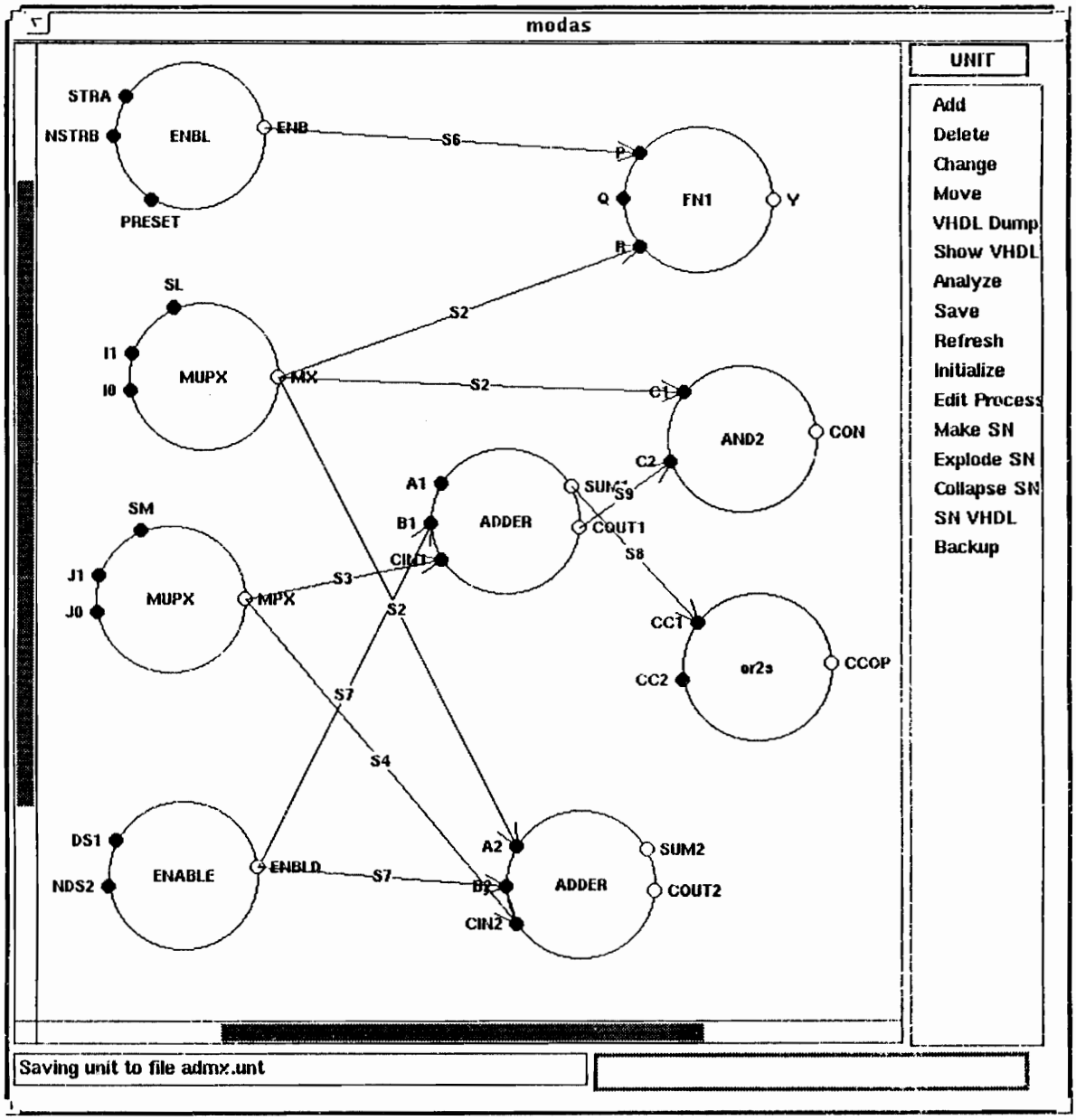


Figure-45: Model ADMX (an Adder-Multiplexer unit)

```
Enter name of the unit:
ADMX

Unit name is ADMX
No. of signals = 25


Sensitive path 0:
sp[0][0]=14(CC2)
sp[0][1]=11(CCOP)

Sensitive path 1:
sp[1][0]=19(NDS2)
sp[1][1]=17(ENBLD)
sp[1][2]=51(B1)
sp[1][3]=46(SUM1)
sp[1][4]=13(CC2)
sp[1][5]=11(CCOP)

Sensitive path 2:
sp[2][0]=20(DS1)
sp[2][1]=17(ENBLD)
sp[2][2]=42(B2)
sp[2][3]=37(SUM2)

Sensitive path 3:
sp[3][0]=25(PRESET)
sp[3][1]=23(ENB)
sp[3][2]=34(P)
sp[3][3]=30(Y)

Sensitive path 4:
sp[4][0]=26(NSTRB)
sp[4][1]=23(ENB)
sp[4][2]=34(P)
sp[4][3]=30(Y)

Sensitive path 5:
sp[5][0]=27(STRA)
sp[5][1]=23(ENB)
sp[5][2]=34(P)
sp[5][3]=30(Y)

Sensitive path 6:
sp[6][0]=33(Q)
sp[6][1]=30(Y)

Sensitive path 7:
sp[7][0]=52(A1)
sp[7][1]=49(COUT1)
```

```
sp[7][2]=7(C2)
sp[7][3]=5(CON)

Sensitive path 8:
sp[8][0]=58(SM)
sp[8][1]=55(MPX)
sp[8][2]=41(CIN2)
sp[8][3]=40(COUT2)

Sensitive path 9:
sp[9][0]=59(J0)
sp[9][1]=55(MPX)
sp[9][2]=50(CIN1)
sp[9][3]=46(SUM1)
sp[9][4]=13(CC1)
sp[9][5]=11(CCOP)

Sensitive path 10:
sp[10][0]=60(J1)
sp[10][1]=55(MPX)
sp[10][2]=50(CIN1)
sp[10][3]=49(COUT1)
sp[10][4]=7(C2)
sp[10][5]=5(CON)

Sensitive path 11:
sp[11][0]=66(SL)
sp[11][1]=63(MX)
sp[11][2]=8(C1)
sp[11][3]=5(CON)

Sensitive path 12:
sp[12][0]=67(I0)
sp[12][1]=63(MX)
sp[12][2]=32(R)
sp[12][3]=30(Y)

Sensitive path 13:
sp[13][0]=68(I1)
sp[13][1]=63(MX)
sp[13][2]=43(A2)
sp[13][3]=37(SUM2)

No. of Spath = 14
```

Figure-46: Sensitive Path Construction for model ADMX

Table-8: Test-Sequence for model ADMX

frame	SG0	SG1	SG2	SG3	SG4	SG5	SG6	SG7	SG8	SG9	SG10	SG11	SG12	SG13	SG14	SG15	SG16	SG17	SG18	SG19	SG20	SG21	SG22	SG23	SG24
0	X	X	X	X	X	X	R	R	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
1	X	X	X	X	X	X	1	0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
4	X	X	X	F	0	F	1	0	X	X	X	X	X	X	X	X	X	R	0	R	0	X	X	X	X
3	X	X	X	0	0	0	1	0	X	X	X	X	X	X	F	R	R	1	0	1	0	X	X	X	X
2	F	F	X	0	0	0	1	0	X	X	X	X	X	X	0	1	1	1	0	1	0	X	X	X	X
6	F	0	X	F	0	0	1	0	X	X	X	X	X	X	0	1	1	F	1	1	F	X	X	X	X
5	F	0	X	R	F	1	1	0	X	X	X	X	X	X	F	R	1	0	1	1	0	X	X	X	X
8	F	0	R	R	0	1	1	0	X	X	X	X	X	X	F	R	1	F	1	1	0	R	0	R	0
7	0	0	1	F	0	F	1	0	X	X	X	X	R	F	0	1	1	0	1	1	0	1	0	1	0
10	F	0	F	F	0	0	1	0	X	X	X	X	1	0	R	F	1	F	1	1	0	F	1	1	F
9	0	0	0	0	0	0	R	R	0	0	R	1	1	0	0	1	1	0	1	1	0	0	1	1	0
11	0	0	0	0	0	0	R	0	F	1	R	1	1	0	0	1	1	0	1	1	0	0	1	1	0
12	0	0	0	0	0	0	R	0	0	R	R	1	1	0	0	1	1	0	1	1	0	0	1	1	0
14	F	0	F	F	0	0	R	0	0	R	R	1	1	0	R	F	1	F	1	1	0	F	1	1	0
13	0	0	0	0	0	0	1	0	0	1	R	R	1	0	0	1	1	0	1	1	0	0	1	1	0
16	F	0	R	F	0	F	R	0	0	1	R	1	1	0	0	1	1	R	0	R	0	R	0	R	0
15	0	0	R	0	0	0	1	0	0	1	1	1	1	0	F	R	R	1	0	1	0	1	0	1	0
18	F	0	F	R	0	R	R	0	0	1	1	1	1	0	0	1	1	R	0	1	0	F	1	1	F
17	0	0	1	1	0	1	1	0	0	1	1	1	F	R	0	1	1	R	R	0	1	0	1	1	0
19	F	0	1	1	0	1	1	0	0	1	1	1	0	1	F	R	0	R	0	R	0	0	1	1	0
21	F	0	R	R	0	1	R	0	0	1	1	1	0	1	0	1	0	R	0	1	0	R	0	R	0
20	0	0	R	1	0	1	1	0	0	1	1	1	0	1	F	R	0	R	1	0	R	1	0	1	0
24	F	0	R	F	0	F	R	0	0	1	1	1	0	1	0	1	0	R	0	R	0	R	0	1	0
23	F	0	R	0	0	0	R	0	0	1	1	1	F	R	F	R	R	1	0	1	0	R	0	1	0
22	0	0	R	0	0	1	1	0	0	1	1	1	0	1	0	1	1	1	1	0	1	R	R	0	1
25	0	0	1	0	0	1	1	0	0	1	F	1	0	1	0	1	1	1	1	0	1	R	0	R	0
26	0	0	1	0	0	1	1	0	0	1	0	1	F	R	0	1	1	1	1	0	1	R	1	0	R

