Integration of VHDL Simulation and Test Verification into a Process
Model Graph Design Environment

by

David M. Dailey Jr.

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. Cyrus

Dr. F. G. Gray

June 1994

Blacksburg, Virginia
Integration of VHDL Simulation and Test Verification into a Process Model Graph Design Environment

by

David M. Dailey Jr.

Dr. James R. Armstrong, Chairman

Electrical Engineering

(ABSTRACT)

This thesis discusses the ability to maintain a consistent design, simulation, and test verification environment by use of the Process Model Graph (PMG) throughout the development process. This ability extends the functionality of the PMG to include the visualization of simulation results and the verification of test paths within the simulation. These ideas have been implemented within a development tool called the Modeler's Assistant. The integration of the test generation environment into the tool is discussed. The design methodology used in creating the simulation environment is also discussed. Other enhancements to increase the abilities of the tool and improve its usefulness to behavioral test generation and verification are also discussed. Many examples of the new extentions to the tool are presented.
Acknowledgments

I wish to express my greatest thanks to my advisor Dr. James R. Armstrong for his constant support through the development of this thesis. His guidance and knowledge of the field have been invaluable. The opportunities provided by him have been a catalyst in the learning process. I would like to thank Dr. Walling R. Cyre and Dr. F. Gail Gray for serving as members of my committee.

It has been a pleasure working with all the members of my research group. I would especially like to thank Shekar Kapoor and Sanat Rao for their help with understanding the testing aspects of this work.

I most appreciate the funding support provided by the National Science Foundation.

I would like to dedicate this work to my parents and brother who have lovingly encouraged me and stood behind me throughout my life's endeavors. They have shown me the value of education and hard work and I am eternally in their debt. I wish to thank all my friends for the good times they provided and for always being there when needed. I especially wish to thank Jennifer Biller who has shown me the beauty of life and that with faith in God and in myself, there is no goal too high.
Table of Contents

Chapter 1. Introduction ........................................................................................................... 1
  1.1 Motivation ...................................................................................................................... 1
  1.2 The Modeler's Assistant ............................................................................................... 2
  1.3 Test Generation Tools .................................................................................................. 5
  1.4 Contents ....................................................................................................................... 6

Chapter 2. Background of the Modeler's Assistant ............................................................... 8
  2.1 Previous Work ............................................................................................................... 8
  2.2 Enhancement of the Modeler's Assistant ...................................................................... 10
  2.3 Programming Environment ......................................................................................... 12

Chapter 3. Test Bench Creation ............................................................................................ 16
  3.1 Value of the PMG in Test Bench Creation ................................................................... 16
  3.2 Automatic Test Bench Creation .................................................................................. 20
  3.3 User Defined Test Benches ......................................................................................... 22
Chapter 4. The Process Model Graph as a Test Verification Environment ..... 24
4.1 Existing Simulation Tools ................................................................. 24
4.2 Consistency with the Design Environment ..................................... 26
4.3 Correspondence to Test Generation Techniques ............................. 28
4.4 Relation between Simulation Results and Test Paths .................... 29
4.5 Visualization and Animation of Simulation in a PMG Form ............. 30

Chapter 5. Integration of Design, Test Generation, and Verification into a Single Environment ................................................................. 28
5.1 Previous Development Cycle Turnaround Time .............................. 38
5.2 Improvements to the Development Cycle Through Environment Integration .... 40

Chapter 6. The Simulation System Design Methodology .......................... 45
6.1 Contribution of the Existing PMG Data Structure to Simulation ........ 45
6.2 The Choice of Data Structure to Store Events .............................. 47
6.3 Mapping of the Simulation Data Structure to the Simulation Environment .... 52
6.4 The Simulation Environment Algorithm .................................. 54

Chapter 7. Other Enhancements ............................................................ 56
7.1 External Signals .......................................................................... 56
7.2 Enhancement of Other Features .................................................. 59

Chapter 8. Approach to Model Development ........................................ 62
8.1 The Design Process as a Whole ................................................................. 62
8.2 Example Development ............................................................................... 65

Chapter 9. Conclusions and Future Work..................................................... 113
9.1 Creating Synthesizable Code .................................................................. 113
9.2 Maintenance of Procedure and Type Library ......................................... 113
9.3 Test Generation for Supernodes .............................................................. 114
9.4 Combining the Test Generation Programs .............................................. 114
9.5 User Help System .................................................................................. 114
9.6 Conclusions ........................................................................................... 114

Bibliography .................................................................................................. 116

Appendix A: Guide for Simulation and Test Evaluation with the Modeler's
Assistant ........................................................................................................ 118

Appendix B: Programmer's Guide ................................................................. 133

Vita ................................................................................................................ 150
List of Illustrations

Figure 1. Example PMG and Generated VHDL Code ........................................... 3
Figure 2. Modeler’s Assistant Programming Environment ..................................... 13
Figure 3. 8-Bit Latch PMG and Generated Sensitive Paths .................................... 18
Figure 4. The Test Generation Environment ....................................................... 21
Figure 5. Textual Output of Synopsys Simulator .................................................... 25
Figure 6. Synopsys Waveform Viewer Output ...................................................... 26
Figure 7. 8-Bit Latch PMG with Monitors Selected ............................................... 32
Figure 8. PMG in the Paths View of the Simulation Environment .......................... 34
Figure 9. Coverage Results Window for the 8-Bit Latch ...................................... 35
Figure 10. Signal History for the ENABLE Signal of the 8-Bit Latch ......................... 37
Figure 11. The Process Test Generator Interface ................................................. 42
Figure 12. The Test Bench Generator Interface .................................................... 43
Figure 13. The Custom Test Bench Entry Window ............................................... 44
Figure 14. Flowchart for Simulation of Events .................................................... 48
Figure 15. Overview of the Simulation Environment Data Structure ....................... 51
Figure 16. 8 Bit Latch PMG with an External Signal ............................................ 57
Figure 17. Supernodes with External Signals ....................................................... 59
Figure 18. VHDL Development Environment ...................................................... 63
Figure 19. Model IOSYS (Input/Output Subsystem) [2] ....................................... 66
Figure 20. Test Bench for IOSYS ........................................................................ 73
Figure 21. IOSYS Test Paths .............................................................................. 74
Figure 22. IOSYS Test Paths after 49 NS of Simulation ......................................... 75
Figure 23. IOSYS after 0 NS of Simulation ......................................................... 76
Figure 49. TWOSTMUX after 31 NS of Simulation ........................................... 102
Figure 50. TWOSTMUX after 79 NS of Simulation ........................................... 103
Figure 51. History of the IN0 Signal ................................................................. 104
Figure 52. Coverage Window for TWOSTMUX Model .................................... 105
Figure 53. Waveform Window for TWOSTMUX Model ................................. 106
Figure 54. Model Testing and Simulation Using the Modeler’s Assistant .......... 119
Figure 55. The Simulation Environment .......................................................... 123
Figure 56. Display of Test Paths ................................................................. 125
Figure 57. Port and Signal History Window .................................................. 128
Figure 58. Coverage Results ........................................................................... 129
Figure 59. Overview of the Test Generation Interface .................................... 135
Figure 60. Structure of the PMG Linked List Database ................................. 140
Figure 61. Overview of the Simulation Environment Data Structure ............ 143
Figure 62. C Structures Used in the Simulation Data Structure .................... 144
Figure 63. Example Simulation Control File ............................................... 147
Figure 64. Sim.wav File for Latch8 Model ....................................................... 148
List of Tables

Table 1. Environment Variables Used by the Modeler's Assistant ..................... 137
Table 2. Intermediate Files Created and when They Are Generated ............... 138
Table 3. List of Pointer References ................................................................. 139
Chapter 1. Introduction

1.1 Motivation

Traditionally, design of digital devices and systems has been done through the graphical layout of components and routing. It is obvious that this type of design does not give a clear indication of the behavior of complex designs. As another drawback, the testing of such designs is often difficult due to the many levels of the design hierarchy. Testing of such systems is usually done by first creating large test sets and then using expensive methods to exercise the system with those tests. When functional faults are found in complex systems, they are very difficult to trace backward through the design. As a result, complex traditionally designed systems suffer from a long turnaround time and therefore larger costs.

The use of a hardware description language known as the VHSIC Hardware Description Language, or VHDL, helps to simplify this process by providing the user
with a behavioral testing environment. Textual models can be created which describe
the function and behavior of a system, the models simulated, and the results observed.
However, the methods of creating, testing, and verifying these models is a very time
consuming process. This work aims to shorten this turnaround time.

1.2 The Modeler's Assistant

The use of the Process Model Graph [13], or PMG, in the design of VHDL
models helps to greatly decrease the model development time (Figure 1) [1]. In this
approach, a graph is constructed to represent the VHDL process interaction that
makes up a VHDL entity. This VHDL entity is also called a unit. In the unit PMG,
the large circles of the graph represent VHDL processes, and the smaller circles on the
perimeter of the large circles represent process inputs. These are called process ports.
Shaded ports represent the signals in the sensitivity list for the process. This means
that a change in the signal value for this port will cause the execution of the process.
The interconnecting lines represent VHDL signals which are shared among the
processes. Using the PMG, the user creates a graph of VHDL processes that can be
used to quickly create the VHDL description file based on the graph structure. This
graphical view of the model brings about a natural style of modeling and allows the
user to visualize process interaction.
use WORK.VHDLCAD.all;
WORK.USER_TYPES.all;
-- **************************************************************
entity PMG is
  port (out2: out BIT;
         out1: out BIT;
         input4: in BIT;
         input3: in BIT;
         input2: in BIT;
         input1: in BIT);
end PMG;
-- **************************************************************
architecture BEHAVIORAL of PMG is
  signal SIGNAL3: BIT_VECTOR(7 downto 0);
signal SIGNAL2: BIT;
signal SIGNAL1: BIT;
begin

-- Process Name: PROCESS3
-- ---------------------------------------------------------------
PROCESS3_9: process
begin
  User Specifies Code Here
end process PROCESS3_9;

-- Process Name: PROCESS2
-- ---------------------------------------------------------------
PROCESS2_13: process
begin
  User Specifies Code Here
end process PROCESS2_13;

-- Process Name: PROCESS1
-- ---------------------------------------------------------------
PROCESS1_18: process
begin
  User Specifies Code Here
end process PROCESS1_18;
end BEHAVIORAL;

Figure 1. Example PMG and Generated VHDL Code
This idea of using the PMG as a design tool is implemented in a tool known as the Modeler's Assistant. The tool can be used to create the PMG and then dump a VHDL file derived from the graph and the process specific VHDL code. This tool's features have been extended to include the use of parameterized process primitives to allow for quicker creation of PMGs from reusable processes, and to allow the use of supernodes, i.e., a graph node can now represent more than one process and can therefore be used to create structural VHDL models.

The sole purpose of the Modeler's Assistant is to help speed the development of VHDL models. The use of the graphical tool for creation of the model helps to do this by eliminating the need to manually write the declarative and interfacing parts of a VHDL model. It also allows the user to reuse processes in a parameterized way so to eliminate repetitive coding. The use of the PMG in modeling eases the understanding of the model and reduces development time losses due to complexity. It is obvious, however, that a significant portion of the development of a VHDL model can still be improved. In order to complete a VHDL design, a developer must also create tests for the model, simulate the model, and then compare the simulation results to the design specifications. The PMG can be used naturally to help achieve these tasks. It can be used by test generation tools, which already exist, and as a simulation environment for the verification of these tests. The enhancement of the Modeler's Assistant to help achieve these goals in a rapid manner is the focus of this work.
1.3 Test Generation Tools

The work of Pan [11], Rao [2], and Kapoor [3] describe the creation and use of test generation tools based on the Modeler's Assistant PMG design. These tools can be used to create a test bench for a model. There are three programs required in the creation of a test bench, the Process Test Generator [3], the Heirarchical Behavioral Test Generator [2, 11], and the Test Bench Generator [3].

The first tool needed, the Process Test Generator, or PTG, uses the specified VHDL code and the generated control flow graph for a process and creates tests which can be used to exercise all of the statements of that process. The only values that need be specified by the user are the data values used by the model. The control signals are automatically generated. The second tool, the Heirarchical Behavioral Test Generator, can use these process tests along with the PMG data file to generate tests for the entire PMG model. The third tool, the Test Bench Generator, takes the model information and the created tests for the entire PMG and creates a VHDL test bench file which implements these tests.

Due to the large amount of parameters and interaction these tools require, this work describes ways in which the Modeler's Assistant has been enhanced to interface to these tools, to ease the use of the tools, and to allow for rapid creation of a test bench. This enhancement also allows the user to easily create their own custom test bench without using the test generation tools.
1.4 Contents

Chapter 2, "Background of the Modeler's Assistant," discusses the history of the Modeler's Assistant VHDL design tool, its limitations, the enhancements to the tool, and the programming environment.

Chapter 3, "Test Bench Creation," briefly explains how the Process Model Graph can be used in the test bench generation process, how to use the existing test bench generation tools, and how the PMG is used in allowing for easy creation of a customized test bench.

Chapter 4, "The Process Model Graph as a Test Verification Environment," explains the benefits of extending the usage of the PMG into a simulation verification model and how this is done with the Modeler's Assistant.

Chapter 5, "Integration of Design, Test Generation, and Verification into a Single Environment," discusses the benefits of the integration of design, test generation, and model verification into a single tool and how this is accomplished within the Modeler's Assistant.

Chapter 6, "The Simulation System Design Methodology," discusses how some data from the existing PMG data structures can be used for simulation purposes, the choice of data structures used to store events in the simulation, how this data structure is mapped onto its graphical environment, and the algorithm used in moving through the simulation.
Chapter 7, "Other Enhancements," describes the other additions and enhancements that have been made to existing features of the Modeler's Assistant.

Chapter 8, "Approach to Model Development," explains the design process in using the Modeler's Assistant for quick development and verification of VHDL models.

Chapter 9, "Conclusions and Future Work", discusses possible future improvements to the Modeler's Assistant.


Appendix B, "Programming Manual," describes the C code used to program the features of the Modeler's Assistant.
Chapter 2. Background of the Modeler's Assistant

2.1 Previous Work

The previous versions of the Modeler's Assistant [1, 4, 5] provided a fairly complete tool for VHDL model creation using the Process Model Graph. The tool was first developed to provide for creation of a PMG to allow for a basic dump of the VHDL model based on the PMG. The user could rapidly develop a model by creating processes and interconnecting them to form a PMG model. The tool was then ported into a X-Windows environment. It was then enhanced further so the user could store a set of parameterized process primitives to allow for the reuse of commonly used processes in a flexible manner. It was also enhanced to allow the user to create "supernodes" which represent a "sub" PMG within another PMG. This allowed the creation of hierarchical models that could be partitioned, therefore allowing structural models to be created from the PMG.
Although the earlier versions of the Modeler's Assistant provided a rapid way of designing VHDL models, they were limited in their use for the entire development process. The limitations of the earlier versions in the overall development process are enumerated below:

1. A significant time was required to generate tests for the model. Even with the addition of the automatic test bench generation tools, much time was spent going from the PMG to the process test generation tool, from there to the hierarchical behavioral test generator, and from there to the test bench generator. An easy interface to the Modeler's Assistant was needed that could simplify the use of the tools and hide the unnecessary handling of intermediate files.

2. Even hand-writing a test bench file required the user to create the entire test bench file, including the test bench "shell" (declarative parts). This part of the test bench could be automatically generated within the Modeler's Assistant

3. Unnecessary time was spent in the setup for simulation. The user was required to appropriately label design entities and then create a simulation control file to monitor the simulation.