Legend for table-8:

SG0 = CC2

SG1 = CCOP

SG2 = CON

SG3 = ENBLD

SG4 = NDS2

SG5 = DS1

SG6 = ENB

SG7 = PRESET

SG8 = NSTRB

SG9 = STRA

SG10 = Y

SG11 = Q

SG12 = SUM2

SG13 = COUT2

SG14 = SUM1

SG15 = COUT1

SG16 = A1

SG17 = MPX

SG18 = SM

SG19 = J0

SG20 = J1

SG21 = MX

SG22 = SL

SG23 = I0

SG24 = I1

```
                Copyright (c) 1990-1992 by Synopsys, Inc.
                        ALL RIGHTS RESERVED
        This program is proprietary and confidential information
        of  Synopsys, Inc. and may be used and disclosed only as
        authorized in a  license agreement  controlling such use
        and disclosure.

data date: Sun Apr 11 04:25:57 1993
simulation time: 109
coverage data: ADMX.cov
VHDL source:   ADMX.vhd

 Line      Count             Text
-------------------------------------------------------
   1
   2              use WORK.VHDLCAD.all, WORK.USER_TYPES.all;
   3              -- ************************************************************
   4              entity ADMX is
   5                port (CCOP: out BIT;
   6                  CC2: in BIT;
   7                  CON: out BIT;
   8                  NDS2: in BIT;
   9                  DS1: in BIT;
  10                   PRESET: in BIT;
  11                   NSTRB: in BIT;
  12                   STRA: in BIT;
  13                   Y: out BIT;
  14                   Q: in BIT;
  15                   SUM2: out BIT;
  16                   COUT2: out BIT;
  17                   A1: in BIT;
  18                   SM: in BIT;
  19                   J0: in BIT;
  20                   J1: in BIT;
  21                   SL: in BIT;
  22                   I0: in BIT;
  23                   I1: in BIT);
  24              end ADMX;
  25              -- ************************************************************
  26
  27              architecture BEHAVIORAL of ADMX is
  28
  29                signal S8: BIT;
  30                signal S9: BIT;
  31                signal S2: BIT;
  32                signal S7: BIT;
  33                signal S6: BIT;
  34                signal S4: BIT;
  35                signal S3: BIT;
  36              begin
  37
  38                -- ---------------------------------------------------
  39                -- Process Name: or2s
  40                -- ---------------------------------------------------
```

```
41
42              or2s_8: process (CC2,S8)
43              begin
44      16          CCOP <= (S8 or CC2);
45
46
47              end process or2s_8;
48
49
50              -- ---------------------------------------------------
51              -- Process Name: AND2
52              -- ---------------------------------------------------
53
54              AND2_4: process (S9,S2)
55              begin
56       9          CON <= S2 and S9;
57
58
59              end process AND2_4;
60
61
62              -- ---------------------------------------------------
63              -- Process Name: ENABLE
64              -- ---------------------------------------------------
65
66              ENABLE_14: process (NDS2,DS1)
67              begin
68       8          S7 <= DS1 and not NDS2;
69
70
71              end process ENABLE_14;
72
73
74              -- ---------------------------------------------------
75              -- Process Name: ENBL
76              -- ---------------------------------------------------
77
78              ENBL_19: process (PRESET,NSTRB,STRA)
79              begin
80      10          if (PRESET = '1') then
81       2              S6 <= '1';
82       8          else
83       8              S6 <= STRA and not NSTRB;
84      10          end if;
85
86              end process ENBL_19;
87
88
89              -- ---------------------------------------------------
90              -- Process Name: FN1
91              -- ---------------------------------------------------
92
93              FN1_25: process (S2,Q,S6)
94              begin
95      17          Y <=  S6 and Q and not S2;
```

```
96
97              end process FN1_25;
98
99
100             -- ---------------------------------------------------
101             -- Process Name: ADDER
102             -- ---------------------------------------------------
103
104             ADDER_31: process (S4,S7,S2)
105             begin
106     11          SUM2 <= S2 xor S7 xor S4;
107     11          COUT2 <= (S2 and S7) or (S2 and S4) or (S7 and S4);
108
109
110             end process ADDER_31;
111
112
113             -- ---------------------------------------------------
114             -- Process Name: ADDER
115             -- ---------------------------------------------------
116
117             ADDER_38: process (S3,S7,A1)
118             begin
119     16          S8 <= A1 xor S7 xor S3;
120     16          S9 <= (A1 and S7) or (A1 and S3) or (S7 and S3);
121
122
123             end process ADDER_38;
124
125
126             -- ---------------------------------------------------
127             -- Process Name: MUPX
128             -- ---------------------------------------------------
129
130             MUPX_45: process (SM,J0,J1)
131             begin
132     13          case SM is
133     12              when '0' => S3 <= J0;
134     14              when '1' => S3 <= J1;
135                 end case;
136
137
138             end process MUPX_45;
139
140
141             -- ---------------------------------------------------
142             -- Process Name: MUPX
143             -- ---------------------------------------------------
144
145             MUPX_51: process (SL,I0,I1)
146             begin
147     10          case SL is
148     16              when '0' => S2 <= I0;
149      4              when '1' => S2 <= I1;
150                 end case;
```

```
151
152
153              end process MUPX_51;
154
155
156            end BEHAVIORAL;
```