4. Before this work, the verification process often involved calling the simulator and verifying the simulation by looking at either text based simulation results or waveforms that do not adhere to the design environment structure. It would clearly be easier to verify simulation results in the same PMG form that was used in the design phase.
5. No easy method of validating the automatic test generators existed. Since the tests were being generated based on the Modeler's Assistant PMG, a natural verification environment could exist using the PMG to visualize sensitive test paths through the model.

6. No simulator existed which placed the simulation into the perspective of the design. In order to check a design, a user was required to use a textual or waveform simulation output, which gives no relation to the correlation of processes and signals as in the PMG form.

7. In addition to these listed problems, the earlier version contained some bugs and other problems not previously detected. For example, no method existed for allowing an interconnection signal between PMG processes to be declared in the entity section of the behavioral model. A way of marking these signals as "external" was needed. This also applies to internal signals inside of supernodes.

8. Although the previous version made significant improvements in program documentation support, some areas were in need of improvement.

2.2 Enhancement of the Modeler's Assistant

The new version of the Modeler's Assistant (Version 4) adds many new features and enhances some existing ones. The focus of this work has been on the correction of the previously listed limitations of earlier versions. The goals given earlier in Chapter 1
have been implemented in this new version. The major changes to the Modeler's Assistant concern those limitations listed above concerning interfacing to the test generation features and the creation and integration of a simulation environment into the Modeler's Assistant.

In order to reduce the overall design time required in generating a test bench and using the test generation tools, a complete integrated menu-driven test generation interface to these tools has been added to the Modeler's Assistant. This includes interfaces to the process level test generator (PTG), the test generator for an entire unit (Heirarchical Behavioral Test Generator, or HBTG), and the Test Bench Generator (TBG). Without this interface, the process of using VHDL as a development tool is greatly slowed. Instead of having one tool to control the entire design process, the developer is forced to use and manage multiple tools. The use of this interface also provides the user with a test verification medium, i.e., tests are generated based on the PMG model structure, and therefore the basis of these tests can be validated naturally against this structure. This interface is fully discussed in Chapter 3, parts of Chapter 4, and Chapter 5.

To provide a user with a complete development tool, a simulation environment has also been added to the tool. Using this environment, the user can visualize the simulation in the same form as the design, moving forward and backward through the simulation to see the "flow" of signal events in the simulation. The control signals and interdependencies of the processes are seen in the PMG along with their current values. The time at which a signal becomes tested can be viewed and its effect on the rest of the model noted. The simulation environment can be switched into a "paths" mode in which the sensitive test paths through the model are overlaid onto the PMG. The simulation environment also provides the user with other results which can be helpful in the
verification of a design. These include such results as a signal history, coverage results, and the waveform results. A flexible data structure that is easy to traverse and manipulate was used in constructing the basis of the simulation environment. The simulation environment is discussed fully in Chapters 4, 5, and 6.

In addition to the major enhancements already listed, another feature was added to the original version to allow for signals internal to the PMG to be accessible outside of the VHDL entity or supernode. This is called the "External Signal" feature. Other "fine tuning" of the existing system was done. These features are discussed in detail in Chapter 7. A guide to using the newer parts of the Modeler's Assistant was also created and is given in Appendix A. To aid in future development, a programmer's guide detailing the C programming of the features was also created, and it is given in Appendix B.

2.3 Programming Environment

The previous version of the Modeler's Assistant was fully developed on a Sun Workstation in the X-Windows environment. This new version (version 4) has been partially developed on a Sun Sparc II™ workstation, but the majority of the code development was done on a Gateway 2000™ P5-60 PC compatible computer running the Linux operating system and the X-Windows environment. The C code for the Modeler's Assistant that runs under the Linux operating system is identical to that which runs on the workstations, and uses the same C libraries, so it was easily ported back and forth between the systems as the new version was developed. Figure 2 illustrates the programming environment used to develop the new version.
Figure 2 shows that this project used many different tools for program development. In the early stages of the work, the Sun Source Browser [6] was the primary development tool used because of its ease of use and ability to search through code objects quickly. This tool provided great help in learning the large amount of code in the early stages of work. With the current version consisting of 14,000 lines of source code, any future development of the program will also benefit from the use of this tool.

The debugging tools used in development were also greatly beneficial. Both the Sun Dbxtool and Linux xxgdb debuggers were used heavily in fixing old code and verifying new. The ability of both tools to read core dumps from the program errors and locate the erroneous line was invaluable. Both tools allowed the user to specify breakpoints in the code and then step through the code, one line at a time, from that point...
while monitoring the data structure. Without this ability to look at the complex data structure of the simulation environment, testing and debugging the program would have been a very tedious effort.

Much of the work to interface with the test generation tools required that the program call the execution of the other test generation programs. This required the use of many operating system calls within the code. This often resulted in the operating system itself becoming a part of the programming environment, since testing the code often required checking active system processes, verifying file output, and understanding input and output configurations (such as pipes). The input and output characteristics of the test generation tools, i.e., their necessary parameters, were also an important consideration in the interface creation.

As in the previous two versions, the graphics of the Modeler's Assistant uses the Athena Widget C libraries, X libraries, and Xt Intrinsic libraries. The Source Browser and X-Windows manuals eased the understanding and use of these libraries [7, 8, 9, 10]. Programming in the X-Windows environment requires the use of object-oriented structures called widgets. These widgets are mostly predefined but some of the drawing routines created in the earlier versions use the Xlib library of lower level functions to perform the graphics rather than the higher level widgets. Using widgets to program the windows allows the various menus and graphics to be created quickly. All of the X-Windows graphics used in the new version used similar widget configurations to those of the earlier version, since most new windows performed the same basic tasks. Some Xlib routines were used, mostly for piping output from the Synopsys and Vtcp Analyzers and the Synopsys Simulator to an X-Windows. The graphical features of the simulation environment required the modification of the drawing routines for the PMG. The regular Athena widget set was used for the Sun version. The Linux version used a slightly newer
version of the widget set called the Athena 3-D Widget Set. This set had some slight enhancements in looks, like a 3-D scrollbar. Reference [5] discusses the main graphical structure of the tool.
Chapter 3. Test Bench Creation

3.1 Value of the PMG in Test Bench Generation

Manual test generation for VHDL models is a very time consuming and tedious task. In order to manually generate tests for a model, the designer must be fully aware of the "structure" of the VHDL model. For example, the designer must know which signals in the model cause the execution of the various parts of the code, the types of events necessary to cause this execution, which signals are data values for this execution, and must in turn know how to order the signal events to properly execute the code. The goal of a designer when testing a model is usually to determine if the functionality of the model is correct. One requirement for the verification of the functionality is that the tests for the model completely exercise all written code. All statements of the code should be "covered". Full coverage of a VHDL model is a requirement for all test sets which are used in the verification of a design.
As stated earlier, the Modeler's Assistant uses the Process Model Graph to give a VHDL model a low level "structure" that can be used as a representational structure for VHDL models. The PMG "structure" can therefore be used as the earlier mentioned structure required in generating tests to fully cover the model. A tool has been developed to use this capability of the PMG structure to develop tests to cover a VHDL model. This tool is called the Heirarchical Behavioral Test Generator, or HBTG [2, 11].

The HBTG uses the relation between processes in the PMG and process level tests stored in a primitive test library to generate tests for a model. The primitive test library is a library of process level tests which, when exercised, will fully execute a particular process of the design. Another tool exists that generates the tests for single processes in the PMG and stores them in a file that can be used by HBTG in generating tests for the model. This is the Process Test Generator, or PTG [3]. The use of this tool will be discussed later in this chapter.

In order to generate tests based on the PMG and the primitive test library for a model, the HBTG must first generate sensitive paths through the PMG. A Sensitive Path is a directed path that starts at a sensitive primary input port and ends at a primary output port with the intermediate ports along the path consisting of as many sensitive ports as possible [2]. Figure 3 shows an example 8-bit latch model and its generated sensitive paths.
Sensitive path 0:
sp[0][0]=13(NDS2)
sp[0][1]=11(ENBLD)
sp[0][2]=7(ENBLD)
sp[0][3]=5(DO)

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

sp[1][3]=5(DO)

Sensitive path 2:
sp[2][0]=20(CLK)
sp[2][1]=17(OP)
sp[2][2]=8(REG)
sp[2][3]=5(DO)

No. of Spath = 3

Figure 3. 8-Bit Latch PMG and Generated Sensitive Paths
Once the HBTG program creates the sensitive paths for the model, the HBTG algorithm uses the activation - propagation - justification test sequence in generating tests. This algorithm is as follows [2]:

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.

This algorithm generates a complete test sequence for the model. It is clear from the algorithm that not only are the tests generated based on the "structure", or process interaction, of the PMG, but also on the movement, or propagation, of signal events through it. This observation is the basis for much of this work:

Observation: Since the Process Model Graph is the base of information used in test generation, the Process Model Graph provides a foundation and a
channel in which the generated behavioral tests for a model can be exercised, propagated, and observed.

The work for this thesis will show that, based on this observation, we can integrate the separate design environment, test generation environment, and simulation environment into a single environment based around the PMG - the environment within the Modeler's Assistant. The design environment discussed in the previous chapter already exists within the Modeler's Assistant. The remainder of this chapter will discuss the existing test generation environment. The integration of the environments into the Modeler's Assistant will be discussed in Chapter 5.

3.2 Automatic Test Bench Creation

The existing test generation tools provide the environment needed for generating a test bench that completely exercises the VHDL model. However this environment as it exists involves complex relations between programs and intermediate files. Figure 4 shows the test generation environment. This environment requires use of the Process Test Generator, or PTG, to generate the process level tests. This tool uses constructed control flow graphs and repeated simulation of tests to develop tests to exercise the process. The Heirarchical Behavioral Test Generator, or HBTG, then uses these tests and the PMG data to create tests for the entire model. These tests are based on paths from the sensitive inputs of the model to the primary outputs. Another program called the Test Bench Generator, or TBG, takes the HBTG results and converts them into a VHDL test bench. Reference [3] gives details on how these two programs work.
As shown in the figure, the starting point for automatic test bench creation is the Modeler's Assistant and the generated VHDL model. To start the process, the user must first go to the Modeler's Assistant and do a "VHDL Dump" command to generate the VHDL model. In order for the Process Test Generator to work, the user must then use the "mvhd" program included with the PTG to create a VHDL entity containing just the one process for which tests are being generated. The Modeler's Assistant then prompts the user for the name of the unit and the process name. Next this process must be analyzed by both the Synopsys and Vtip Analyzers to generate the control flow graphs for the process and to allow the process to be simulated. Then the user calls PTG. The PTG program requires six prompts for the user unrelated to the test generation. The file name,
unit name, process name, entity name, architecture name, and design library name must all be manually entered by the user. After the user enters the necessary data values for the test generation, the ".tst" file for that process is generated. This procedure, starting with the mvhd program, is then repeated to create the process tests for each process in the PMG. This is obviously a very time consuming procedure.

After the PTG creates all process level tests, the user must then call the HBTG program to generate the tests. The command must be invoked so that the output of the program gets directed into a ".test" file for the unit. When the program executes, the Modeler's Assistant prompts the user for the name of the unit file for which to generate tests.

The final step in generating the test bench is to call the TBG. The Modeler's Assistant then prompts the user for the unit VHDL file, the architecture name to be generated, the entity name to be generated, and the test file name from HBTG. It then asks for initial values needed for data signals. As this program executes, it generates a test bench file with the name "unit_TB.VHD" to exercise the model based on the HBTG results. After many steps and parameter passing, the user completes automatic test bench generation.

3.3 User Defined Test Benches

An additional use for the Process Model Graph that must be noted here is the ability to derive much of the VHDL code needed for any test bench file. The declarative regions for the VHDL entity, architecture, and component instantiations can be derived from information in the PMG database. Therefore, the PMG can be used to create a "shell" for a user to enter a custom test bench file, rather than an automatically generated one. This work not only includes an interface to automatic test bench generation tools,
but to a custom test bench entry window as well. This makes the test generation portion of the integrated environment more flexible and complete. Chapter 5 on the integration of the environments into the Modeler's Assistant details the creation of a custom test bench file.
Chapter 4. The Process Model Graph as a Test Verification Environment

4.1 Existing Simulation Tools

Current methods for simulating a VHDL model and verifying its correctness involve the use of commercial VHDL analyzers and simulators. The simulator currently used is the Synopsys Simulator [12]. This simulator provides the user with two forms of simulation output, textual and waveform.

The first output format is a textual output. Figure 5 shows this form of output. To create this output, the user must manually enter the signal names as monitors or create a simulation control file with the monitors listed. After the user creates this control file, the user must then call the simulator using the control file to create the text output file. Then the user must view the correct output text file with some sort of viewer. This process obviously requires much parameter passing and file handling. This process is also very time consuming.
<table>
<thead>
<tr>
<th>Time (NS)</th>
<th>Event Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>WRITE /LATCH8_TEST_BENCH/R1/LATCH_14/VAR (value = X&quot;00&quot;)</td>
</tr>
<tr>
<td>0</td>
<td>EVENT /LATCH8_TEST_BENCH/DO (value = &quot;ZZZZZZZZ&quot;)</td>
</tr>
<tr>
<td>6</td>
<td>EVENT /LATCH8_TEST_BENCH/DS1 (value = '1')</td>
</tr>
<tr>
<td></td>
<td>EVENT /LATCH8_TEST_BENCH/NDS2 (value = '1')</td>
</tr>
<tr>
<td>9</td>
<td>EVENT /LATCH8_TEST_BENCH/NDS2 (value = '0')</td>
</tr>
<tr>
<td></td>
<td>EVENT /LATCH8_TEST_BENCH/R1/ENBLD (value = '1')</td>
</tr>
<tr>
<td></td>
<td>EVENT /LATCH8_TEST_BENCH/DO (value = &quot;00000000&quot;)</td>
</tr>
<tr>
<td>11</td>
<td>EVENT /LATCH8_TEST_BENCH/DS1 (value = '0')</td>
</tr>
<tr>
<td></td>
<td>EVENT /LATCH8_TEST_BENCH/R1/ENBLD (value = '0')</td>
</tr>
<tr>
<td></td>
<td>EVENT /LATCH8_TEST_BENCH/DO (value = &quot;ZZZZZZZZ&quot;)</td>
</tr>
<tr>
<td>14</td>
<td>EVENT /LATCH8_TEST_BENCH/DS1 (value = '1')</td>
</tr>
<tr>
<td></td>
<td>EVENT /LATCH8_TEST_BENCH/R1/ENBLD (value = '1')</td>
</tr>
<tr>
<td></td>
<td>EVENT /LATCH8_TEST_BENCH/DO (value = &quot;00000000&quot;)</td>
</tr>
<tr>
<td>16</td>
<td>EVENT /LATCH8_TEST_BENCH/DATA (value = X&quot;AA&quot;)</td>
</tr>
<tr>
<td></td>
<td>EVENT /LATCH8_TEST_BENCH/CLK (value = '1')</td>
</tr>
<tr>
<td>19</td>
<td>EVENT /LATCH8_TEST_BENCH/CLK (value = '0')</td>
</tr>
<tr>
<td></td>
<td>WRITE /LATCH8_TEST_BENCH/R1/LATCH_14/VAR (value = X&quot;AA&quot;)</td>
</tr>
<tr>
<td></td>
<td>EVENT /LATCH8_TEST_BENCH/R1/REG (value = X&quot;AA&quot;)</td>
</tr>
<tr>
<td></td>
<td>EVENT /LATCH8_TEST_BENCH/DO (value = &quot;10101010&quot;)</td>
</tr>
</tbody>
</table>

**Figure 5. Textual Output of Synopsys Simulator**

The second form of output available by the simulator is the waveform output. Figure 6 shows an example of the Synopsys Waveform Viewer output. This form of output can be easier to create than the text form if the Synopsys VHDL Debugger is used, but in order to save the monitors (or traces in this case) used during simulation, you must still create a simulation control file containing all monitors and include it in the simulation while using the Synopsys Debugger.
Figure 6. Synopsys Waveform Viewer Output

4.2 Consistency with the Design Environment

The output formats listed in the last section obviously require a good amount of interaction from the user for their setup. However this is just one drawback of using these types of simulation results. The first form of output is simply a textual form which lists all events or activations (depending on the monitor) with their times. These results give no correlation to the model other than the signal names. The designer has no idea which processes this signal drives or propagates toward. The designer must depend fully on his or her memory of the model process "structure". Also it is difficult to determine, without constant scanning of the output, the sequence of the signal events, i.e. to determine the cause of an event, the designer must serially scan the previous text in the list.
The second form of output, the waveform, also lacks correlation to the design model. Waveforms give no indication as to which processes drive them or to which processes they drive. For instance, if an output of a process drives an input for two different processes, the designer cannot conclude this from simply looking at the waveforms. The designer would in this case have to refer to his or her memory or to some sort of design diagram. To determine dependencies among the signals, the user must track two separate waveforms and determine the order of signal transitions.

These output formats are quite a contrast to the methodology used in designing the system. In using a PMG to design a system, the designer creates a VHDL model by creating a graphical representation of the process and signal dependencies. The PMG indicates signals which are control signals for processes. The PMG shows the full process "structure" of the behavioral model. This is one advantage of using a PMG in the design of a VHDL behavioral model.

The nature of the Process Model Graph as a kind of "structural" representation for a VHDL model, along with the lack of correlation between existing types of simulation results, yields a natural solution. Back annotating the simulation results into the Process Model Graph to form a new simulation environment for verifying the model is the solution. This is a clear solution to the problems with current simulation formats. It solves the problem concerning the lack of signal interdependency information since the PMG is a directed graph showing the relation of processes to signals and the process sensitivity to signals. For instance, when a signal event causes the latching of a data value, the PMG shows the event and the designer can note the data value present on the latch process data input.

Using the PMG for simulation also solves the problem of correlating the results to the design environment. With the simulation viewed on the PMG, the designer no longer
has to visualize or transform the simulation output to match his or her view of the behavioral model. The model seen in the simulation environment can match the design model exactly, and the designer can be more assured of the correctness (or incorrectness) of the simulation. This philosophy is stated in the following observation:

**Observation:** *Since the Process Model Graph is the primary tool used in the design of a behavioral model, back annotation of the simulation results onto the PMG will result in a more consistent and secure development environment.*

### 4.3 Correspondence to Test Generation Techniques

Since the test generation programs use the PMG is for more than purely design, the role of the PMG in the simulation domain extends for test generation uses as well. As stated in the previous chapter, the Process Model Graph provides a foundation and a channel in which the generated behavioral tests for a model can be exercised and propagated. Therefore, the developer can use the back annotation of simulation results into the PMG as a method of verifying that the test generation process is successful.

To verify the test generation process, the user is in effect verifying that the HBTG algorithm is functioning as desired. As stated in [2], the major criterion used by the HBTG algorithm is to activate as many sensitive ports of the VHDL behavioral model as possible. An **effective test sequence** for a VHDL model is defined as one which generates an event on every sensitive port of the model at least once.¹ By using the PMG as a simulation medium, we are able to verify that this is occurring. In order to aid in this verification, we can also annotate onto the PMG the number of events that have occurred

¹ In this thesis, we will often refer to a sensitive port which has had an event as being **tested**.
on a port or signal. This provides direct verification that an event has occurred on each signal.

Another form of verification of the HBTG algorithm is to actually verify that the steps of the algorithm are being performed. In order to do this, the user would need to be aware of the steps used in the HBTG algorithm [2]. As mentioned earlier, the basic steps of the algorithm are activation - propagation - justification. By back annotating the simulation results into the PMG and stepping forward and backward through the simulation, the designer can verify each step of the algorithm. An event on a sensitive input port indicates activation. The movement of a signal from a primary input toward the primary outputs of the model indicates propagation. A value in the PMG which allows for the propagation of an event to a process output, for example, an enable signal, indicates justification.

In order to provide another method of verifying both the effectiveness of the HBTG algorithm and the steps within the algorithm itself, the view of the process model graph can be modified to show both the testing of a port or signal and the propagation of the tests. In order to show that a signal has not yet been tested, the signal can be represented by a broken, or dashed, line. The names of ports and signals that are not tested can be shown in italics. As tests propagate through the PMG, they are shown by the solidifying of lines and the changing of italic fonts to normal. The movement of these tests gives the model a type of "flow" that cannot be seen in other simulation formats.

4.4. Relation between Simulation Results and Test Paths

As mentioned in the previous chapter, the tests for a Process Model Graph are developed by first creating sensitive paths through the model and then applying the HBTG
algorithm to the model using the generated paths. This follows the notion of path testing, in which a set of test paths is constructed through a model which, when exercised, should achieve some degree of thoroughness in the test generation process [2].

The use of paths through the model as a basic framework for test generation, along with the changes listed in the previous section for verifying events through the model, imply another way of verifying the test generation algorithm. The actual test paths can be back annotated onto the PMG by numbering the paths through the model and then labeling each port and signal along a path with its path number. This will allow the designer to visually verify that the paths generated are appropriate for the model. The designer will also be able to use the changes of the previous section to visualize the "flow" of tests along the paths. This type of visual verification of path testing is not directly available using any other simulation environment.

4.5 Visualization and Animation of Simulation in a PMG Form

The previous sections in this chapter explained the usefulness and need for back annotating simulation results into the Process Model Graph. They also suggested ways in which the PMG can be enhanced as a tool for simulation verification. This section will show how the improvement of the Modeler's Assistant met these needs and implemented these enhancements. Many features have been added to the tool in an attempt to make a complete simulation environment within the tool.

In order to implement the back annotation of the simulation results onto the Process Model Graph, the simulation results are parsed and stored in a separate internal database within the Modeler's Assistant. This occurs when the user selects Simulate from
the main edit menu. Chapter 6 discusses this further. Once stored properly, the simulation results are ready for annotation onto the PMG. In order for the results to be viewed, however, the designer must first select the ports, variables, and signals he or she wishes to monitor. This is done by using the Add Monitor feature on the Simulate menu. This is a desired feature to allow the designer to focus on the ports or signals relevant to the design verification. Monitors are also removed from the PMG by using the Hide Monitor feature. Add All and Hide All menu selections are also available to simply add or hide all possible monitors.

Once the user selects the appropriate monitors for the design, the number of events which occurred through the current simulation time are concatenated onto the names of monitored ports, variables, and signals and placed in parentheses. The Modeler's Assistant also concatenates the simulation value for the current simulation time onto the end of the name after the event count. Italics show ports and signals which have not had an event. A dashed, rather than solid, line shows signals which have not had an event. For the sake of completeness, the user can also monitor variables and track the number of writes to the variable in a similar way to signal events. Figure 7 shows an example PMG with selected monitors.
Figure 7. 8-Bit Latch PMG with Monitors Selected

Once the user adds monitors, the next step in verifying the design is to step through the simulation and verify the simulation values. This is done using the Step Fwd and Step Back menu selections. Each step through the simulation updates the time listed at the bottom of the window. The Goto Time menu selection can be used to move to the nearest event time greater than or equal to the entered time. At each step, the designer
can verify that the functionality and state of the model is correct. The HBTG testing algorithm or any other procedure used in designing the test bench can be verified.

In order to contrast the simulation results to the sensitive test paths through the model, the designer can select the **Show Paths** option from the Simulate menu to see the enumerated test paths through the model, as suggested in the previous section. The designer is then able to step forward and backward through the simulation in this mode to see the tests being propagated along their paths and verify them. To return to the normal simulation view, the designer can select the **Hide Paths** option from the menu. Figure 8 shows a PMG in the simulation environment with the paths displayed.
As stated in [2] and in the previous chapter, another property used in evaluating test quality is *statement coverage*. To incorporate the coverage results into the simulation environment, a **Coverage** menu selection has been added to the Simulate menu. Selecting this option will display a window with the coverage results of simulation. The designer can then verify that all statements have executed as desired. Figure 9 shows an example of this.
architecture BEHAVIORAL of latch8 is
signal ENBLD: BIT;
signal REG: BIT_VECTOR(0 to 7);
begin

-- Process Name: OUTPUT

OUTPUT_4: process (ENBLD, REG)
begin
  if (ENBLD = '1') then
    DO <= BV_to_MVL(REG);
  else
    DO <= "0";
  end if;
end process OUTPUT_4;

-- Process Name: ENABLE

ENABLE_9: process (NDS2, DS1)
begin
  ENBLD <= DS1 and not NDS2;
end process ENABLE_9;

-- Process Name: LATCH

LATCH_14: process (CLK)
variable VAR: BIT_VECTOR(0 to 7);
begin
  if (CLK = '0') then
    REG <= DATA;
    VAR <= DATA;
  else
    REG <= VAR;
  end if;
end process LATCH_14;

Figure 9. Coverage Results Window for the 8-Bit Latch

Other useful features have been added to the simulation environment to allow for better verification of the model. In order to see the changes occurring on a particular signal, a Show Hist menu selection has been added to allow the designer to click on a signal and show all events that occur in the simulation for that signal. If the designer still
wishes to see a waveform output for the simulation, he or she can use the **Show Waves** selection to bring up the Synopsys Waveform Viewer showing the waveform output. Previously, Figure 6 showed an example waveform output. Figure 10 shows an example history output window.

Now that the improvements to the Modeler's Assistant have enhanced the usage of the Process Model Graph, a developer can completely simulate and verify a model within the tool. The next chapter will explain how this enhancement serves as an important part of the integration of design, test generation, and system verification. Chapter 6 will detail the design methodology used in creating the simulation environment and explain the reasoning behind its internals and data structures.
Figure 10. Signal History for the ENABLE Signal of the 8-Bit Latch
5.1 Previous Development Cycle Turnaround Time

The previous chapters gave a background on the environments for design, test generation, and simulation. This chapter will attempt to quantify the improvements that have been made to the Modeler's Assistant to decrease development time of VHDL behavioral models by integrating these environments. The environment before the improvement to the Modeler's Assistant was very heterogeneous. In order to design, generate tests, and simulate a system, many different programs were required to be run in a sequence of serialized steps. Many intermediate files and parameters were required and contributed to the distractions of using such an environment, and to the complexity of the development process. The old process is detailed in the steps given below:
The old development process:

Step 1: The designer creates a process model graph with the Modeler's Assistant based on the design specifications and analyzes the model.

Step 2: The user goes to the Process Edit menu to dump the VHDL code for each process.

Step 3: The user creates a VHDL model consisting of a single process using the MVHD program with appropriate parameters.

Step 4: The user calls the Synopsys Analyzer to analyze the process - if incorrect, the user must re-edit the process within the Modeler's Assistant.

Step 5: The user calls the Vtip Analyzer to create the needed data structures for generation of control flow graphs within the PTG.

Step 6: The user calls the PTG program, giving many parameters such as the source file, unit name, process name, architecture name, entity name, and design library name.

Step 7: The developer repeats steps 3-6 for each process in the PMG.

Step 8: The user calls the HBTG program to create the ".test" file for a unit PMG.
Step 9. The user calls the TBG program to produce the test bench file, giving it the appropriate parameters.

Step 10. The user calls the Synopsys Analyzer to analyze the test bench.

Step 11. The user creates a simulation control file, recalling the signal names which are necessary to monitor.

Step 12. The user calls the Synopsys Simulator to produce simulation output.

Step 13. The user compares either the waveform or textual output to the design either mentally or by comparison to some form of the model.

Step 14. If the design is correct, the process is complete, else the developer modifies the PMG and repeats the process at Step 2.

5.2 Improvements to the Development Cycle Through Environment Integration

The development process in the previous section has an obvious distribution among many tools. The purpose in the development of the Modeler's Assistant is to decrease the development time of VHDL models. Clearly centralizing these programs and development steps into the Modeler's Assistant will serve to greatly reduce development overhead time. The test generation tools have been interfaced into the Modeler's Assistant to allow for quick creation of a test bench for a VHDL model. A method of creating a custom test bench from the information stored in the PMG has also been developed. The
simulation environment was fully developed within the Modeler's Assistant to naturally integrate the design and simulation environments. This section will discuss the interface to the test generation tools which makes the integrated environment complete.

In order to develop the process level tests for a model, an interface was made into the process editing portion of the Modeler's Assistant. To create the VHDL code for a process and analyze it with the two different analyzers, the user just simply clicks on the Analyze and Analyze Vtip menu selections of the process menu. To then call the PTG program to generate the tests, the user simply clicks the Call PTG menu selection. A window then appears, and the user enters the appropriate information for data values to the PTG [3]. Figure 11 shows an example of this window. This completes the test generation for that process. The user then repeats the process test generation, moving to the other processes within the PMG. To then generate a test bench for the PMG, the user goes back to the unit menu and selects Test Bench. The Modeler's Assistant then calls HBTG in the background, usually without any user input except for feedback models, in which case a window appears asking for initial values. Next another window appears (Figure 12). The Test Bench Generator then prompts the user for any initial values needed and creates the test bench.

It should be noted that there is the ability to create a custom test bench (not automatically generated) as well. If the user does not automatically generate a test bench, and the user selects the Edit TB menu selection from the unit menu, the Modeler's Assistant will give the user the option to create their own custom test bench. If the user agrees, a window appears with the "shell" for a VHDL test bench file. The user only provides the tests for the model that go between the "begin" and "end" of the test bench. Figure 13 shows the input window used in creating a "custom" test bench.
Figure 11. The Process Test Generator Interface
Figure 12. The Test Bench Generator Interface
The Custom Test Bench Entry Window

After creation of the test bench, the user can analyze it using the Analyze TB menu selection. After this is successful, the user can then verify the model by selecting the Simulate menu option and using the simulation environment described in the previous chapter. Using these interfaces to the different environments, the developer can completely develop and test a VHDL model entirely within the Modeler's Assistant.

Chapter 5. Integration of Design, Test Generation, and Verification into a Single Environment
Chapter 6. The Simulation System Design Methodology

6.1 Contribution of the Existing PMG Data Structure to Simulation

This chapter explains the design decisions and methodology used in creating the simulation environment portion of the Modeler’s Assistant. Although much of the design for the environment was done within a separate data structure for simulation, the design needed certain portions of the existing data structure used to store the Process Model Graph. Since the PMG data structure was used to build the structures for the graphics display of the PMG, it can link to the simulation data structure in a convenient way to aid in the display of the simulation graphics.

The existing data structure for the PMG consists of a large (800 element) array of nodes. Each node in the PMG data structure represents one element of the PMG. Nodes exist for the entire unit, processes, ports, signals, variables, generics, constants, supernodes, and supernode ports. Other nodes exist for things such as the signal source
and destination, port maps, and intermediate linking nodes between processes. Reference [1] and Appendix B give more extensive details on this data structure. The primary link between the PMG and the simulation data structure is the node number of the PMG elements which the user marks as monitors within the simulation environment. The node number of the element is simply its index number into the array of nodes. The Modeler's Assistant stores each event of the simulation with a node number linking it to a part of the PMG. Since much of the information for a port or signal, such as name and type, are stored within the PMG data structure, the Modeler's Assistant uses this link often in extracting descriptive information about the signals of the model.