Figure-47: Coverage Results for model ADMX

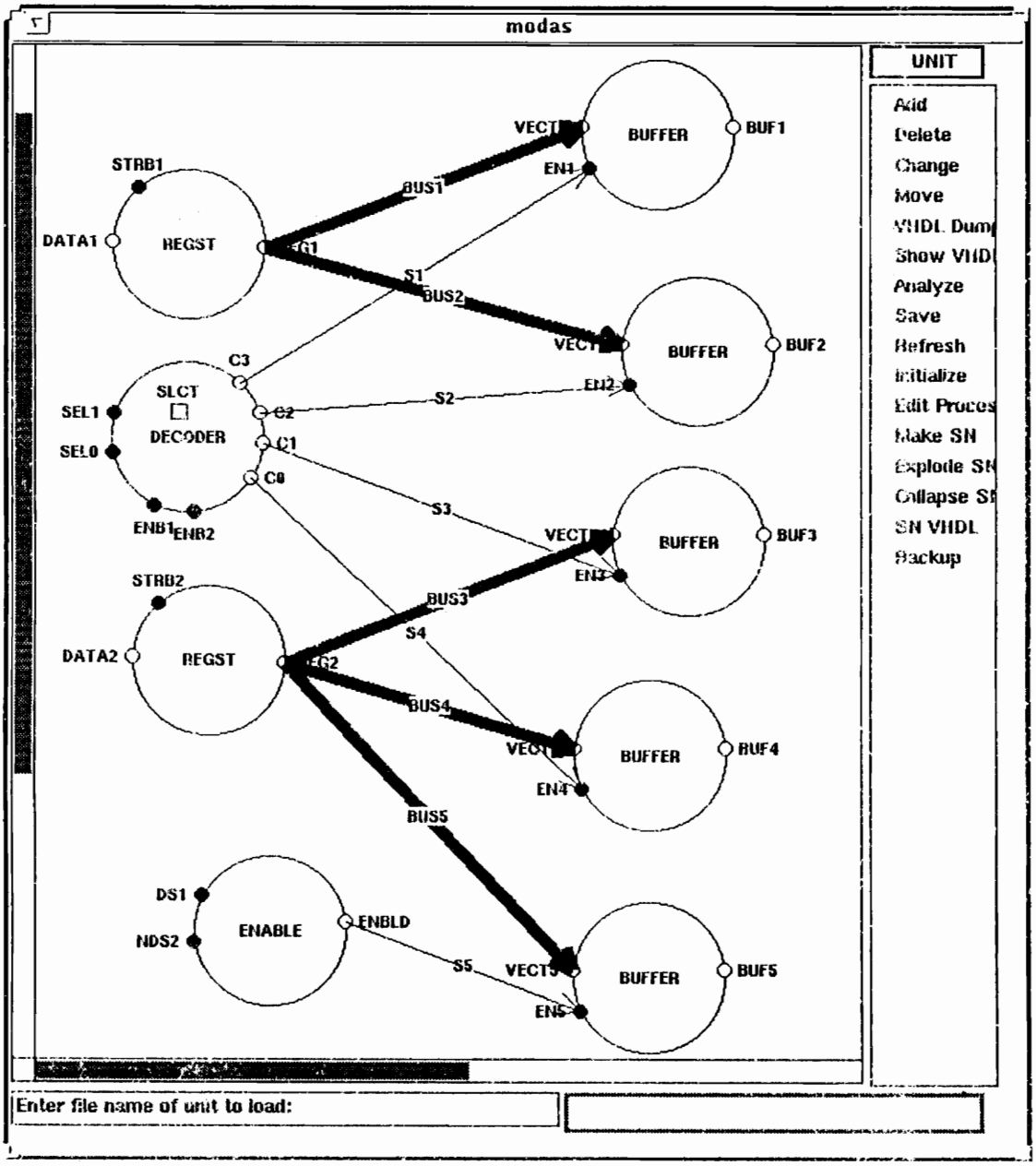


Figure-48: Model IOSYS (Part of an Input/Output System)

```
Enter name of the unit:
IOSYS

Unit name is IOSYS
No. of signals = 22

Sensitive path 0:
sp[0][0]=12(SEL0)
sp[0][1]=5(C3)
sp[0][2]=56(EN1)
sp[0][3]=54(BUF1)

Sensitive path 1:
sp[1][0]=13(SEL1)
sp[1][1]=9(C2)
sp[1][2]=50(EN2)
sp[1][3]=48(BUF2)

Sensitive path 2:
sp[2][0]=14(ENB1)
sp[2][1]=10(C1)
sp[2][2]=38(EN3)
sp[2][3]=36(BUF3)

Sensitive path 3:
sp[3][0]=15(ENB2)
sp[3][1]=11(C0)
sp[3][2]=32(EN4)
sp[3][3]=30(BUF4)

Sensitive path 4:
sp[4][0]=26(NDS2)
sp[4][1]=24(ENBLD)
sp[4][2]=20(EN5)
sp[4][3]=18(BUF5)

Sensitive path 5:
sp[5][0]=27(DS1)
sp[5][1]=24(ENBLD)
sp[5][2]=20(EN5)
sp[5][3]=18(BUF5)

Sensitive path 6:
sp[6][0]=44(STRB2)
sp[6][1]=42(REG2)
sp[6][2]=21(VECT5)
sp[6][3]=18(BUF5)

Sensitive path 7:
sp[7][0]=62(STRB1)
sp[7][1]=60(REG1)
sp[7][2]=51(VECT2)
sp[7][3]=48(BUF2)

No. of Spath = 8
```

Figure-49: Sensitive Path Construction for model IOSYS

Table-9: Test-Sequence for model IOSYS

frame	C3	C2	C1	C0	SEL0	SEL1	ENB1	ENB2	BUF5	ENBLD	NDS2	DS1	BUF4	BUF3	REG2	STRB2	DATA2	BUF2	BUF1	REG1	STRB1	DATA1
0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
2	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	D1	R	D1
1	R	F	0	0	R	1	1	1	X	X	X	X	X	X	X	X	X	X	D1	D1	1	D1
3	0	R	0	F	0	R	1	1	X	X	X	X	X	X	X	X	X	D1	D1	D1	1	D1
5	F	R	0	0	0	1	1	1	X	X	X	X	X	X	X	R	D1	D1	Z	D1	1	D1
4	0	0	R	F	1	0	R	1	X	X	X	X	X	D1	D1	1	D1	D1	D1	D1	1	D1
6	0	0	R	F	0	0	1	R	X	X	X	X	Z	D1	D1	1	D1	D1	D1	D1	1	D1
7	0	0	1	0	0	0	1	1	D1	R	F	1	Z	D1	D1	1	D1	D1	D1	D1	1	D1
8	0	0	1	0	0	0	1	1	Z	F	0	F	Z	D1	D1	1	D1	D1	D	D1	1	D1
9	0	0	F	R	0	0	1	1	D2	R	F	1	D2	D1	D2	R	D2	D1	D1	D1	1	D1
10	0	R	F	R	0	0	1	1	D2	R	0	0	D2	D1	D2	1	D2	D2	D1	D2	R	D2

data date: Sun Apr 11 01:15:41 1993
simulation time: 49
coverage data: IOSYS.cov
VHDL source: IOSYS.vhd

Line	Count	Text
1		
2		use WORK.VHDLCAD.all, WORK.USER_TYPES.all;
3		-- **********
4		entity IOSYS is
5		port (SEL0: in BIT;
6		SEL1: in BIT;
7		ENB1: in BIT;
8		ENB2: in BIT;
9		BUF5: out MVL_VECTOR(0 to 7);
10		NDS2: in BIT;
11		DS1: in BIT;
12		BUF4: out MVL_VECTOR(0 to 7);
13		BUF3: out MVL_VECTOR(0 to 7);
14		STRB2: in BIT;
15		DATA2: in BIT_VECTOR(0 to 7);
16		BUF2: out MVL_VECTOR(0 to 7);
17		BUF1: out MVL_VECTOR(0 to 7);
18		STRB1: in BIT;
19		DATA1: in BIT_VECTOR(0 to 7));
20		end IOSYS;
21		-- **********
22		
23		architecture BEHAVIORAL of IOSYS is
24		
25		signal S1: BIT;
26		signal S2: BIT;
27		signal S3: BIT;
28		signal S4: BIT;
29		signal S5: BIT;
30		signal BUS5: BIT_VECTOR(0 to 7);
31		signal BUS4: BIT_VECTOR(0 to 7);
32		signal BUS3: BIT_VECTOR(0 to 7);
33		signal BUS2: BIT_VECTOR(0 to 7);
34		signal BUS1: BIT_VECTOR(0 to 7);
35		begin
36		
37		-- ----------
38		-- Process Name: DECODER
39		-- ----------
40		

```
41              DECODER_4: process (SEL0,SEL1,ENB1,ENB2)
42                variable SLCT: BIT_VECTOR(1 downto 0);
43              begin
44
45      8         if ((ENB1 and ENB2) = '1') then
46      5           SLCT(0) := SEL0;
47      5           SLCT(1) := SEL1;
48      5           case SLCT is
49      2             when "00" => S4 <= '1';
50      2             when "01" => S3 <= '1';
51      4             when "10" => S2 <= '1';
52      2             when "11" => S1 <= '1';
53                  end case;
54      8         end if;
55
56
57              end process DECODER_4;
58
59
60              -- ---------------------------------------------------------
61              -- Process Name: BUFFER
62              -- ---------------------------------------------------------
63
64              BUFFER_15: process (S5)
65              begin
66      3         if S5 = '1' then
67      1               BUF5 <= BV_TO_MVL(BUS5);
68      2         else
69      2               BUF5 <= "ZZZZZZZZ";
70      3         end if;
71
72
73              end process BUFFER_15;
74
75
76              -- ---------------------------------------------------------
77              -- Process Name: ENABLE
78              -- ---------------------------------------------------------
79
80              ENABLE_20: process (NDS2,DS1)
81              begin
82      4         S5 <= DS1 and not NDS2;
83
84
85              end process ENABLE_20;
86
87
88              -- ---------------------------------------------------------
89              -- Process Name: BUFFER
90              -- ---------------------------------------------------------
91
92              BUFFER_25: process (S4)
93              begin
94      2         if S4 = '1' then
95      1               BUF4 <= BV_TO_MVL(BUS4);
```

```
 96    1          else
 97    1               BUF4 <= "ZZZZZZZZ";
 98    2          end if;
 99
100
101          end process BUFFER_25;
102
103
104          -- ---------------------------------------------------
105          -- Process Name: BUFFER
106          -- ---------------------------------------------------
107
108          BUFFER_30: process (S3)
109          begin
110    2          if S3 = '1' then
111    1               BUF3 <= BV_TO_MVL(BUS3);
112    1          else
113    1               BUF3 <= "ZZZZZZZZ";
114    2          end if;
115
116
117          end process BUFFER_30;
118
119
120          -- ---------------------------------------------------
121          -- Process Name: REGST
122          -- ---------------------------------------------------
123
124          REGST_35: process (STRB2)
125          begin
126    4          if (STRB2 = '1') then
127    2                    BUS3  <= DATA2;
128    4          end if;
129
130
131          end process REGST_35;
132
133
134          -- ---------------------------------------------------
135          -- Process Name: BUFFER
136          -- ---------------------------------------------------
137
138          BUFFER_40: process (S2)
139          begin
140    2          if S2 = '1' then
141    1               BUF2 <= BV_TO_MVL(BUS2);
142    1          else
143    1               BUF2 <= "ZZZZZZZZ";
144    2          end if;
145
146
147          end process BUFFER_40;
148
149
150          -- ---------------------------------------------------
```

```
151             -- Process Name: BUFFER
152             -- ------------------------------------------------------
153
154             BUFFER_45: process (S1)
155             begin
156     2           if S1 = '1' then
157     1                   BUF1 <= BV_TO_MVL(BUS1);
158     1           else
159     1                   BUF1 <= "ZZZZZZZZ";
160     2           end if;
161
162
163             end process BUFFER_45;
164
165
166             -- ------------------------------------------------------
167             -- Process Name: REGST
168             -- ------------------------------------------------------
169
170             REGST_50: process (STRB1)
171             begin
172     4           if (STRB1 = '1') then
173     2                       BUS1  <= DATA1;
174     4           end if;
175
176
177             end process REGST_50;
178
179
180         end BEHAVIORAL;
```