The existing PMG structure is also useful in portions of the setup for simulation and the user interface. For example, the control file for the simulation determines the signals to monitor by searching through the PMG database and creating appropriate signal monitors for the ports and signals of the PMG. The existing menu system of the Modeler's Assistant was also extended to allow for the simulation menu.

In order to allow the user to select signals to monitor or display the history, the Modeler's Assistant searches the PMG database for the closest port to the coordinate where the user clicked the mouse button. Once the port or signal is located, the Modeler's Assistant marks the port or signal node within the PMG database as monitored by setting previously unused bits of the node information. The graphics routines then use this information in determining how to display portions of the PMG.

The Modeler's Assistant uses the graphics information stored in the PMG database in drawing the simulation PMG. The drawing routines that existed for drawing the PMG were modified extensively for annotation of simulation values or test paths and for the addition of the new graphics contexts needed in the display of untested ports and signals.
With these modifications, the Modeler's Assistant could use the same drawing routines to display the simulation PMG as used in the design environment.

6.2 The Choice of Data Structure to Store Events

As mentioned earlier, an entirely new data structure was created within the Modeler's Assistant to store the events of simulation. This section explains the reasons for using this structure and how the Modeler's Assistant uses the structure to hold the simulation values.

Deo [14] gives the mechanism for the flow of time in a simulation model. Figure 14 [14] gives the flowchart for discrete next-event simulation. As can be seen by the flow chart cycle, in order to quickly find the next event of the simulation, a data structure which can easily and quickly be indexed based on time is needed. If the user is to be able to step to any simulation time, it should be easy to determine from the data structure which events occur at any given time. For test verification purposes, it should also be easy to ascertain from the data structure the times at which any given signal or port has an event. For example, to determine the history of a signal, it should be easy to move through the data structure and collect the times of all events on that signal. Information needed for the display of the PMG in the simulation environment is also an important consideration for the data structure. At any given simulation time, new events should be collected and displayed annotated onto the PMG.
Figure 14. Flowchart for Simulation of Events

These considerations imply the need for a data structure in which the events can be traversed in two "directions". The data should be able to be referenced by time or by signal name. All events at a particular time or for a particular signal should be gathered easily. In order to step to the next simulation time, there should be an easy path to follow to the index for that time.
Another consideration in the choice of data structure relates to the way in which one adds elements to that structure. In the case of back annotation of simulation results, the parsing of the simulation output is unpredictable in that any signal could receive a new event element at any time. It is also not guaranteed that an event will occur at any particular time in the output. Therefore, a structure is needed in which elements can be added "anywhere" quickly and easily.

The unpredictability of the simulation output brings about another important factor in the data structure design. Since events could be present indefinitely within the simulation, there is no optimal fixed size for the data structure to store the events. Allocating a large fixed size "guess" for the structure size that would handle most cases would not only be restrictive, but would waste valuable memory resources.

These considerations in the design of the data structure resulted in only one suitable solution - a two dimensional quadruple linked list. Aho, Hopcroft, and Ullman [15] give the advantages of using a linked list. "Lists are a particularly flexible structure because they can grow and shrink on demand, and elements can be accessed, inserted, or deleted at any position within a list." Figure 15 shows the structure chosen to hold the simulation events. The four boxes in the lower right corner of the figure are elements of the structure. Each element has four links from it to other elements. One can traverse this structure in two "directions". The prev_sig and next_sig pointers shown in the figure represent the links among signal events at the same simulation time. This chain of events for the same time can be referred to as a uniform time list. The prev_time and next_time pointers represent the links among all event times for a particular signal. This chain of events for the same signal can be called a uniform signal list. These bi-directional links meet the needs for accessing the structure mentioned earlier. The Modeler's Assistant also uses dynamic allocation for this structure. This means that the size of the data structure
only depends on how many events need to be stored. There is no "wasted" space within
the structure. Due to the way that tests are generated using time frames for each test, and
due to the use of delta delay for all signal assignments within the model, events often
occur at the same time. This allows for an even distribution of events throughout the
structure.

This linked list structure also provides the ability to insert items easily. No sorting
of the events is necessary while parsing the simulation results. For example, in order to
insert a new event for a signal, the program simply finds the end of the uniform signal list
for the signal and the end of the uniform time list for the time of the event, allocates a new
event node, and sets the pointers to attach the new node to the end of each list. This is a
very easy process due to the design of the data structure.

In order to index the data structure quickly and easily, two other pointer structures
have been used. The two boxes in the upper portion of Figure 15 are part of a double-
linked list of time nodes which make up a time list. Each time node in the list serves as a
pointer to the beginning and end of one uniform time list. These pointers are the
sig_list_head and sig_list_tail pointers of the figure. The unique event time for the
uniform time list is also stored within the node. In order to move to a particular time, the
program simply traverses the time list until the desired time is reached, then affects the
events of the associated uniform time list.

The boxes at the left of Figure 15 represent a dynamic array of monitor nodes. A
dynamic array instead of a list was used here since the program knows the number of
signals monitored when calling the simulator and entering the simulation environment,
because searching the array for a monitor is simple, and because there is never a need to
insert into the middle of the array. A dynamic structure was again used to conserve
memory. The purpose of the array of monitors is to serve as a pointer to the beginning
and end of one uniform signal list. These pointers are the list_head and list_tail pointers of the figure. The Modeler's Assistant determines the name of the signal by using the nodenum field as an index into the PMG database.

Figure 15. Overview of the Simulation Environment Data Structure
6.3 Mapping of the Simulation Data Structure to the Simulation Environment

The last section showed how the data structure to hold simulation events was chosen and implemented. It described the foundation over which the simulation environment functions. This section will detail the ways in which the data structure for events was mapped into the simulation environment and into the routines used in its display.

The only significant difference between the simulation environment and the flow chart of Figure 14 is that all of the gathered statistics about the simulation are available at any point while the simulation environment is being used. The basic function of the flow chart is followed as the user steps forward and back through the simulation, but since the Modeler's Assistant has already created the entire data structure from which it gathers information, the statistics in the last block of the flow chart can be output at any time. For example, the user can view the history of a signal (all of its events during simulation) at any time.

In order to track when a signal becomes tested, i.e., has its first event after time zero, each node in the array of monitors includes a field event_time. As the data structure is built, the first event time for a signal past time zero is stored in this field. As the user steps through the simulation, the current simulation time is compared to this time to determine how to display the port or signal on the PMG.

The elements of the quadruple linked list of events and the array of monitors both contain the node number of the PMG element corresponding to the signal or port. This allows information about the ports and signals to be easily extracted from the PMG.
database. For example, this is the way that the Modeler’s Assistant determines the name of a signal within the structure. Also, the event nodes store the time of the event. This helps by not requiring the time list to be searched when only event times for a single signal are needed.

To track the number of events, a dynamic array of integers is overlaid onto the PMG database. When the user moves to a new time in the simulation, the Modeler’s Assistant increments or decrements the event count for the signals of the arriving time (if stepping forward) or departing time (if moving backward). The elements of the event_counts array are indexed by the same number as the PMG database. This makes it easy to find the event count for a PMG port or signal when displaying the PMG. Since the Modeler’s Assistant stores the PMG node numbers with the events, it also makes the counts array easy to access when searching the uniform time lists.

In order to hold the current state of a signal while using the simulation environment, the Modeler’s Assistant uses another array overlay. The event_values structure stores pointers to the current event value for a given PMG element. Therefore, these pointers actually point to the event value stored within the simulation data structure. This is simply a dynamically allocated array of pointers to strings. These pointers are moved to new simulation values as the user steps through the simulation by using the uniform signal list.

The last of the data structures used in the display of information to the simulation environment is the static array used to store the paths through the model. The Modeler’s Assistant parses the HBTG test file for the path numbers and their elements, and then stores the paths in the array. It stores the path number and the port PMG node number in each array element. When the Modeler’s Assistant displays paths for a PMG, it searches
this array by node number for each PMG element and, if found, displays the path number beside the PMG port or signal element corresponding to the node number.

The programming manual (Appendix B) reiterates and extends the detailed information about the C programming of the data structure.

6.4 The Simulation Environment Algorithm

The last section and Appendix B discuss the mechanisms for building the simulation data structure. The pseudocode below gives the basic algorithm for adding the events into the data structure. Correct design of the data structure was crucial in making the algorithm so compact.

While more simulation results exist repeat
    Get event-time value from simulation output
    If time of event <> last event time
        Allocate a new time list node for the new time
        Attach new time node to the time list
        Move insertion time pointer to the new node
    end
    Allocate a new event structure node for the current event
    For each occurrence of the name in the simulation monitor list repeat
        Attach the new event node to the appropriate uniform signal list
        Attach the new event node to the end of the uniform time list pointed to by the current time list node
    end
end

The flow chart of Figure 14 was the design basis for the basic algorithm for interacting with the simulation data structure. The pseudocode below gives the basic algorithm for moving from time to time within the structure and displaying results:
Start
Set time = 0
Update display
While in the simulation environment repeat
  If Step Fwd selected
    Move the current simulation time pointer to the next time
    Update event_values for the new event values at the current time
    Increment event_count for events at the current time
  end
  If Step Back selected
    Decrement the event_count for events at the current time
    Change event_values of signals at the old time to the prev value
    Move the current simulation time pointer to the next time
  end
  If Goto Time selected
    Reset event_counts and event_values
    Move to time 0
    while time < desired time repeat
      Increment event_counts
      Update event_values
      Move the current simulation time pointer to the next time
    end
    Increment event_counts
    Update event_values
  end
  Update the display with the new information
end

These given algorithms are just the basic pseudocode dealing with the building and time traversal through the data structure. Algorithms for the display of windows, drawing routines, menus, and others are integrated with all other parts of the Modeler's Assistant and are too complex to be explained in pseudocode.
Chapter 7. Other Enhancements

7.1 External Signals

In creating a behavioral model, it is sometimes desirable to create entity ports for signals shared among the processes of the entity. In previous versions of the Modeler's Assistant, this was not possible since signals shared among processes were automatically declared in the architecture section of the entity. This made it impossible for test generation programs like HBTG to generate tests for feedback signals, since they lie among processes within the model.

The solution to this problem was to create an external signals feature for the Modeler's Assistant. This feature allows the user to mark an external signal within the PMG by selecting the source or destination port of the signal. Once selected, these signals are marked by placement of a second concentric circle around the perimeter of the source and destination ports of the signal. Figure 16 shows an external signal.
Figure 16. 8 Bit Latch PMG with an External Signal

External ports allow flexibility in the scope of a signal. The signals marked as external are declared in the entity declaration rather than the architecture, so they can be accessed outside the entity in a structural model. These signals are of mode “inout” since they are both read and driven.
Signals within supernodes can also be marked as external. When this occurs, another port is drawn on the perimeter of the supernode to represent an attachment to a signal internal to the supernode. These signals also must be of mode "inout". The use of the external signal feature allows any signal within a model to be accessed at any level of abstraction. Figure 17 shows an external port within a supernode.
7.2 Enhancement of Other Features and Bug Fixes

There were many other smaller features added and modifications made to the Modeler's Assistant. These changes are enumerated and described below:
1. The method of renaming a signal changed so that all signals connected to the same port are renamed. This should have been an obvious requirement for renaming a signal since a process port only represents one signal.

2. Removed the limitation on the number of variable length process ports. This is simply a bug fix dealing with memory allocation.

3. Parameters within primitive code can now be used with variables. The earlier version only allowed parameters to be specified for signals within primitives. Now, variables can take on the appropriate parameters as well.

4. Vectors of equal length but different ranges can now be connected by signals. This allows, for example, a bit vector of range 0 to 7 to be connected to a bit vector of range 1 to 8. In earlier versions, the ranges had to exactly match.

5. Added #portname.Z and #portname.X primitive parameters to allow variable length strings of "Z's" and "X's" to assign to signals or variables in the primitive code. This was necessary to make primitive parameters flexible enough to use with multi-valued logic. These parameters within a process primitive code will be replaced by the appropriate length vector determined from the port type.

6. For convenience, all moving of signals, ports, etc. is repetitive until blank space is chosen. This makes restructuring a PMG faster and easier.
7. Confirmation is now sought before overwriting existing units. This is to help protect the user against the accidental loss of valuable models.

8. Changing a port, variable, or signal name now scans the process code and replaces the name. This is actually one of the more beneficial changes. Before, when changing a port name, the developer had to re-edit the process code and change all occurrences of the old port name to the new name. Now the Modeler’s Assistant will scan the VHDL and make the changes itself.

9. Bit vectors can now be split without using the splitter primitives. In earlier versions, there was no way of connecting a bit or subrange of a vector type to a smaller sized vector. In the new version, when a user tries to connect ports with the same base type but different sized ranges, the Modeler's Assistant prompts the user for the subrange of the vector signal to use for the smaller port. The Modeler's Assistant then uses this range when replacing the port name in the process while generating the VHDL code for an entity. The use of subranges is only available for signals which fanout to or from a single process port. For instance, if the user connects a signal to the smaller end of a signal, then the same subrange must be used for the new signal as in the smaller port. This is done because subranges of another subrange are not allowed or needed. Any new subrange can be taken from the larger end of the signal.

10. An interface to Meenakshi Manek’s English Specification Tool [16] has been made into the Process Edit menu to allow the use of this program to create the process code by writing an English description of the functionality.
Chapter 8. Approach to Model Development

8.1 The Design Process as a Whole

Figure 18 shows the new VHDL development environment in use at Virginia Tech. The entire development system runs on Sun Sparc II Workstations, and can be ported to any system with the Synopsys VHDL Analyzer and Simulator [12], the CLSI Vtip system, the appropriate X-Window C libraries, and source code for all the programs. All development is now based around the Modeler’s Assistant unified development environment. The approach to model development with the Modeler’s Assistant is summarized in the following steps:
Figure 18. VHDL Development Environment

1. Partition the design requirement functionality into parts which can be represented by VHDL processes with a few shared signals among them.

2. If one desires a hierarchical design or structural model, group the processes into groups of similar function (to become supernodes).
3. Design the Process Model Graph for the model, using the primitive process library when possible.

4. If one desires a grouping of processes or a structural VHDL model, create supernodes from the process groups.

5. Analyze the model for syntactical correctness.

6. Generate the process level tests for any process which does not already have generated tests using the procedures described in Chapter 5.

7. Call the test generation tools to automatically generate the tests and test bench for the entire model, or, optionally, create a customized test bench for the VHDL model. Note that current test generation tools cannot handle models with supernodes.

8. Analyze the test bench.

9. Simulate and verify the model using the simulation environment described in this thesis. Verification could include verifying test paths through the model, the PMG functionality, the model coverage, timing considerations, or other functionality.

10. If functional errors are found, modify the PMG or specified VHDL code accordingly.
11. Use the verified VHDL model for further purposes, e.g., synthesis.

This procedure greatly reduces the development and verification time for VHDL models. The following section will give examples which use the above development process.

8.2 Examples

Previous chapters gave the method for using the simulation environment and verifying tests. This chapter gives examples where various models use the simulation environment.