Figure-50: Coverage Results for model IOSYS

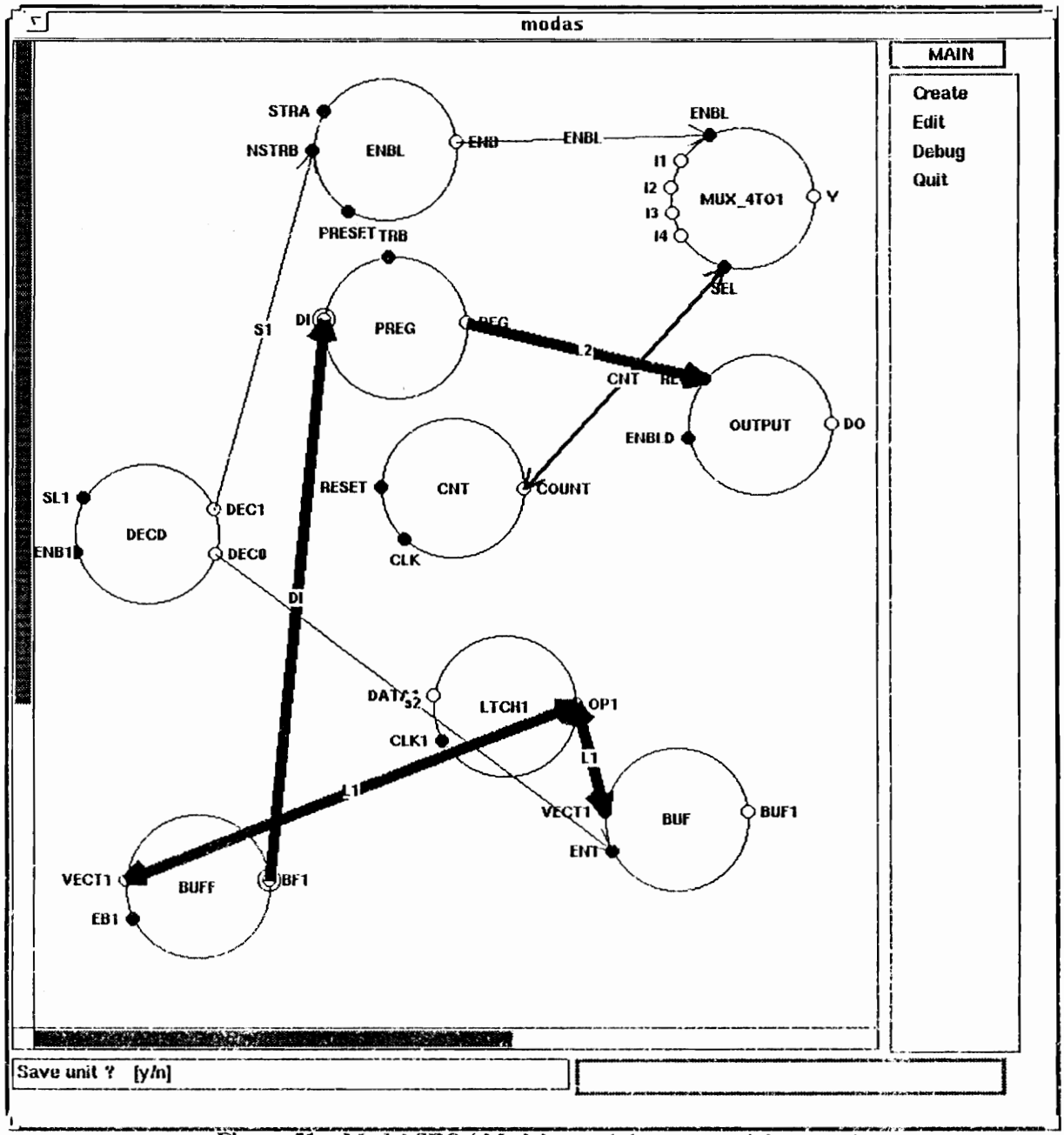


Figure-51: Model SEQ (Model containing sequential macros)

```
Enter name of the unit:
seq

Unit name is seq
No. of signals = 26

Sensitive path 0:
sp[0][0]=8(ENB1)
sp[0][1]=5(DEC1)
sp[0][2]=45(NSTRB)
sp[0][3]=42(ENB)
sp[0][4]=51(ENBL)
sp[0][5]=49(Y)

Sensitive path 1:
sp[1][0]=9(SL1)
sp[1][1]=7(DEC0)
sp[1][2]=14(EN1)
sp[1][3]=12(BUF1)

Sensitive path 2:
sp[2][0]=21(STRB)
sp[2][1]=18(REG)
sp[2][2]=33(REG)
sp[2][3]=30(DO)

Sensitive path 3:
sp[3][0]=26(CLK1)
sp[3][1]=24(OP1)
sp[3][2]=15(VECT1)
sp[3][3]=12(BUF1)

Sensitive path 4:
sp[4][0]=32(ENBLD)
sp[4][1]=30(DO)

Sensitive path 5:
sp[5][0]=38(EB1)
sp[5][1]=36(BF1)

Sensitive path 6:
sp[6][0]=44(PRESET)
sp[6][1]=42(ENB)
sp[6][2]=51(ENBL)
sp[6][3]=49(Y)

Sensitive path 7:
sp[7][0]=46(STRA)
sp[7][1]=42(ENB)
sp[7][2]=51(ENBL)
sp[7][3]=49(Y)

Sensitive path 8:
sp[8][0]=61(CLK)
sp[8][1]=59(COUNT)
sp[8][2]=52(SEL)
sp[8][3]=49(Y)

Sensitive path 9:
sp[9][0]=62(RESET)
sp[9][1]=59(COUNT)
sp[9][2]=52(SEL)
sp[9][3]=49(Y)

No. of Spath = 10
```