References [2, 3] give examples of using the test generation tools. Each example gives the Process Model Graph along with the test bench used for the model. Next, the example shows the display of test paths through the model. Each following page then steps through the simulation showing the simulation environment at each stage. Some simulation steps in the later stages of simulation after the paths have been tested are omitted since they are very similar to the earlier steps.
Figure 19. Model IOSYS (Input/Output Subsystem) [2]
use work.VHDLCAD.all, work.all;

entity IOSYS_TEST_BENCH is
end IOSYS_TEST_BENCH;

architecture BEHAVIOR of IOSYS_TEST_BENCH is
signal SEL0, SEL1, ENB1, ENB2, NDS2, DS1, STRB2, STRB1: BIT;
signal BUF5, BUF4, BUF3, BUF2, BUF1: MVL_VECTOR(0 to 7);
signal DATA2, DATA1: BIT_VECTOR(0 to 7);

component IOSYS_A
  port( SEL0: in BIT; SEL1: in BIT; ENB1: in BIT; ENB2: in BIT; BUF5: out MVL_VECTOR(0 to 7); NDS2: in BIT; DS1: in BIT; BUF4: out MVL_VECTOR(0 to 7); BUF3: out MVL_VECTOR(0 to 7); STRB2: in BIT; DATA2: in BIT_VECTOR(0 to 7); BUF2: out MVL_VECTOR(0 to 7); BUF1: out MVL_VECTOR(0 to 7); STRB1: in BIT; DATA1: in BIT_VECTOR(0 to 7));
end component;

for all: IOSYS_A use entity work.IOSYS(BEHAVIORAL);

begin

R1: IOSYS_A
  port map(SEL0, SEL1, ENB1, ENB2, BUF5, NDS2, DS1, BUF4, BUF3, STRB2, DATA2, BUF2, BUF1, STRB1, DATA1);

process
begin

  SEL0 <= transport '0' after 1 ns;
  SEL1 <= transport '0' after 1 ns;
  ENB1 <= transport '0' after 1 ns;
  ENB2 <= transport '0' after 1 ns;
  NDS2 <= transport '0' after 1 ns;
  DS1 <= transport '0' after 1 ns;
  STRB2 <= transport '0' after 1 ns;
  DATA2 <= transport "00000000" after 1 ns;
  STRB1 <= transport '0' after 1 ns;
  DATA1 <= transport "00000000" after 1 ns;

  SEL0 <= transport '0' after 4 ns;

end process;

Chapter 8. Approach to Model Development
SEL1 <= transport '0' after 4 ns;
ENB1 <= transport '0' after 4 ns;
ENB2 <= transport '0' after 4 ns;
NDS2 <= transport '0' after 4 ns;
DS1 <= transport '0' after 4 ns;
STRB2 <= transport '0' after 4 ns;
DATA2 <= transport "00000000" after 4 ns;
STRB1 <= transport '0' after 4 ns;
DATA1 <= transport "00000000" after 4 ns;

SEL0 <= transport '0' after 6 ns;
SEL1 <= transport '0' after 6 ns;
ENB1 <= transport '0' after 6 ns;
ENB2 <= transport '0' after 6 ns;
NDS2 <= transport '0' after 6 ns;
DS1 <= transport '0' after 6 ns;
STRB2 <= transport '0' after 6 ns;
DATA2 <= transport "00000000" after 6 ns;
STRB1 <= transport '0' after 6 ns;
DATA1 <= transport "10101010" after 6 ns;

SEL0 <= transport '0' after 9 ns;
SEL1 <= transport '0' after 9 ns;
ENB1 <= transport '0' after 9 ns;
ENB2 <= transport '0' after 9 ns;
NDS2 <= transport '0' after 9 ns;
DS1 <= transport '0' after 9 ns;
STRB2 <= transport '0' after 9 ns;
DATA2 <= transport "00000000" after 9 ns;
STRB1 <= transport '1' after 9 ns;
DATA1 <= transport "10101010" after 9 ns;

SEL0 <= transport '0' after 11 ns;
SEL1 <= transport '1' after 11 ns;
ENB1 <= transport '1' after 11 ns;
ENB2 <= transport '1' after 11 ns;
NDS2 <= transport '0' after 11 ns;
DS1 <= transport '0' after 11 ns;
STRB2 <= transport '0' after 11 ns;
DATA2 <= transport "00000000" after 11 ns;
STRB1 <= transport '1' after 11 ns;
DATA1 <= transport "10101010" after 11 ns;
SEL0 <= transport '1' after 14 ns;
SEL1 <= transport '1' after 14 ns;
ENB1 <= transport '1' after 14 ns;
ENB2 <= transport '1' after 14 ns;
NDS2 <= transport '0' after 14 ns;
DS1 <= transport '0' after 14 ns;
STRB2 <= transport '0' after 14 ns;
DATA2 <= transport "00000000" after 14 ns;
STRB1 <= transport '1' after 14 ns;
DATA1 <= transport "10101010" after 14 ns;

SEL0 <= transport '0' after 16 ns;
SEL1 <= transport '0' after 16 ns;
ENB1 <= transport '1' after 16 ns;
ENB2 <= transport '1' after 16 ns;
NDS2 <= transport '0' after 16 ns;
DS1 <= transport '0' after 16 ns;
STRB2 <= transport '0' after 16 ns;
DATA2 <= transport "00000000" after 16 ns;
STRB1 <= transport '1' after 16 ns;
DATA1 <= transport "10101010" after 16 ns;

SEL0 <= transport '0' after 19 ns;
SEL1 <= transport '1' after 19 ns;
ENB1 <= transport '1' after 19 ns;
ENB2 <= transport '1' after 19 ns;
NDS2 <= transport '0' after 19 ns;
DS1 <= transport '0' after 19 ns;
STRB2 <= transport '0' after 19 ns;
DATA2 <= transport "00000000" after 19 ns;
STRB1 <= transport '1' after 19 ns;
DATA1 <= transport "10101010" after 19 ns;

SEL0 <= transport '0' after 21 ns;
SEL1 <= transport '1' after 21 ns;
ENB1 <= transport '1' after 21 ns;
ENB2 <= transport '1' after 21 ns;
NDS2 <= transport '0' after 21 ns;
DS1 <= transport '0' after 21 ns;
STRB2 <= transport '0' after 21 ns;
DATA2 <= transport "01010101" after 21 ns;
STRB1 <= transport '1' after 21 ns;
DATA1 <= transport "10101010" after 21 ns;
SEL0 <= transport '0' after 24 ns;
SEL1 <= transport '1' after 24 ns;
ENB1 <= transport '1' after 24 ns;
ENB2 <= transport '1' after 24 ns;
NDS2 <= transport '0' after 24 ns;
DS1 <= transport '0' after 24 ns;
STRB2 <= transport '1' after 24 ns;
DATA2 <= transport "01010101" after 24 ns;
STRB1 <= transport '1' after 24 ns;
DATA1 <= transport "10101010" after 24 ns;

SEL0 <= transport '1' after 26 ns;
SEL1 <= transport '0' after 26 ns;
ENB1 <= transport '0' after 26 ns;
ENB2 <= transport '1' after 26 ns;
NDS2 <= transport '0' after 26 ns;
DS1 <= transport '0' after 26 ns;
STRB2 <= transport '1' after 26 ns;
DATA2 <= transport "01010101" after 26 ns;
STRB1 <= transport '1' after 26 ns;
DATA1 <= transport "10101010" after 26 ns;

SEL0 <= transport '1' after 29 ns;
SEL1 <= transport '0' after 29 ns;
ENB1 <= transport '1' after 29 ns;
ENB2 <= transport '1' after 29 ns;
NDS2 <= transport '0' after 29 ns;
DS1 <= transport '0' after 29 ns;
STRB2 <= transport '1' after 29 ns;
DATA2 <= transport "01010101" after 29 ns;
STRB1 <= transport '1' after 29 ns;
DATA1 <= transport "10101010" after 29 ns;

SEL0 <= transport '0' after 31 ns;
SEL1 <= transport '0' after 31 ns;
ENB1 <= transport '1' after 31 ns;
ENB2 <= transport '0' after 31 ns;
NDS2 <= transport '0' after 31 ns;
DS1 <= transport '0' after 31 ns;
STRB2 <= transport '1' after 31 ns;
DATA2 <= transport "01010101" after 31 ns;
STRB1 <= transport '1' after 31 ns;
DATA1 <= transport "10101010" after 31 ns;

SEL0 <= transport '0' after 34 ns;
SEL1 <= transport '0' after 34 ns;
ENB1 <= transport '1' after 34 ns;
ENB2 <= transport '1' after 34 ns;
NDS2 <= transport '0' after 34 ns;
DS1 <= transport '0' after 34 ns;
STRB2 <= transport '1' after 34 ns;
DATA2 <= transport "01010101" after 34 ns;
STRB1 <= transport '1' after 34 ns;
DATA1 <= transport "10101010" after 34 ns;

SEL0 <= transport '0' after 36 ns;
SEL1 <= transport '0' after 36 ns;
ENB1 <= transport '1' after 36 ns;
ENB2 <= transport '1' after 36 ns;
NDS2 <= transport '0' after 36 ns;
DS1 <= transport '0' after 36 ns;
STRB2 <= transport '0' after 36 ns;
DATA2 <= transport "01010101" after 36 ns;
STRB1 <= transport '1' after 36 ns;
DATA1 <= transport "10101010" after 36 ns;

SEL0 <= transport '0' after 39 ns;
SEL1 <= transport '0' after 39 ns;
ENB1 <= transport '1' after 39 ns;
ENB2 <= transport '1' after 39 ns;
NDS2 <= transport '0' after 39 ns;
DS1 <= transport '0' after 39 ns;
STRB2 <= transport '1' after 39 ns;
DATA2 <= transport "01010101" after 39 ns;
STRB1 <= transport '1' after 39 ns;
DATA1 <= transport "10101010" after 39 ns;

SEL0 <= transport '0' after 41 ns;
SEL1 <= transport '0' after 41 ns;
ENB1 <= transport '1' after 41 ns;
ENB2 <= transport '1' after 41 ns;
NDS2 <= transport '1' after 41 ns;
DS1 <= transport '1' after 41 ns;
STRB2 <= transport '1' after 41 ns;
DATA2 <= transport "01010101" after 41 ns;
STRB1 <= transport '1' after 41 ns;
DATA1 <= transport "10101010" after 41 ns;

SEL0 <= transport '0' after 44 ns;
SEL1 <= transport '0' after 44 ns;
ENB1 <= transport '1' after 44 ns;
ENB2 <= transport '1' after 44 ns;
NDS2 <= transport '0' after 44 ns;
DS1 <= transport '1' after 44 ns;
STRB2 <= transport '1' after 44 ns;
DATA2 <= transport "01010101" after 44 ns;
STRB1 <= transport '1' after 44 ns;
DATA1 <= transport "10101010" after 44 ns;

SEL0 <= transport '0' after 46 ns;
SEL1 <= transport '0' after 46 ns;
ENB1 <= transport '1' after 46 ns;
ENB2 <= transport '1' after 46 ns;
NDS2 <= transport '0' after 46 ns;
DS1 <= transport '1' after 46 ns;
STRB2 <= transport '1' after 46 ns;
DATA2 <= transport "01010101" after 46 ns;
STRB1 <= transport '1' after 46 ns;
DATA1 <= transport "10101010" after 46 ns;

SEL0 <= transport '0' after 49 ns;
SEL1 <= transport '0' after 49 ns;
ENB1 <= transport '1' after 49 ns;
ENB2 <= transport '1' after 49 ns;
NDS2 <= transport '0' after 49 ns;
DS1 <= transport '0' after 49 ns;
STRB2 <= transport '1' after 49 ns;
DATA2 <= transport "01010101" after 49 ns;
STRB1 <= transport '1' after 49 ns;
DATA1 <= transport "10101010" after 49 ns;

SEL0 <= transport '0' after 51 ns;
SEL1 <= transport '0' after 51 ns;
ENB1 <= transport '1' after 51 ns;
ENB2 <= transport '1' after 51 ns;
NDS2 <= transport '0' after 51 ns;
DS1 <= transport '0' after 51 ns;
STRB2 <= transport '0' after 51 ns;
DATA2 <= transport "01010101" after 51 ns;
STRB1 <= transport '1' after 51 ns;
DATA1 <= transport "10101010" after 51 ns;

SEL0 <= transport '0' after 54 ns;
SEL1 <= transport '0' after 54 ns;
ENB1 <= transport '1' after 54 ns;
ENB2 <= transport '1' after 54 ns;
NDS2 <= transport '0' after 54 ns;
DS1 <= transport '0' after 54 ns;
STRB2 <= transport '1' after 54 ns;
DATA2 <= transport "01010101" after 54 ns;
STRB1 <= transport '1' after 54 ns;
DATA1 <= transport "10101010" after 54 ns;

SEL0 <= transport '0' after 56 ns;
SEL1 <= transport '0' after 56 ns;
ENB1 <= transport '1' after 56 ns;
ENB2 <= transport '1' after 56 ns;
NDS2 <= transport '0' after 56 ns;
DS1 <= transport '0' after 56 ns;
STRB2 <= transport '1' after 56 ns;
DATA2 <= transport "01010101" after 56 ns;
STRB1 <= transport '0' after 56 ns;
DATA1 <= transport "11111111" after 56 ns;

SEL0 <= transport '0' after 59 ns;
SEL1 <= transport '0' after 59 ns;
ENB1 <= transport '1' after 59 ns;
ENB2 <= transport '1' after 59 ns;
NDS2 <= transport '0' after 59 ns;
DS1 <= transport '0' after 59 ns;
STRB2 <= transport '1' after 59 ns;
DATA2 <= transport "01010101" after 59 ns;
STRB1 <= transport '1' after 59 ns;
DATA1 <= transport "11111111" after 59 ns;
wait;
end process;
end BEHAVIOR;

---

**Figure 20. Test Bench for IOSYS**
Figure 21. IOSYS Test Paths
Figure 22. IOSYS Test Paths after 49 NS of Simulation
Figure 23. IOSYS after 0 NS of Simulation
Figure 24. IOSYS after 6 NS of Simulation
Figure 25. IOSYS after 9 NS of Simulation
Figure 26. IOSYS after 11 NS of Simulation
Figure 27. IOSYS after 14 NS of Simulation
Figure 28. IOSYS after 16 NS of Simulation
Figure 29. IOSYS after 19 NS of Simulation
Figure 30. IOSYS after 21 NS of Simulation
Figure 31. IOSYS after 24 NS of Simulation
Figure 32. IOSYS after 26 NS of Simulation
Figure 33. IOSYS after 29 NS of Simulation
Figure 34. IOSYS after 31 NS of Simulation
Figure 35. IOSYS after 41 NS of Simulation
Figure 36. IOSYS after 44 NS of Simulation
Figure 37. IOSYS after 49 NS of Simulation
Name: BUS1

<table>
<thead>
<tr>
<th>Time</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>9 ns</td>
<td>X&quot;AA&quot;</td>
</tr>
<tr>
<td>59 ns</td>
<td>X&quot;FF&quot;</td>
</tr>
</tbody>
</table>

Figure 38. History of the BUS1 Signal
Figure 39. Coverage Window for IOSYS Model
Figure 40. Waveform Window for IOSYS Model
Figure 41. Model TWOSTMUX (A Two Stage Multiplexer)
use work.VHDLCAD.all, work.all;

entity TWOSTMUX_TEST_BENCH isend TWOSTMUX_TEST_BENCH;

architecture BEHAVIOR of TWOSTMUX_TEST_BENCH is
signal EN, SEL1, SEL0: BIT;
signal QOUT, IN3, IN2, IN1, IN0: BIT_VECTOR(7 downto 0);

component TWOSTMUX_A
port( EN: in BIT; QOUT: out BIT_VECTOR(7 downto 0); SEL1: in BIT; SEL0: in BIT; IN3: in BIT_VECTOR(7 downto 0); IN2: in BIT_VECTOR(7 downto 0); IN1: in BIT_VECTOR(7 downto 0); IN0: in BIT_VECTOR(7 downto 0));
end component;

for all: TWOSTMUX_A use entity work.TWOSTMUX(BEHAVIORAL);

begin

R1: TWOSTMUX_A
port map(EN, QOUT, SEL1, SEL0, IN3, IN2, IN1, IN0);

process
begin

EN <= transport '0' after 1 ns;
SEL1 <= transport '0' after 1 ns;
SEL0 <= transport '0' after 1 ns;
IN3 <= transport "00000000" after 1 ns;
IN2 <= transport "00000000" after 1 ns;
IN1 <= transport "00000000" after 1 ns;
IN0 <= transport "00000000" after 1 ns;

EN <= transport '0' after 4 ns;
SEL1 <= transport '0' after 4 ns;
SEL0 <= transport '0' after 4 ns;
IN3 <= transport "00000000" after 4 ns;
IN2 <= transport "00000000" after 4 ns;
IN1 <= transport "00000000" after 4 ns;
IN0 <= transport "00000000" after 4 ns;


Chapter 8. Approach to Model Development
EN <= transport '0' after 6 ns;
SEL1 <= transport '0' after 6 ns;
SEL0 <= transport '0' after 6 ns;
IN3 <= transport "10101010" after 6 ns;
IN2 <= transport "01010101" after 6 ns;
IN1 <= transport "11111111" after 6 ns;
IN0 <= transport "01111111" after 6 ns;

EN <= transport '1' after 9 ns;
SEL1 <= transport '0' after 9 ns;
SEL0 <= transport '0' after 9 ns;
IN3 <= transport "10101010" after 9 ns;
IN2 <= transport "01010101" after 9 ns;
IN1 <= transport "11111111" after 9 ns;
IN0 <= transport "01111111" after 9 ns;

EN <= transport '1' after 11 ns;
SEL1 <= transport '0' after 11 ns;
SEL0 <= transport '0' after 11 ns;
IN3 <= transport "10101010" after 11 ns;
IN2 <= transport "01010101" after 11 ns;
IN1 <= transport "11111111" after 11 ns;
IN0 <= transport "01111111" after 11 ns;

EN <= transport '1' after 14 ns;
SEL1 <= transport '0' after 14 ns;
SEL0 <= transport '0' after 14 ns;
IN3 <= transport "10101010" after 14 ns;
IN2 <= transport "01010101" after 14 ns;
IN1 <= transport "11111111" after 14 ns;
IN0 <= transport "01111111" after 14 ns;

EN <= transport '1' after 16 ns;
SEL1 <= transport '1' after 16 ns;
SEL0 <= transport '0' after 16 ns;
IN3 <= transport "10101010" after 16 ns;
IN2 <= transport "01010101" after 16 ns;
IN1 <= transport "11111111" after 16 ns;
IN0 <= transport "01111111" after 16 ns;

EN <= transport '1' after 19 ns;
SEL1 <= transport '0' after 19 ns;
SEL0 <= transport '0' after 19 ns;
IN3 <= transport "10101010" after 19 ns;
IN2 <= transport "01010101" after 19 ns;
IN1 <= transport "11111111" after 19 ns;
IN0 <= transport "01111111" after 19 ns;

EN <= transport '0' after 21 ns;
SEL1 <= transport '0' after 21 ns;
SEL0 <= transport '0' after 21 ns;
IN3 <= transport "10101010" after 21 ns;
IN2 <= transport "01010101" after 21 ns;
IN1 <= transport "11111111" after 21 ns;
IN0 <= transport "01111111" after 21 ns;

EN <= transport '1' after 24 ns;
SEL1 <= transport '0' after 24 ns;
SEL0 <= transport '0' after 24 ns;
IN3 <= transport "10101010" after 24 ns;
IN2 <= transport "01010101" after 24 ns;
IN1 <= transport "11111111" after 24 ns;
IN0 <= transport "01111111" after 24 ns;

EN <= transport '1' after 26 ns;
SEL1 <= transport '0' after 26 ns;
SEL0 <= transport '0' after 26 ns;
IN3 <= transport "10101010" after 26 ns;
IN2 <= transport "01010101" after 26 ns;
IN1 <= transport "11111111" after 26 ns;
IN0 <= transport "01111111" after 26 ns;

EN <= transport '1' after 29 ns;
SEL1 <= transport '0' after 29 ns;
SEL0 <= transport '0' after 29 ns;
IN3 <= transport "10101010" after 29 ns;
IN2 <= transport "01010101" after 29 ns;
IN1 <= transport "11111111" after 29 ns;
IN0 <= transport "01111111" after 29 ns;

EN <= transport '1' after 31 ns;
SEL1 <= transport '1' after 31 ns;
SEL0 <= transport '1' after 31 ns;
IN3 <= transport "10111111" after 31 ns;
IN2 <= transport "11011111" after 31 ns;
IN1 <= transport "11101111" after 31 ns;
IN0 <= transport "01111111" after 31 ns;

EN <= transport '1' after 34 ns;
SEL1 <= transport '0' after 34 ns;
SEL0 <= transport '1' after 34 ns;
IN3 <= transport "10111111" after 34 ns;
IN2 <= transport "11011111" after 34 ns;
IN1 <= transport "11101111" after 34 ns;
IN0 <= transport "01111111" after 34 ns;

EN <= transport '0' after 36 ns;
SEL1 <= transport '0' after 36 ns;
SEL0 <= transport '0' after 36 ns;
IN3 <= transport "10111111" after 36 ns;
IN2 <= transport "11011111" after 36 ns;
IN1 <= transport "11101111" after 36 ns;
IN0 <= transport "01111111" after 36 ns;

EN <= transport '1' after 39 ns;
SEL1 <= transport '0' after 39 ns;
SEL0 <= transport '0' after 39 ns;
IN3 <= transport "10111111" after 39 ns;
IN2 <= transport "11011111" after 39 ns;
IN1 <= transport "11101111" after 39 ns;
IN0 <= transport "01111111" after 39 ns;

EN <= transport '1' after 41 ns;
SEL1 <= transport '0' after 41 ns;
SEL0 <= transport '0' after 41 ns;
IN3 <= transport "11110111" after 41 ns;
IN2 <= transport "11111101" after 41 ns;
IN1 <= transport "11110111" after 41 ns;
IN0 <= transport "01111111" after 41 ns;

EN <= transport '1' after 44 ns;
SEL1 <= transport '0' after 44 ns;
SEL0 <= transport '0' after 44 ns;
IN3 <= transport "11111011" after 44 ns;
IN2 <= transport "11111101" after 44 ns;
IN1 <= transport "11110111" after 44 ns;
IN0 <= transport "01111111" after 44 ns;

EN <= transport '1' after 46 ns;
SEL1 <= transport '1' after 46 ns;
SEL0 <= transport '1' after 46 ns;
IN3 <= transport "11111110" after 46 ns;
IN2 <= transport "10000000" after 46 ns;
IN1 <= transport "01000000" after 46 ns;
IN0 <= transport "01111111" after 46 ns;

EN <= transport '1' after 49 ns;
SEL1 <= transport '0' after 49 ns;
SEL0 <= transport '1' after 49 ns;
IN3 <= transport "11111110" after 49 ns;
IN2 <= transport "10000000" after 49 ns;
IN1 <= transport "01000000" after 49 ns;
IN0 <= transport "01111111" after 49 ns;

EN <= transport '1' after 51 ns;
SEL1 <= transport '0' after 51 ns;
SEL0 <= transport '1' after 51 ns;
IN3 <= transport "00100000" after 51 ns;
IN2 <= transport "00100000" after 51 ns;
IN1 <= transport "01000000" after 51 ns;
IN0 <= transport "01111111" after 51 ns;

EN <= transport '1' after 54 ns;
SEL1 <= transport '0' after 54 ns;
SEL0 <= transport '1' after 54 ns;
IN3 <= transport "00100000" after 54 ns;
IN2 <= transport "00100000" after 54 ns;
IN1 <= transport "01000000" after 54 ns;
IN0 <= transport "01111111" after 54 ns;

EN <= transport '0' after 56 ns;
SEL1 <= transport '0' after 56 ns;
SEL0 <= transport '0' after 56 ns;
IN3 <= transport "00100000" after 56 ns;
IN2 <= transport "00010000" after 56 ns;
IN1 <= transport "00010000" after 56 ns;
IN0 <= transport "01111111" after 56 ns;

EN <= transport '1' after 59 ns;
SEL1 <= transport '0' after 59 ns;
SEL0 <= transport '0' after 59 ns;
IN3 <= transport "00100000" after 59 ns;
IN2 <= transport "00010000" after 59 ns;
IN1 <= transport "00001000" after 59 ns;
IN0 <= transport "01111111" after 59 ns;

EN <= transport '1' after 61 ns;
SEL1 <= transport '1' after 61 ns;
SEL0 <= transport '1' after 61 ns;
IN3 <= transport "00000100" after 61 ns;
IN2 <= transport "00010000" after 61 ns;
IN1 <= transport "00000010" after 61 ns;
IN0 <= transport "00000001" after 61 ns;

EN <= transport '1' after 64 ns;
SEL1 <= transport '0' after 64 ns;
SEL0 <= transport '0' after 64 ns;
IN3 <= transport "00000100" after 64 ns;
IN2 <= transport "00010000" after 64 ns;
IN1 <= transport "00000010" after 64 ns;
IN0 <= transport "00000001" after 64 ns;

EN <= transport '1' after 66 ns;
SEL1 <= transport '0' after 66 ns;
SEL0 <= transport '0' after 66 ns;
IN3 <= transport "11100011" after 66 ns;
IN2 <= transport "00010000" after 66 ns;
IN1 <= transport "00000010" after 66 ns;
IN0 <= transport "00000001" after 66 ns;

EN <= transport '1' after 69 ns;
SEL1 <= transport '0' after 69 ns;
SEL0 <= transport '0' after 69 ns;
IN3 <= transport "11100011" after 69 ns;
IN2 <= transport "00010000" after 69 ns;
IN1 <= transport "00000010" after 69 ns;
IN0 <= transport "00000001" after 69 ns;

EN <= transport '0' after 71 ns;
SEL1 <= transport '0' after 71 ns;
SEL0 <= transport '0' after 71 ns;
IN3 <= transport "11100011" after 71 ns;
IN2 <= transport "00010000" after 71 ns;
IN1 <= transport "00011100" after 71 ns;
IN0 <= transport "00111000" after 71 ns;
EN <= transport '1' after 74 ns;
SEL1 <= transport '0' after 74 ns;
SEL0 <= transport '0' after 74 ns;
IN3 <= transport "11100011" after 74 ns;
IN2 <= transport "00010000" after 74 ns;
IN1 <= transport "00011100" after 74 ns;
IN0 <= transport "00111000" after 74 ns;

EN <= transport '1' after 76 ns;
SEL1 <= transport '1' after 76 ns;
SEL0 <= transport '1' after 76 ns;
IN3 <= transport "11000111" after 76 ns;
IN2 <= transport "00010000" after 76 ns;
IN1 <= transport "11000011" after 76 ns;
IN0 <= transport "00111100" after 76 ns;

EN <= transport '1' after 79 ns;
SEL1 <= transport '0' after 79 ns;
SEL0 <= transport '0' after 79 ns;
IN3 <= transport "11000111" after 79 ns;
IN2 <= transport "00010000" after 79 ns;
IN1 <= transport "11000011" after 79 ns;
IN0 <= transport "00111100" after 79 ns;

wait;

end process;
end BEHAVIOR;

Figure 42. Test Bench for TWOSTMUX
Figure 43. TWOSTMUX Test Paths
Figure 44. TWOSTMUX Test Paths after 79 NS of Simulation
Figure 45. TWOSTMUX after 0 NS of Simulation
Figure 46. TWOSTMUX after 6 NS of Simulation
Figure 47. TWOSTMUX after 9 NS of Simulation
Figure 48. TWOSTMUX after 16 NS of Simulation
Figure 49. TWOSTMUX after 31 NS of Simulation
Figure 50. TWOSTMUX after 79 NS of Simulation
Figure 51. History of the IN0 Signal
end process MUX_2to1_OUT_4;

-- Process Name: MUX_2to1_V2

MUX_2to1_V2_11: process (EN, SEL0, IN3, IN2)
begin
  if EN='1' then
    if SEL0='0' then
      Q1 <= IN2;
    else
      Q1 <= IN3;
    end if;
  end if;
end process MUX_2to1_V2_11;

-- Process Name: MUX_2to1_V

MUX_2to1_V_18: process (EN, SEL0, IN1, IN0)
begin
  if EN='1' then
    if SEL0='0' then
      Q0 <= IN0;
    else
      Q0 <= IN1;
    end if;
  end if;
end process MUX_2to1_V_18;

Figure 52. Coverage Window for TWOSTMUX Model
Figure 53. Waveform Window for TWOSTMUX Model
Chapter 9. Conclusions and Future Work

9.1. Creating Synthesizable Code

In order for synthesis software to use VHDL code, the designer must make certain precautions. For example, the code generated by the Modeler's Assistant uses a sensitivity list for all processes. Sensitivity lists are not allowed in the Synopsys Synthesis tool. If "wait" statements were generated instead, this part of the code could be used for synthesis. The Modeler's Assistant could be modified so that the parts of the VHDL code it automatically generates are synthesizable.

9.2. Maintenance of Procedure and Type Library

The signal types used by the Modeler's Assistant are currently stored in the files VHDLCAD.vhd and USER_TYPES.vhd. The Modeler's Assistant scans these files on startup to determine the user defined types. The current method of adding a new user defined type requires that the user edit one of these files to add the new types. An interface between these files and the Modeler's Assistant could be made so that the user can add new types or procedures in some sort of easy manner.
9.3. Test Generation for Supernodes

One of the limitations of the Hierarchical Behavioral Test Generator is its inability to create tests for supernodes. This limits the areas of the simulation environment which relate to test path verification. The simulation environment has been programmed to support test paths through behavioral supernode models once this development is complete.

9.4. Combining the Test Generation Programs

Currently, the test generation programs are all stand-alone programs. It would be advantageous if the HBTG could call the PTG program for all processes in the PMG which are missing process level tests. This would save the user from having to use the PTG interface for each process of the PMG and allow the entire test bench to be created by the selection of the Test Bench menu item.

9.5. User Help System

In order to make learning the tool easier, a help system for the tool could be built into the menu system so that the user could click on a help selection and then click on any command to receive help for that command. This would simply require the creation of a help file for each command, adding one more selection to the menus to turn on help mode, and the modification of the main menu control function to take appropriate action in help mode.

9.6 Conclusions

The Modeler's Assistant has evolved into not only a fast design tool, but a test generation and design verification tool also. The tool now allows the user to create tests
for a model without leaving the tool. Also, the user can simulate and verify the design against the PMG. The user can step forward and backward through the simulation, verifying the functionality of the model, verifying the test paths, checking the coverage, viewing the waveforms, etc. until the design is satisfactory. The data structure used in the design of the system allows this to be done easily. This structure is fitting for the algorithm used in moving through the simulation, is dynamic in size, and is flexible enough to support future enhancements or utilization. Its design also allows it to interact well with the existing PMG database.