Figure-52: Sensitive Path Construction for model SEQ

Table-10: Test-Sequence for model SEQ

frame	SG0	SG1	SG2	SG3	SG4	SG5	SG6	SG7	SG8	SG9	SG10	SG11	SG12	SG13	SG14	SG15	SG16	SG17	SG18	SG19	SG20	SG21	SG22	SG23	SG24	SG25
0	X	X	X	X	X	X	D2	X	D2	F	D2	X	X	X	X	R	R	X	X	X	X	X	X	C1	X	R
1	X	X	X	X	X	X	D2	X	D2	0	D2	X	X	X	X	1	0	X	X	X	X	X	X	C1	R	0
2	X	X	X	X	X	X	D2	X	D2	0	D2	X	X	X	X	1	0	X	X	X	X	X	X	C2	F	0
3	X	X	X	X	X	X	D2	X	D2	0	D2	X	X	X	X	1	0	X	X	X	X	X	X	C2	R	0
4	X	X	X	X	X	X	D2	X	D2	0	D2	X	X	X	X	1	0	X	X	X	X	X	X	C3	F	0
5	X	X	X	X	X	X	D2	X	D2	0	D2	X	X	X	X	1	0	X	X	X	X	X	X	C3	R	0
6	X	X	X	X	X	X	D2	X	D2	0	D2	X	X	X	X	1	0	X	X	X	X	X	X	C4	F	0
8	X	X	X	X	X	X	D2	X	D2	0	D2	X	X	X	X	1	0	X	X	X	X	X	X	C2	F	0
7	R	Z	R	1	X	X	D2	X	D2	0	D2	X	X	X	X	F	0	1	F	1	1	1	1	C2	0	0
10	R	F	1	1	Z	X	D2	X	D1	F	D1	X	X	X	X	F	0	1	F	1	1	1	1	C2	0	0
9	F	R	1	F	D1	X	D2	X	D1	0	D1	X	X	X	X	0	0	1	0	1	1	1	1	C2	0	0
11	0	1	1	0	D1	D1	D2	R	D1	0	D1	D2	1	X	X	0	0	1	0	1	1	1	1	C2	0	0
12	0	1	1	0	D1	D1	D2	1	D1	R	D1	D2	1	X	X	0	0	1	0	1	1	1	1	C2	0	0
13	0	1	1	0	D1	D1	D2	1	D1	1	D1	D1	R	X	X	0	0	1	0	1	1	1	1	C2	0	0
14	0	1	1	0	D1	D1	D2	1	D1	1	D1	D1	1	D1	R	0	0	1	0	1	1	1	1	C2	0	0
16	F	R	1	0	D1	D1	D2	1	D1	1	D1	D1	1	D1	1	R	0	1	0	1	1	1	1	C1	F	0
15	0	1	1	0	D1	D1	D2	1	D1	1	D1	D1	1	D1	1	R	R	0	R	0	0	0	1	C1	0	0
17	0	1	1	0	D1	D1	D2	1	D1	1	D1	D1	1	D1	1	R	0	R	R	0	0	0	1	C1	0	0
19	F	R	1	0	D1	D1	D2	1	D1	1	D1	D1	1	D1	1	R	0	R	R	0	0	0	1	C1	0	0
18	0	1	1	0	D1	D1	D2	1	D1	1	D1	D1	1	D1	1	1	0	1	F	1	1	0	1	C2	F	0
20	0	1	1	0	D1	D1	D2	1	D1	1	D1	D1	1	D1	1	R	0	1	R	0	0	0	1	C1	0	R

Legend for table 10:

SG0 = DEC1
SG1 = DEC0
SG2 = ENB1
SG3 = SL1
SG4 = BUF1
SG5 = REG
SG6 = DI
SG7 = STRB
SG8 = OP1
SG9 = CLK1
SG10 = DATA1
SG11 = DO
SG12 = ENBLD
SG13 = BF1
SG14 = EB1
SG15 = ENB
SG16 = PRESET
SG17 = STRA
SG18 = Y
SG19 = I4
SG20 = I3
SG21 = I2
SG22 = I1
SG23 = COUNT
SG24 = CLK
SG25 = RESET

```
                Copyright (c) 1990-1992 by Synopsys, Inc.
                        ALL RIGHTS RESERVED
              This program is proprietary and confidential information
              of  Synopsys, Inc. and may be used and disclosed only as
              authorized in a  license agreement  controlling such use
              and disclosure.

data date: Sun Apr 11 17:17:29 1993
simulation time: 104
coverage data: seq.cov
VHDL source:   seq.vhd

 Line      Count              Text
------------------------------------------------------------
   1
   2              use WORK.VHDLCAD.all, WORK.USER_TYPES.all;
   3              -- ************************************************************
   4              entity SEQ is
   5                port (ENB1: in BIT;
   6                  SL1: in BIT;
   7                  BUF1: out MVL_VECTOR(0 to 7);
   8                  STRB: in BIT;
   9                  CLK1: in BIT;
  10                  DATA1: in BIT_VECTOR(0 to 7);
  11                  DO: out MVL_VECTOR(0 to 7);
  12                  ENBLD: in BIT;
  13                  EB1: in BIT;
  14                  PRESET: in BIT;
  15                  STRA: in BIT;
  16                  Y: out BIT;
  17                  I4: in BIT;
  18                  I3: in BIT;
  19                  I2: in BIT;
  20                  I1: in BIT;
  21                  CLK: in BIT;
  22                  RESET: in BIT;
  23                  DI: inout TSL_VECTOR(0 to 7)
  24                  );
  25              end SEQ;
  26              -- ************************************************************
  27
  28              architecture BEHAVIORAL of SEQ is
  29
  30                signal S1: BIT;
  31                signal s2: BIT;
  32                signal L1: BIT_VECTOR(0 to 7);
  33                signal L2: BIT_VECTOR(0 to 7);
  34                signal ENBL: BIT;
  35                signal CNT: BIT_VECTOR(0 to 1);
  36              begin
  37
  38                -- ---------------------------------------------------
  39                -- Process Name: DECD
  40                -- ---------------------------------------------------
```

```
41
42          DECD_23: process (ENB1,SL1)
43          begin
44    4        if ENB1 = '1' then
45    2           case (SL1) is
46    2                when '0' => s2 <= '1';
47    2                when '1' => S1 <= '1';
48             end case;
49    4        end if;
50
51
52          end process DECD_23;
53
54
55          -- ---------------------------------------------------
56          -- Process Name: BUF
57          -- ---------------------------------------------------
58
59          BUF_29: process (s2,L1)
60          begin
61    4        if s2 = '1' then
62    1              BUF1 <= BV_TO_MVL(L1);
63    3        else
64    3              BUF1 <= "ZZZZZZZZ";
65    4        end if;
66
67
68          end process BUF_29;
69
70
71          -- ---------------------------------------------------
72          -- Process Name: PREG
73          -- ---------------------------------------------------
74
75          PREG_4: process (STRB)
76          begin
77    2        if (STRB = '1') then
78    1              L2  <=  DI;
79    2        end if;
80
81          end process PREG_4;
82
83
84          -- ---------------------------------------------------
85          -- Process Name: LTCH1
86          -- ---------------------------------------------------
87
88          LTCH1_9: process (CLK1)
89          begin
90    6        if (CLK1 = '0') then
91    3              L1 <= DATA1;
92    3        else
93    3              L1 <= L1;
94    6        end if;
95
```

```
96
97              end process LTCH1_9;
98
99
100             -- ---------------------------------------------------
101             -- Process Name: OUTPUT
102             -- ---------------------------------------------------
103
104             OUTPUT_14: process (ENBLD,L2)
105             begin
106     4           if (ENBLD = '1') then
107     2                   DO <= BV_to_MVL(L2);
108     2           else
109     2                   DO <= "ZZZZZZZZ";
110     4           end if;
111
112
113             end process OUTPUT_14;
114
115
116             -- ---------------------------------------------------
117             -- Process Name: BUFF
118             -- ---------------------------------------------------
119
120             BUFF_19: process (EB1)
121             begin
122     2           if EB1 = '1' then
123     1                   DI <= L1;
124     2           end if;
125
126
127             end process BUFF_19;
128
129
130             -- ---------------------------------------------------
131             -- Process Name: ENBL
132             -- ---------------------------------------------------
133
134             ENBL_35: process (PRESET,S1,STRA)
135             begin
136    10           if (PRESET = '1') then
137     2                   ENBL <= '1';
138     8           else
139     8                   ENBL <= STRA and not S1;
140    10           end if;
141
142             end process ENBL_35;
143
144
145             -- ---------------------------------------------------
146             -- Process Name: MUX_4TO1
147             -- ---------------------------------------------------
148
149             MUX_4TO1_41: process (ENBL,CNT)
150             begin
```

```
151    14          if (ENBL = '1') then
152     7              case CNT is
153     4                      when "00" => Y <= I1;
154     4                      when "01" => Y <= I2;
155     4                      when "10" => Y <= I3;
156     2                      when "11" => Y <= I4;
157                    end case;
158    14          end if;
159
160
161            end process MUX_4TO1_41;
162
163
164            -- ------------------------------------------------------
165            -- Process Name: CNT
166            -- ------------------------------------------------------
167
168            CNT_50: process (CLK,RESET)
169            begin
170    16          if (RESET = '1') then
171     2                  CNT <= "00";
172    14          else
173    14             if (CLK = '0') then
174     8                  CNT <= INC(CNT);
175    14             end if;
176    16          end if;
177
178
179            end process CNT_50;
180
181
182          end BEHAVIORAL;
```

Figure-53: Coverage Results for model SEQ

Several other models were simulated and their coverage results have been summarized in table-10 on the next page. In this table, the terms MIN, MAX and AVG indicate the minimum, maximum and average number of times respectively that each statement in the model is executed during simulation.