The combination of the design, test generation, and verification environments allows the user to have a single, integrated development environment based around the Process Model Graph as the sole development view. This environment has little management overhead caused by interfacing to other programs and therefore dramatically decreases development time. Other enhancements to make the tool easier to use and more flexible have been added. Overall, the Modeler's Assistant has significantly improved on its goal of allowing VHDL behavioral models to be developed more rapidly. Future work on the Modeler's Assistant and its relatives should help in increasing the usefulness of the tool.
Bibliography


Appendix A: 
Guide for Simulation and Test Evaluation with the Modeler’s Assistant

Introduction:

The previous version of the Modeler's Assistant provided a tool for VHDL model development using the Process Model Graph (PMG) as the basic representation for CAD. The Simulation and Test Analysis additions to the Modeler's Assistant extend the usage of the tool and PMG from the design domain into the test generation and simulation evaluation domains. This guide describes how to use the simulation environment and test generation interface portions of the Modeler's Assistant. An overview of the design and test process is shown in Figure 54. It assumes a basic knowledge of the other existing features of the Modeler's Assistant as described in [1]. The steps for test generation and simulation are as follows:

(1) Creation of processes needed for the model.
(2) Creation of the Process Model Graph (PMG)
(3) Analysis of processes using both Synopsys and Vtip Analyzers
(4) Invocation of PTG process test generator.
(5) Analysis of PMG with the Synopsys Analyzer
(6) Invocation of the HBTG and TBG tools for test bench generation
(7) Analysis of the test bench with the Synopsys analyzer
(8) Invocation of the simulator to produce simulation results
(9) Addition of monitors on ports and signals
(10) Use of visualization features and displays
Figure 54. Model Testing and Simulation using the Modeler's Assistant

Section 1 of this guide will describe the development of and test generation for processes. Section 2 will discuss the creation of a test bench by calling the Hierarchical Behavioral Test Generator and the Test Bench Generator from within the Modeler's Assistant. Section 3 discusses all of the various steps in using the Simulator Environment in the debugging process. Section 4 describes other features that supplement the earlier version described in [1].
Section 1 - Development and Test Generation for Processes

1.1 Process Development with Test Generation Considerations

If the test generation interfaces of the Modeler's Assistant are to be used, the constraints of the Test Generation programs should be considered [2]. This includes using only delta delay within the VHDL code, limiting the fanout from an output port, and using less than 3 sensitive input ports. Designing a PMG within these limitations will allow the user to create tests for the models easily and quickly.

1.2 Interfacing to the Test Generation Software

In order to create a Test Bench, the user must first create the process level tests for each process. Creation of process tests should be done after the creation of the entire PMG unit since the test generation software requires the PMG unit information in the test creation process. Once this has been done, process level tests can be easily created from within the Process Edit menu.

Before calling the Process Test Generator (PTG), the user must first analyze the process using the Synopsys and Vtip Analyzers [3,4]. This is done by clicking the mouse on the Analyze Process and Vtip Analyze menu selections. For each selection, the user is prompted for the name of the VHDL output file (usually default), the unit name of the process (if not using the Process Edit selection from the Unit menu), and the input module name (usually same as output file, default). A window then appears showing the analysis results for that particular analyzer.

After analysis is done successfully, the user calls the PTG software by clicking the Call PTG menu selection. After entering the unit name for that process, a window will appear, prompting the user for the needed information [5]. Once complete, the PTG
program will have created the necessary ".tst" file for that process. These files hold the process level test information for each process.

Section 2 - Creating a Test Bench

Once a ".tst" file has been created for each process in the PMG, the Hierarchical Behavioral Test Generator (HBTG) and Test Bench Generator (TBG) programs can be used to create a test bench for the entire unit [5]. This is done by selecting the Test Bench menu selection under the UNIT menu. This will then prompt the user for the unit name (for saving). The HBTG program is then called, which takes the ".tst" files for each process and creates a ".test" file for the entire unit. This program requires no user information. The TBG program is then automatically called. The user is then prompted for an initial value for any symbolic bit vectors used in creation of the test bench. This will create a test bench with the name "unit_TB.VHD".

After the test bench has been successfully created, the user can view the new test bench by selecting the Show TB menu selection. The test bench code should work properly as generated, but the user can edit the code using the Edit TB menu selection. If this selection is used before a test bench has been generated, the user will be asked if he or she wishes to create their own test bench. If the user replies yes, then all of the test bench code except the code between the "begin" and "end process" lines will be generated. The edit window will then appear allowing the user to create their own test bench.

Once the test bench is verified, the next step is to analyze the new test bench. It must be noted, however, that it is first necessary to analyze the unit model before analyzing the test bench. This is done using the Analyze menu selection. Obviously, it is a good practice to have already done this before the creation of the test bench. If the model has been analyzed, the test bench can then be analyzed using the Analyze TB menu.
selection. The user is prompted for the name of the test bench to analyze. If TBG was used to generate the test bench, the default value will be correct. Once this is successful, the user is ready to simulate.

**Section 3 - Using the Simulation Environment**

The Simulation Environment is started by clicking on the **Simulate** menu selection in the **UNIT** menu. The user is then prompted for the name of the test bench entity and the name of the component name of the model. The default entity name is the capitalized entity name with "_TEST_BENCH" appended. This is the default entity name produced by TBG. The default component name is R1, which is also the TBG default. If the user is using a self-made test bench, these labels should match those used in the test bench. The simulator is then automatically called to get the simulation results for the entire model.

After the simulation is complete and the user closes the popup simulation window, the PMG is shown along with the **SIMULATE** menu, as in Figure 55. The current simulation time is also shown at the bottom of the window. The first step in the simulation process is to add monitors to the simulation. To add a single monitor to the display, the user selects the **Add Monitor** menu selection. The user is then asked to select the ports, variables, and signals to monitor. To add a port or variable, the user selects that port or variable. To select a signal, the user selects one of the ports connected to that signal. The user repeats this process until all desired ports and signals are monitored, then clicks on empty space to finish adding monitors. If the user wishes to monitor all ports, variables, and signals within the PMG, he or she may select **Add All** from the menu. A similar procedure is used to hide monitors by using the **Hide Monitor** and **Hide All** menu selections.
Figure 55. The Simulation Environment
When a monitor is added to the display environment, the representation of that port or signal is changed to show that the signal is being monitored. All signals and ports that have not had an event past time zero are shown in italics. The signal lines for this type of signal are shown as dashed lines. This type of representation easily allows the user to visualize the propagation of tests and events through the model. Ports and signals that are monitored but have had an event past time zero are shown in the regular font and solid signal lines. If the port or signal just became tested (had its first event) at the current simulation time, the label for that port or signal is surrounded by brackets. All monitored ports and signals also show two additional fields besides the name. The first field following the signal or port name represents the number of events that have occurred on that signal or port from the start of simulation through the current simulation time. Note that events at time zero count as events in the event count, but are not considered tests, so the lines and fonts at time zero remain solid. The last field shows the value of the port or signal at the current simulation time. Ports and signals that have not received an event will show the default value for that type.

After all monitors have been added by the user, the user can then step forward and backward through the simulation times to visualize the event and test propagation through the circuit. To step forward to the next event time, the user can select **Step Fwd** on the menu. To step backward to the previous event time, the user can select **Step Back** on the menu. If the user selects **Goto Time** on the menu, the user will be prompted for a simulation time. Simulation time will be moved to the closest event time greater than or equal to the time entered.
Figure 56. Display of Test Paths
By stepping through the simulation, the user visualizes the movement of events through the model. This movement can be associated with the behavioral level testing paths by using the **Show Paths** menu selection. After prompting the user for the test file name (which is the unit name with ".test" appended by default) the PMG display changes again, showing the test paths through the model (Figure 56). The test paths are labeled by number corresponding to the test path number in the ".test" file. After switching to this view, the user can see where in the model the test paths lie by following the ports and signals by number from each sensitive input port to each output port. The user can switch back to the simulation view by using the **Hide Paths** menu selection. By switching to and from the path view, the user can visually verify that the model is being exercised as desired. The user is also able to step forward and backward through the simulation in this view. As in the normal simulation view, tested ports and signals are distinguished from untested ports and signals by solid lines and solid fonts. This easy visual verification is essential to the rapid design and verification of VHDL models in a consistent design and test environment.

If the user wishes to save monitors for later retrieval, he or she may use the **Save Mons** menu selection. The user is then prompted for the name of the file in which to save the monitors. These monitors may be restored by providing the file name after clicking on the **Rest Mons** menu selection. Since the monitors depend on the PMG, the monitors may only be restored if no structural changes have been made to the PMG (e.g. add or delete a signal).

Another useful feature of the simulation environment is the ability to see all the events of a particular port or signal and the time of each. This is done by selecting the **Show Hist** menu selection. The user is then asked to select a port or the port of a signal. After selection, a window will appear showing the port or signal name followed by a list of
events (Figure 57). Each event shows the value of the port or signal followed by the time at which the event occurred. The user may then verify that all planned events occurred on a particular port or signal.

One of the main goals of the test generation algorithms is to give a "coverage" of the VHDL model, i.e. to execute each line of the VHDL code. The coverage of the VHDL model can be shown to the user by clicking the Coverage menu selection. This calls the Synopsys Coverage program to get coverage results along side the VHDL model and displays the results in a window (Figure 58).

Another interface added to ease the verification of a model and its simulation is an interface to the Synopsys Waveform Viewer. By selecting the Call Waves menu selection, the user can popup a display window showing all of the ports and signals for the unit and their corresponding waveform.

After the model is verified, the user can then select Backup from the menu to return to the UNIT menu. If the user then selects Simulate again, the simulation environment will remember the previous monitors and simulation time as it was before selecting Backup. If the user wishes to restart the environment, including monitor selection, he or she can select the Reset Sim selection from the UNIT menu. Then, when the user selects Simulate, the simulation environment is reset and the simulator is called again.
### Figure 57. Port and Signal History Window

<table>
<thead>
<tr>
<th>Name: ENBLD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time = 14 ns</td>
</tr>
<tr>
<td>Time = 19 ns</td>
</tr>
<tr>
<td>Time = 24 ns</td>
</tr>
</tbody>
</table>
architectural Behavioral of latch8 is

begin

-- Process Name: OUTPUT

OUTPUT.4: process (ENBLD, REG)
begin
if (ENBLD = '1') then
  DO <= BV_to_MVL(REG);
else
  DO <= "11111111";
end if;
end process OUTPUT.4;

-- Process Name: ENABLE

ENABLE.9: process (NDS2, DS1)
begin
  ENBLD <= DS1 and not NDS2;
end process ENABLE.9;

-- Process Name: LATCH

LATCH.14: process (CLK)
variable VAR: BIT_VECTOR(0 to 7);
begin
  if (CLK = '0') then
  REG <= DATA;
  else
  VAR := DATA;
  end if;
  REG <= VAR;
end if;

Figure 58. Coverage Results
Section 4 - Other New Features of the Modeler's Assistant

In addition to the Simulation and Test Verification environment, other improvements have been made to the previous version. This includes allowing interconnecting signals to be made entity ports and other convenience changes to make the tool more useful to modeling needs.

The first of the features is the *External* selection from the *CHANGE* submenu off of the *UNIT* menu. This allows the user to tag a port of a signal to make the declaration of the signal occur in the entity declaration of the VHDL code rather than in the architecture section. This is especially useful for extending the capabilities of the Test Bench Generator to include tests for models with feedback. The signal can then be used by self made test benches, structural models, etc. After clicking on this selection, the user is then asked to select the port of the signal to make external. Once a port is selected, it is marked external by the display of a second circle around the port. The declaration of this signal will now occur in the entity declaration of the model in the same way unattached ports do. However, since these signals are both driven and read, they are always of type "inout."

The external signal feature also applies to signals within supernodes. When a signal within a supernode is marked as external, and the supernode is collapsed, a new port appears on the stippled circle for that supernode. This port is also marked as external and other signals may be connected to it from the rest of the model.

The next new feature that affects usage of the tool is the *English Specify* selection from the *PROCESS* menu. This provides an interface into Meenakshi Manek's Natural Language Specification tool to allow users to write a simple English description of the process and create the process code. Usage of this tool is described in [5].
Other bugs and limitations were also fixed from the previous version. Some of the more notable are summarized below. Refer to [1] for an explanation of the previous version.

1. The method of renaming a signal changed to rename all signals connected to the same port.
2. Fixed the limitation on the number of variable length process ports.
3. Parameters within primitive code can now be used with variables also.
4. Vectors of equal length but different ranges can now be connected by signals.
5. Added #portname.Z and #portname.X primitive parameters to allow variable length strings of "Z's" and "X's" to assign to signals or variables in the primitive code.
6. For convenience, all moving of signals, ports, etc. is repetitive until blank space is chosen.
7. Confirmation is now sought before overwriting existing units.
8. Changing a port, variable, or signal now scans the code and replaces the name.
9. Bit vectors can now be split without using the splitter primitives. In earlier versions, there was no way of connecting a bit or subrange of a vector type to a smaller sized vector. In the new version, when a user tries to connect ports with the same base type but different sized ranges, the user is prompted for the subrange of the vector signal to use for the smaller port. This range is then used when replacing the port name in the process while generating the VHDL code for an entity. Note that this will only work if all of the subrange fanout for a signal occurs from a single process port. Subranges of a signal which already has a subrange is not allowed. The user is prompted for the beginning and ending
numbers for the vector subrange. If the user specifies the same number for both range endpoints, then the single bit indexed by that number is used.
Appendix B: Programmer's Guide

Introduction:

This guide is to be used as a supplement to the guide provided in [1]. This guide will explain the additions to the data structure of the Modeler's Assistant and will explain the programming and data structures used in creating the simulation environment and test generation interface portions. The Modeler's Assistant is written in C using the Xt Toolkit and X-Windows Athena widget set. It runs under and Unix operating system that supports the widget set and the standard C libraries. The main additions to the code for the creation of the simulation environment were made into the file sim.c. The code for the display of the graphics and parts of the code for the test generation interface was added to the various files described in [1]. The following describes the changes and new features of these files:

menus.c
This contains the callback for menu selections and the resource specification for creating menus. It was modified to add the new menu selections and to create the new "Simulate" menu.

misc.c
This file contains functions for code generation, calling the analyzer, and other functions. It was heavily modified to remove bugs, allow for external signals, and to control external signals within supernodes. Functions were added to interface with the test generation tools, analyzers, and to the "English Process Specification" program [5].

modasTop.c
This file contains all the setup of widgets and drawing functions. It contains the function main and is the entry point into XtAppMainLoop which is the function which waits for events from the X-Windows server and calls appropriate functions. The main changes to this file were the addition of the new global variables used in the simulation environment.
**modules.c**
This is the code for manipulating processes. It also contains the AddSignal function as well as new functions for external signals. It was modified to allow for creating external signals, repetitive moving of processes, and allowing multiple copies of modas to run concurrently.

**ports.c**
This is the code for the manipulation of ports. It was modified to scan the process code when renaming a port, to make moving a port repetitive, and to show the graphics for an untested port in simulation mode.

**primitives.c**
This file contains the code for creation and instantiation of primitives. It was modified to remove some minor bugs, sort the "Add Primitive" menu, allow parameterized variables as well as ports, and to allow for the new ".X" and ".Z" parameters.

**signals.c**
This file contains functions for showing or manipulating signals. This file was changed extensively to allow for the external signal feature and to display the graphics for signals in the simulation mode.

**statement.c**
This contains the *MakeSignal, MakeDest, and MakeSource* functions used in the creation of signals. Only the *MakeSignal* function was changed to allow for the new external signal feature.

**supernodes.c**
This function contains the code for manipulating supernodes. It was modified to remove some bugs in supernode manipulation and to allow for the use of external ports in supernodes.

**units.c**
This file has the callbacks for the menu selections and unit load functions. Some minor bug fixes and convenience changes were made. The only significant change was to the menu structure to allow for new features.