Table-11: Coverage Results for Other Models

MODEL	MIN	MAX	AVG
fan	3	6	4.33
regstr	1	4	2
brfd	3	8	6.5
gt1	7	8	7.33
gt2	6	9	7.71
msd3	1	4	2
comprtr	2	8	4.64
gts1	3	8	6.6
gts2	2	10	5.42
addertest	8	10	9.5
fig694	4	8	4
muxckt	6	15	9.42
gt3	6	9	7.6
gt4	6	9	7.66
gt5	8	12	9.71
regs1	3	6	4
regs2	1	5	2.85
model1	1	4	2.5
admx1	5	18	12.2
bfr1	1	4	2.5
bfr2	1	4	2.35
sq1	2	16	6.08
sq2	2	16	5.72
msda	1	4	2.48
cntr1	2	16	7.83

Chapter 8. Proposed Future Development

A hierarchical approach to systematically generate tests for VHDL behavioral models has been discussed in this thesis. An system has been proposed to evaluate the quality of tests generated by the program. This chapter suggests methods by which HBTG and the system for test quality evaluation can be improved.

8.1 Implementation of a Process-Level Test Generator

In our approach, the primitive tests for each process of the PMG are currently being manually precomputed and stored in the design library. The characteristics of these primitive tests play a crucial role in the generation of the final test sequence. If these tests can be automatically precomputed by implementing a process-level test generator, the HBTG test generation process will be more efficient and systematic. One method of implementing such a program is to analyze the various VHDL constructs in each process of the process model graph. This information is available in the PMG functionality

database of the Modeler's Assistant. A second possibility is to modify the Behavioral Test Generator to develop tests for the processes.

8.2 Test Generation for Models with Reconvergent Fanout

The reconvergent fanout problem occurs more at the gate levels than at higher levels in the abstraction hierarchy. However, in order to make HBTG more versatile, it is necessary to consider reconvergent fanout in the test generation process. For models with reconvergent fanout structure, conflicts may occur in the justification process. When such inconsistencies occur in signal values, *backtracking* needs to be done in order to make a new decision. The HBTG algorithm uses a double linked-list to store the test sequences which could be used as the base on which backtracking is done.

8.3 Automatic Generation of Test-Bench

The test-bench file used to simulate the VHDL model is currently being manually constructed from the test sequence generated by HBTG based on the rules laid out in chapter 6. This is a time-consuming and error-prone process, and is extremely labor-intensive. The automatic construction of the test-bench file from the test vectors would be a major step in making the HBTG process fully automatic. Since symbolic values are used to represent signals, an automatic test-bench generator would need to convert these to appropriate values in the test-bench file. A test bench generator (TBG) has been

developed to construct test benches for the simulation of faulty VHDL models produced by the Behavioral Fault Mapper [32]. This test-bench generator could be modified to generate a test-bench based on the test vectors generated by HBTG.

8.4 Decision Coverage as the Test Generation Criterion

Currently, the criterion of statement coverage is being used to estimate quality of the HBTG test sequence. In this property, every statement in the VHDL model is executed at least once. As explained in chapter 6, a stronger criterion to test generation than statement coverage is ***decision coverage*** which requires every decision within the module must have a true or false decision, in addition to every statement being executed. By modifying HBTG to generate tests based on this criterion, it may be possible to obtain more effective tests.

8.5 Supernode Primitives

If some method of hierarchical representation above that of a process is available in the test generation process, similar to the existence of supernodes in the Modeler's Assistant, the test generation process could be enhanced. Test sequences generated by HBTG at one level of hierarchy could be modified to be used as the primitive tests for the next higher level.

8.6 Library of Primitive Tests

The presence of frequently used processes as primitives in the design library enhances the model development process in the Modeler's Assistant. This is also true to a large extent in the test generation process. A suite of primitive tests with appropriate characteristics could be developed and stored in the design library. These tests could be used by HBTG to construct test sequences for VHDL models. The presence of these tests will reduce the time spent in constructing the primitive tests each time they are needed.

Chapter 9. Conclusion

This thesis has presented an enhanced hierarchical test generation algorithm (HBTG) that can systematically generate tests for VHDL behavioral models. Given the Process Model Graph of the VHDL model and the primitive tests for each process of the PMG, the HBTG algorithm hierarchically constructs the test sequence for the whole entity. When used for simulation of the model, results show that the test sequence exercises the model thoroughly.

HBTG along with the Modeler's Assistant forms a CAD system for the rapid development and testing of VHDL behavioral models. The CAD system is shown in Figure-54. A design engineer or model developer using this system enters the process model graph of the VHDL behavioral model and the functionality of the processes using the Modeler's Assistant. The Modeler's Assistant automatically generates the VHDL behavioral model. Simultaneously, the HBTG algorithm constructs a test sequence for the model using the PMG database and the precomputed tests stored in the dsign library. The test sequence is then used for simulation of the model. Results obtained using the coverage measure during simulation indicate that the test sequence effectively exercises the model.

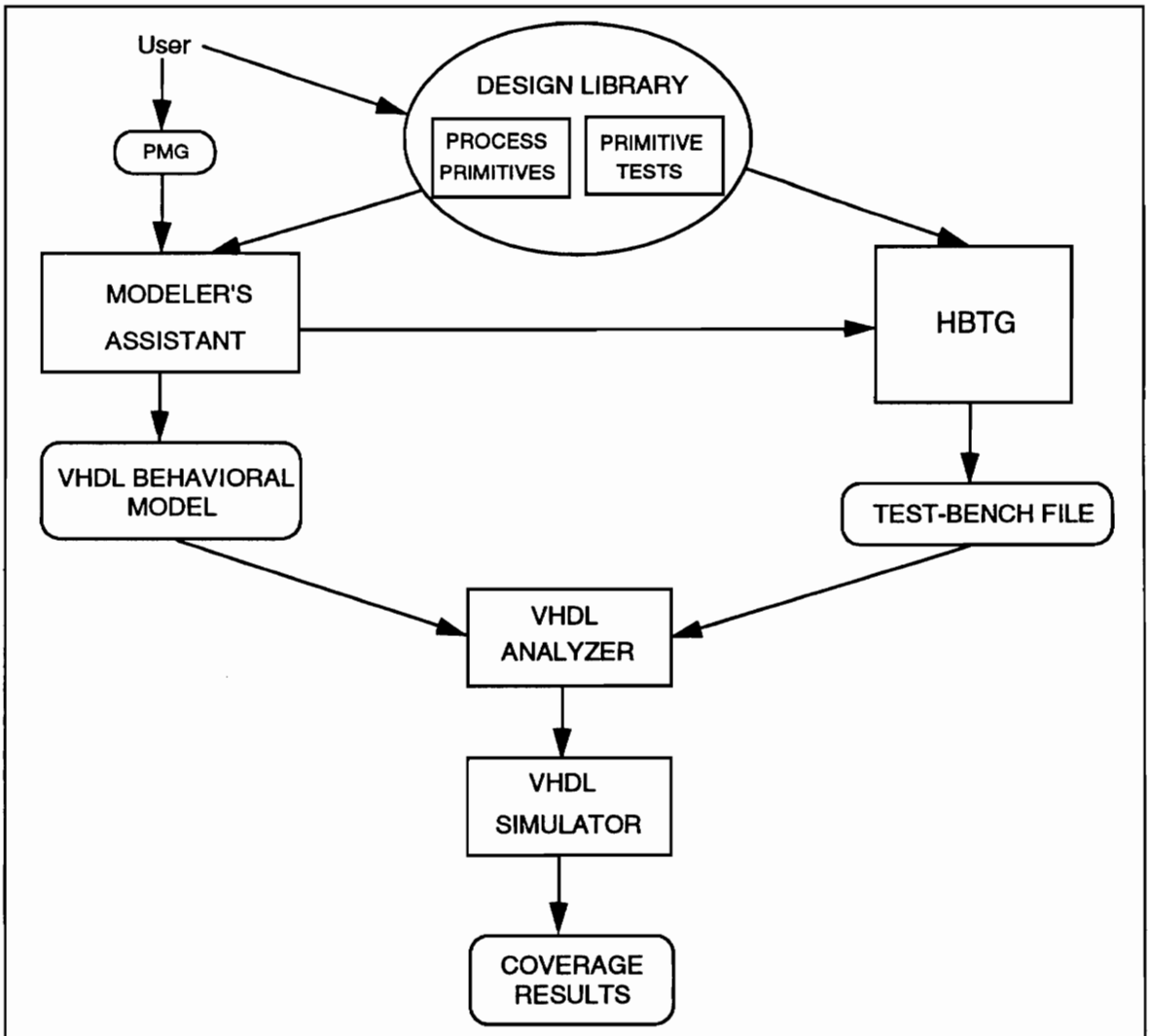


Figure-54: A Complete CAD System for VHDL Model Development and Testing

Bibliography

[1] R. Sartore, "The 80486: A Hardware Perspective," *Byte Magazine, IBM Special Edition*, pp. 67-74, Fall 1989.

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

[3] J. R. Armstrong, "Chip-Level Modeling with VHDL," Prentice Hall Inc., August 1988.

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

[5] 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.

[6] P. Goel, "An Implicit Enumeration Algorithm to generate Tests in Combinational Logic Circuits," *IEEE Transactions on Computers,* vol. c-30, pp. 215-222, March 1981.

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

[8] Ibarra, O. H. and Sahni, S., "Polynomially Complete Fault Detection Problems," *Trans. Computers,* vol. c-24, pp. 242-249, March 1975.

[9] D. Bhattacharya, B. T. Murray and J. P. Hayes, "High-Level Test Generation for VLSI," *IEEE Computer*, pp. 16-24, April 1989.

[10] 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.

[11] F. Somenzi, S. Gai, M. Mezzalama and P. Prinetto, "Testing Strategy and Technique for Macro-Based Circuits," *IEEE Transactions on Computers,* pp. 85-90, vol. c-34, no. 1, January 1985.

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

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

[14] 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.

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

[16] 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.

[17] 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*, pp.347-361, June 1987.

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

[19] B. Singh, "A Parametrized CAD tool for VHDL Model Development with X Windows," Master's Thesis, Virginia Polytechnic Institute and State University, 1990.

[20] P. A. Wright, "Rapid Development of VHDL Behavioral Models," Master's Thesis, Virginia Polytechnic Institute and State University, 1992.

[21] B. Pan, "Hierarchical Test Generation for VHDL Behavioral Models," Master's Thesis, Virginia Polytechnic Institute and State University, 1992.

[22] B. Singh, J. Wicks, P. Wright and J. R. Armstrong, "The Modeler's Assistant: A CAD Tool for Behavioral Model Development," *to be presented at CHDL'93.*

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

[24] Debugging Tools, Sun Source Browser Reference Manual, Sun Microsystems, 1991.

[25] H-K. T. Ma, S. Devadas, A. R. Newton and A Sangiovanni-Vincentelli, "Test Generation for Sequential Circuits," *IEEE Trasactions on Computer-Aided Design*, Vol. 7, No. 10, pp. 1081-1093, October 1988.

[26] R. Razdan, M. Anwaruddin, P. G. Kovijanic, R. Ganesh and H-C. Shih, " An Interactive Sequential Test Pattern Generation System," *International Test Conference*, pp. 38-46, 1989.

[27] W-T. Cheng and T. Chakraborty, "Gentest: An Automatic Test-Generation System for Sequential Circuits," *IEEE Computer*, pp. 43-49, April 1989.

[28] R. Marlett, "An Effective Test Generation System for Sequential Circuits," *23rd Design Automation Conference*, pp. 250-256, 1986.

[29] Myers, G. J., "The Art of Software Testing," John Wiley and Sons, Inc., 1979.

[30] B. Beizer, "Software Testing Techniques," Van Nostrand Reinhold, 1990.

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

[32] P. C. Ward, "Behavioral Fault Simulation in VHDL," Master's Thesis, Virginia Polytechnic Institute and State University, August 1989.

Appendix: Programming Guide for HBTG and its Environment

Introduction

This programming guide has been written to (1) Explain the various source code and header files used by HBTG (2) Illustrate the various data structures referenced by HBTG (3) Explain the compilation procedures to create the various executables.

The source code for the extraction of information from the PMG database, construction of sensitive paths and the test generation process is contained in six C files and four header files. The following is a brief description of the various file contents:

his.c

This is the file that stores information about the process model graph created on the Modeler's Assistant into the PMG database. It contains the function *Init_nodes* which initializes the pointers in the nodes and creates an elementary linked-list. The function *load_unit* stores information about the top PMG node: the unit. Functions *load_subunit* and *load_module* are then called to store information about individual processes.

extract.c

This file is used to extract information about the PMG from the PMG database making use of the functions defined in his.c. The file provides information about the order in which modules are stored in the linked-list of modules and the order in which ports are stored as a linked-list of ports. It also provides information about the modes and sensitivity of the various ports in the PMG.

spath.c

This is the file that contains the *main* function. This file contains functions *Assign_Signal_Order,* which assigns a port order to the various ports, and *Construct_Sensitive_Path,* which constructs sensitive paths through the PMG. Several functions are also present which are used during the sensitive path construction process.

stpath.c

This file also contains the *main* function. It contains all the information that is present in spath.c. In addition to performing the sensitive path construction, this file also calls the procedure to start the test generation process.

input.c

This file contains functions that read information from the primitive *.tst* files stored in the design library for each process and store them in the linked-list of structures: *struct test.* The main function is the *Input_tst* function which creates the linked-list test database of test-sets.

testgen.c

This file contains the procedures to perform the test generation process. This is the largest file and it contains numerous functions that perform selection of tests for activation, propagation and justification, functions to assign values during these activities, etc.

externsm.h

This file is included in all the six files described above. It contains various *extern* declarations for functions that are called from files other than that in which the functions are declared.

vhdlm.h

Various '#define' statements for constants are declared in this files. Globally-used data structures are also defined. Some variables which are used in different files are declared as *externs.*

macrosm.h

This file contains various macros which are used to reference the fields of the data structure *a_node.* The macros are used in the test generation process. These macros play a crucial role in manipulating the data structure fields.

The Data Structure of the Modeler's Assistant PMG Database

The Modeler's Assistant stores information about the process model graph as a linked-list of *nodes.* Each *node* is a data-structure of type *a-node* defined as:

```
typedef struct{
        char name[32];
        int type;
        int ptr[6];
        rect R;
} a_node;
```

```
typedef struct{
        int Xmin;
        int Ymin;
        int Xmax;
        int Ymax;
} rect;
```

Each node stores information about an *object.* Objects can be units, processes, ports, supernodes, etc. In the data structure *a_node,* the field **name** stores the name of the object, **type** is an integer code used to identify the type of object (signal, variable, etc.); **rect** stores information about the location of the object on the screen in terms of screen coordinates; the array elements **ptr[0-5]** are used as pointers to different elements in the PMG.

Table-12: List of Pointer References

object (type #)	ptr[0]	ptr[1]	ptr[2]	ptr[3]	ptr[4]	ptr[5]
Unit (12)	TopModRef	TopSignal				TopSuperNode Ref
Module(1)	TopPort	TopVariable	TopGeneric			TopConstant
Module ref (14)	TopOfModule	NextModRef				
Port (2)	NextPort	PortType	PortSignal	PortStatus		
Variable(4)	NextVariable	VariableType				VarAggString
Generic(3)	NextGeneric	GenericType				
Constant(32)	NextConstant	ConstantType				ConAggString
Signal (13)	NextSignal	SignalSrc	SignalDst			
SrcNode(15)	PortModRef	AbsPortLoc	RelPortLoc			
DstNode(16)	PortModRef	AbsPortLoc	RelPortLoc			
SuperNodeRef (38)	SuperNodeOf Ref	NextSuper NodeRef				TopPortMap
SuperNode (37)	TopModRef	TopSignal	TopSNPort			
SuperNodePort(39)	NextPort	PortType	PortSignal	PortStatus		
PortMap (40)	NextPortMap					

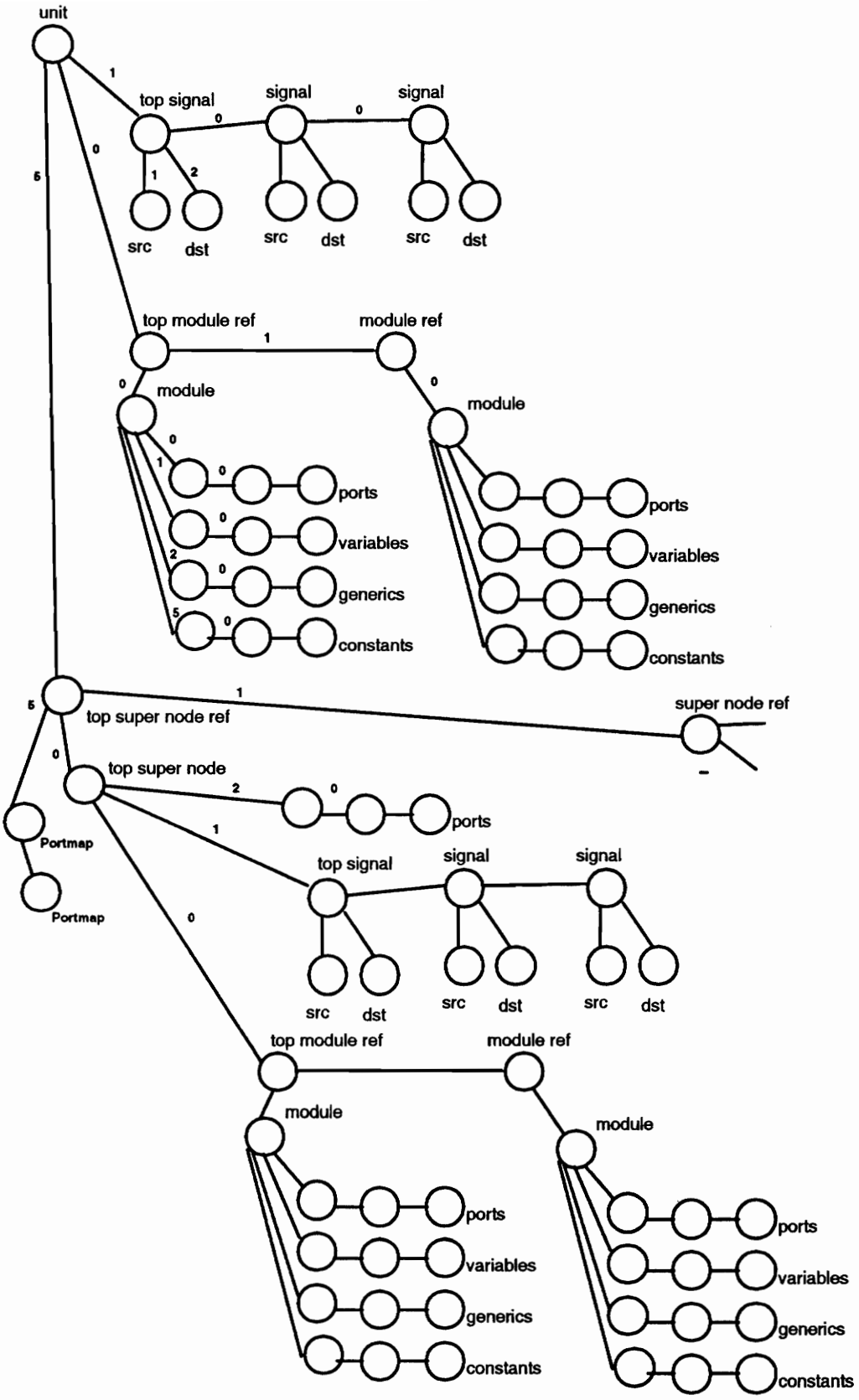


Figure-55: Generalized linked-list database

Figure-55 shows a general linked-list database. Here, each circle represents one element of the array *node* (each *node* is a data structure of type *a_node*). The lines indicate how the pointers ptr[0-5] are used to point to other nodes. For example, for a unit node, ptr[0] points to the top module in the linked list and ptr[1] points to the top signal in the linked-list. The table-11 shows the functions of the pointers for different types of nodes.

The data structure *a-node* as seen by HBTG is :

```
typedef struct{
        char name[32];
        int type;
        int ptr[8];
        rect R;
} a_node;
```

The number of pointers has been increased to 8 in order to incorporate more macros needed during the test generation process. Hence, when the information is read into the structure, the size of the information read is optimized in the *sizeof* field of the file read command *fread.*

Data Structure used to Store Primitive Tests

The primitive tests for each process of the PMG are stores in a linked list of data-structure. This primitive test database uses a data structure *struct test* defined as:

```
struct test{
        int tstorder;
        int nu_frame;
        char tst_buffer[MAX_NU_FRAME][MAX_LINE_CHAR];
        struct test *ptr_to_next;
}
```

In the above data structure, the filed **tstorder** indicates the order of a particular test-set; **nu_frame** indicates the number of time-frames in the test; the array **tst_buffer** is a string used to store the test-set; the pointer **ptr_to_next** points to the next structure in the linked-list of test-sets.

Data Structure used for the generated test sequence

The test sequence generated by HBTG is stored in a linked-list of time frames. Each element in the double linked list is one time frame. Each element in the database is a data structure of type *time_frame* defined as:

```
struct time_frame{
        int frameorder;
        struct time_frame *ptr_to_next;
        struct time_frame *ptr_to_last;
        int signal_value[100];
}frame;
```

In the above data structure, the field **frameorder** indicates the number of the current time frame; the pointer **ptr_to_next** points to the next structure in the double linked-list; the

pointer **ptr_to_last** points to the previous time frame; the array **signal_value** is used to store the entire test sequence for a particular time frame.

Format of files used by HBTG

The process model graph information created on the Modeler's Assistant is mainly stored in two kinds of files: the *.mod* file and the *.unt* file. A *.mod* file stores information about an individual process, whereas a *.unt* file stores information about the entire unit. When a *.unt* file is opened while extracting information for test generation, it invokes the *.mod* files for each process in the PMG. Both the *.mod* and the *.unt* files are binary files. Hence, the *fopen* command in C uses a "wb" option. Data for each node is read using the *fread* command and is written using the *fwrite* command.

Figure-56 shows the structure of a *.mod* file. This file contains information about the functionality of the process followed by information of the node structure used to store ports, generics, etc. The structure of a *.unt* file is shown in Figure-57. This file contains a series of node numbers and node data.

In addition to the above files, precomputed test files for each process in the process model graph are stored as *.tst* files in the same directory as the *.unt* and *.mod* files. i.e. for a process ENABLE stored in ENABLE.mod, the primitive test sets must be stored in ENABLE.tst. The *.tst* files are simple ASCII files.

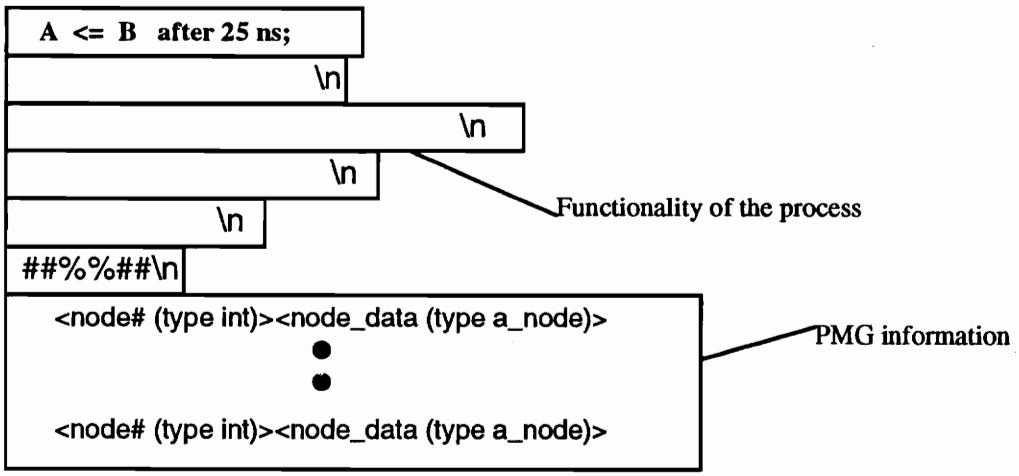


Figure 56: Structure of a .mod (process) File

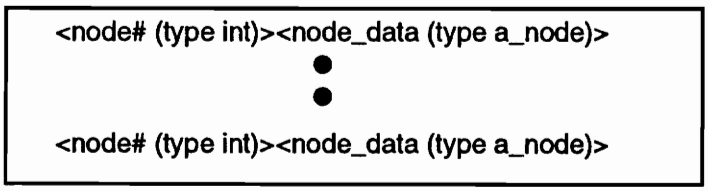


Figure 57: Structure of a .unt (unit) File

Compilation Procedures

There are three executable files in the HBTG environment:

extract : extracts information from the database of the Modeler's Assistant.

spath: constructs sensitive paths through the PMG.

HBTG : performs the test generation process.

In order to generate the above executables, the following compilation procedure is followed.

cc his.c extract.c -o extract

cc his.c spath.c -o spath

cc his.c stpath.c input.c testgen.c -o HBTG

Vita

Sanat R. Rao was born October 19, 1970 in Bangalore, India. He entered the University of Bombay, India, in July 1987, and received a Bachelor of Engineering degree in Electronics Engineering, graduating with *honors* in July 1991. He attended graduate school at the Virginia Polytechnic Institute and State University from August 1991 to May 1993, receiving a Master of Science degree in Electrical Engineering in May 1993.

Sanat has been employed with Intel Corporation in Phoenix, Arizona, since the July of 1993.