**variables.c**
This file contains functions for manipulating variables and constants. The first change is the code that scans the process code when changing a variable or constant name and replaces the old name. The other change is the code that makes moving a variable or constant repetitive until the user clicks on blank space.
No changes were made to the following files:

`vcad.c`, `generics.c`, `expr.c`

The header files `externs.h`, `vhdl.h`, and `macros.h` were modified for the new variables, functions, and macros. The `sn.h` and `xinclud.h` header files were not modified. The icon `vcad.icon` was slightly modified to include a reference to Virginia Tech in the drawing.

**Section 1 - Overview of the Test Generation Interface**

As stated earlier, the Modeler's Assistant now provides an integrated design, test generation, and simulation environment. The interface with the test generation environment is based on the model shown in Figure 59.

![Figure 59. Overview of the Test Generation Interface](image-url)

Programs:
- Synopsys Analyzer
- Vtip Analyzer
- mvhd
- ptg
- hbtg
- tbg
Each of the programs listed in Figure 59 are called by using the filename assigned to various environment variables. If the environment variable is not set, a default filename is used in calling of the programs. The environment variables, the programs called by those variables, a short description of the programs, and the default program names are given in Table 1.

The programs used in the test generation process both require and generate various intermediate files. These programs are called from various points within the Modeler's Assistant menu system. The programs used in test generation, their required input files, their generated output files, and the menu selections that cause the output generation are listed in Table 2.
<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Program Name</th>
<th>Description</th>
<th>Default Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>VCADSIMULATOR</td>
<td>vhdsim</td>
<td>Synopsys Simulator</td>
<td>(no default - must be set)</td>
</tr>
<tr>
<td>VCADANALYZER</td>
<td>vhdlan</td>
<td>Synopsys Analyzer - Analyzes VHDL files</td>
<td>(no default - must be set)</td>
</tr>
<tr>
<td>VCADHBTG</td>
<td>hbtg</td>
<td>Heirarchical Behavioral Test Generator-Generates tests for entire unit</td>
<td>/newhbtg</td>
</tr>
<tr>
<td>VCADSHELL</td>
<td>Any command shell</td>
<td>Shell to execute programs</td>
<td>/usr/openwin/demo/xterm</td>
</tr>
<tr>
<td>VCADTBG</td>
<td>tbg</td>
<td>Test Bench Generator - Generates test bench file from heirarchical test file</td>
<td>/tbg</td>
</tr>
<tr>
<td>VCADPROCESS</td>
<td>process</td>
<td>English Process Specification - Allows writing a &quot;pseudo English&quot; process description</td>
<td>/process</td>
</tr>
<tr>
<td>VCADPRIMDIR</td>
<td>(directory name)</td>
<td>Path to where process primitives are stored</td>
<td>(current directory)</td>
</tr>
<tr>
<td>VCADWAVES</td>
<td>waves</td>
<td>Synopsys Waveform Viewer</td>
<td>(no default - must be set)</td>
</tr>
<tr>
<td>VCADPTG</td>
<td>ptg</td>
<td>Process Test Generator- creates process tests</td>
<td>/newptg</td>
</tr>
<tr>
<td>VCDMVHD</td>
<td>mvhd</td>
<td>Makes process VHDL Entity for ptg</td>
<td>/mvhd</td>
</tr>
<tr>
<td>VCADEVTIP</td>
<td>vhdl</td>
<td>Vtip Analyzer - needed by ptg</td>
<td>(no default - must be set)</td>
</tr>
<tr>
<td>VCADECOVERAGE</td>
<td>coverage</td>
<td>Synopsys coverage tool to generate VHDL coverage file</td>
<td>coverage</td>
</tr>
</tbody>
</table>
### Table 2: Intermediate Files Created and when They Are Generated

<table>
<thead>
<tr>
<th>Program</th>
<th>Input Files</th>
<th>Output Files</th>
<th>Menu Selection</th>
</tr>
</thead>
<tbody>
<tr>
<td>modas</td>
<td>(none)</td>
<td>process.mod unit.unt</td>
<td>PROCESS-&gt;Save UNIT-&gt;Save</td>
</tr>
<tr>
<td>mvhd</td>
<td>unit.vhd process.vhd</td>
<td>process.vhd (modified)</td>
<td>PROCESS-&gt;Analyze PROCESS-&gt;Vtip Analyze</td>
</tr>
<tr>
<td>ptg</td>
<td>process.vhd process.mod unit.unt</td>
<td>process.tst</td>
<td>PROCESS-&gt;Call PTG</td>
</tr>
<tr>
<td>hbfg</td>
<td>unit.unt processes.test</td>
<td>unit.test</td>
<td>UNIT-&gt;Test Bench</td>
</tr>
<tr>
<td>tbg</td>
<td>unit.test</td>
<td>unit_TB.VHD</td>
<td>UNIT-&gt;Test Bench</td>
</tr>
</tbody>
</table>

One thing to note is that the ptg and tbg programs are the only ones which require direct input from the user. All other programs receive their input from files which can be passed as command line parameters by the Modeler's Assistant. Since the ptg and tbg programs need direct input from the user, they are executed within the X-Windows shell specified by the environment variable **VCADSHELL**.
Section 2 - Data Structure Used by the Modeler's Assistant Simulation Environment

As mentioned in [1], the Modeler's Assistant uses a statically declared linked-list structure. This linked list is implemented by indexing a large array of elements called "Nodes", or objects. Each object in the structure contains information on the name, type, and location of the object. In addition, each object contains six pointers which are either the index of another object in the array structure or some sort of information about that object. This method is used to form the linked list structure (Figure 60). Each pointer of the object was given a macro to allow for an easier representation of the structure and for easy traversal between objects. Those pointer references are given in Table 3.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit (12)</td>
<td>TopModRef</td>
<td>TopSignal</td>
<td></td>
<td></td>
<td>TopSuperNodeRef</td>
<td></td>
</tr>
<tr>
<td>Module (1)</td>
<td>TopPort</td>
<td>TopVariable</td>
<td>TopGeneric</td>
<td></td>
<td></td>
<td>TopConstant</td>
</tr>
<tr>
<td>ModuleRef (14)</td>
<td>TopOfModule</td>
<td>NextModRef</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Port (2)</td>
<td>NextPort</td>
<td>PortType</td>
<td>PortSignal</td>
<td>PortStatus</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable (4)</td>
<td>NextVariable</td>
<td>VariableType</td>
<td>VariableStatus</td>
<td>VarAggString</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generic (3)</td>
<td>NextGeneric</td>
<td>GenericType</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Constant (32)</td>
<td>NextConstant</td>
<td>ConstantType</td>
<td></td>
<td></td>
<td>ConAggString</td>
<td></td>
</tr>
<tr>
<td>Signal (13)</td>
<td>NextSignal</td>
<td>SignalSrc</td>
<td>SignalDst</td>
<td>SignalStatus</td>
<td>Subrange</td>
<td></td>
</tr>
<tr>
<td>SrcNode (15)</td>
<td>PortModRef</td>
<td>AbsPortLoc</td>
<td>RelPortLoc</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DstNode (16)</td>
<td>PortModRef</td>
<td>AbsPortLoc</td>
<td>RelPortLoc</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SuperNodeRef (38)</td>
<td>SuperNodeOfRef</td>
<td>NextSuperNodeRef</td>
<td></td>
<td></td>
<td>TopPortMap</td>
<td></td>
</tr>
<tr>
<td>SuperNode (37)</td>
<td>TopModRef</td>
<td>TopSignal</td>
<td>TopSNPort</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SuperNodePort (39)</td>
<td>NextPort</td>
<td>PortType</td>
<td>PortSignal</td>
<td>PortStatus</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PortMap (40)</td>
<td>NextPortMap</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Figure 60. Structure of the PMG Linked List Database
The main changes to this structure was the use of unused bits of the status references to store new information about the PMG and simulation monitors. As shown in the table, the SignalStatus macro was added to the signal object. Bit 0 of the pointer is set if the signal is an external signal. Bit 5 of the pointer is set if the signal is to be monitored in simulation mode. Bit 5 of the PortStatus pointer (pointer 3) of ports, supernode ports, and variables is also used to mark a port or supernode port as monitored. This pointer is new for variables and is only used for setting a monitor. Bit 4 of the PortStatus pointer for supernode ports is also set if that supernode port represents an external signal within the supernode. Pointer 2 of variables is used in simulation mode to hold the parent module number so that the appropriate simulation monitor can be created.

Pointer 4 of a signal node is used to hold a subrange. This is used when connecting vectors of different sizes. The subrange holds the range used when the smaller vector is replaced in the specified VHDL code. The end of the range is in the least significant five bits of this integer. The beginning of the range is in the next higher five bits (bits 9-5). The above pointer changes are the only changes to the existing PMG data structure.

2.2 Structure for the Simulation Environment

In order to hold events for simulation, another data structure was added to the Modeler’s Assistant to hold simulation results. Their are two main parts to this structure, a statically allocated array of monitors and a dynamically allocated quadruple linked list of events and time values. An overview of this structure is given in Figure 61. Each square box in the figure represents a C structure. The structures used in Figure 61 are given in Figure 62. As stated earlier, the linked list is a dynamic structure, elements of the structure do not exist unless allocated during execution. The smon structure is also
dynamic, it contains a variable number of elements, each called a smon_node, depending on the number of signals monitored during simulation, but it is an array allocated on entry into the simulation mode. Therefore, the smon structure is indexed by an integer offset rather than a C pointer.

The smon_node structure contains four fields. The first field, nodenum, is the index of the monitored port or signal in the PMG data structure. This integer can be used to find the other information about the monitored port or signal, such as the name, stored in the array, e.g. "Node[nodenum].name". The second field, event_time contains the time of the first event on that monitor. The third and fourth fields, listhead and listtail, contain pointers to the first element and last element, respectively, of the event list for the monitored port or signal.

The quadruple linked list structure is actually made of a quadruple linked list of events with a double linked list of time values pointing into it. Each element in the quadruple linked list is of type listelm. This C structure is made of six fields. The first field, prev_time is a pointer to the listelm that represents the previous event on the same port or signal. The next_time field is a pointer to the listelm that represents the next event on the port or signal. The prev_sig field is a pointer to the listelm that represents an event on the previous monitored port or signal which had an event at the current time. The next_sig field is a pointer to the listelm that represents an event on the next monitored port or signal which had an event at the current time. If any of these references doesn't exist (at the end of the list in that direction), then this pointer is set to NULL. The fifth field in the listelm data structure is nodenum. This is the node number in the PMG database for the monitored port or signal (same as the one in smon). The time field contains the time of the event in nanoseconds. The last field, value, is the string representation of the value of the event on the port or signal.
Figure 61. Overview of the Simulation Environment Data Structure
typedef struct l_elm *l_elm_ptr;
typedef struct l_elm {
    int nodenum;
    int time;
    char value[MAX_LABEL]; /* allow 32 character signal value */
    l_elm_ptr next_time;
    l_elm_ptr prev_time;
    l_elm_ptr next_sig;
    l_elm_ptr prev_sig;
} listelm;

typedef struct {
    int nodenum;
    int event_time;
    l_elm_ptr listhead;
    l_elm_ptr listtail;
} smon_node;

typedef struct tlist *time_list_ptr;

typedef struct tlist {
    int time;
    l_elm_ptr listhead;
    l_elm_ptr listtail;
    time_list_ptr next_time;
    time_list_ptr prev_time;
} time_list_node;

typedef struct {
    int pathnum;
    char name[MAX_LABEL];
} sens(elm;

**Figure 62. C Structures Used in the Simulation Data Structure**

Each element in the double linked list of time values is of type `time_list_node`. There are two global variables which are pointers to the head and tail of this list, `time_list_head` and `time_list_tail`. There is also a global variable, `current_sim_time`, which points to the element of the list representing the current simulation time. The C
structure for the list is made of five fields. The **time** field is the integer value, in nanoseconds, of the event time that this element represents. The **sig_list_head** field is a pointer to the first event in the list of events (of type listelm) which occur at this time. The **sig_list_tail** field is the pointer to the end of this list. The **prev_time** and **next_time** fields are pointers to the previous time list element and next time list element, respectively.

During simulation mode, these list structures are accessed using the standard C methods of accessing structures. Macros were added to the macros.h file to allow for checking if a port or signal is monitored (Monitored(nodename)) and for determining the name of a monitor (sname(smonptr)).

There are also three other minor data structures used in the simulation environment. The first is for holding the generated sensitive test paths through the model for viewing when the **Show Paths** menu selection is chosen. This is a simple static array consisting of elements with two fields, **pathnum** and **num**. Each element represents a port or signal along a sensitive test path. **Pathnum** is the number of the path that contains node number **num**. When a port or signal is displayed in the path view, this array is searched for all path numbers of which the port node or signal source node is a part. This provides a simple way of storing the sensitive test paths for viewing.

The other simple data structures are the structures that track the value of an event and the number of events on a particular port or signal at a given time in simulation mode. The **event_values** structure is a dynamically allocated array of string pointers that points to the event value at the current simulation time. This structure is allocated the same number of elements as in the PMG database. It therefore provides an easy mapping from the PMG node number of a port or signal to its current value during simulation (e.g. the value of the signal which has node number 123 is pointed to by event_values[123] ). These pointers are updated as the user steps through the simulation to point to the correct
value. The `event_count` structure is similar to `event_values`, except that instead of a
dynamic array of pointers, it is a dynamic array of integers that hold the number of events
that have occurred on a port or signal. This count is incremented and decremented as the
user steps through the simulation.

Section 3 - Files Used by the Simulation Environment

Besides temporary files and files produced by the test generation tools, there are
other files produced in the simulation portion of the Modeler's Assistant. The first of these
is the control file used in simulation. An example control file for the latch8 model is given
in Figure 63. The first lines in the control file specify to the simulator to use the `sim.out`
file as the output file for events. The coverage line tells the simulator to generate the
coverage results of simulation. The next lines tell the simulation to put event monitors on
the signals and ports of the model. The "trace" lines tell the simulator to produce the
waveform output that can be used with the Show Waves menu selection. The final two
lines tell the simulator to run the simulation till completion then quit. When the simulator
runs, the `sim.out` file is generated. After simulation, this file is then parsed and stored in
the event data structure described earlier. A `sim.ow` file is also generated by the simulator
by passing the "-i sim.ow" parameter to the simulator. This file contains the information
needed by the waveform viewer.
cd
open sim.out
logtime -e sim.out
coverage
monitor -o sim.out event LATCH8_TEST_BENCH/R1/ENBLD
trace LATCH8_TEST_BENCH/R1/ENBLD
monitor -o sim.out event LATCH8_TEST_BENCH/R1/REG
trace LATCH8_TEST_BENCH/R1/REG
monitor -o sim.out event LATCH8_TEST_BENCH/DO
trace LATCH8_TEST_BENCH/DO
monitor -o sim.out event LATCH8_TEST_BENCH/NDS2
trace LATCH8_TEST_BENCH/NDS2
monitor -o sim.out event LATCH8_TEST_BENCH/DS1
trace LATCH8_TEST_BENCH/DS1
monitor -o sim.out event LATCH8_TEST_BENCH/CLK
trace LATCH8_TEST_BENCH/CLK
monitor -o sim.out event LATCH8_TEST_BENCH/DATA
trace LATCH8_TEST_BENCH/DATA
monitor -o sim.out write LATCH8_TEST_BENCH/R1/LATCH_14/VAR
trace LATCH8_TEST_BENCH/R1/LATCH_14/VAR
run
quit

Figure 63. Example Simulation Control File

After the user has added all desired monitors, he or she may wish to save the monitors for later retrieval. This monitor is stored in a file with a ".mon" extension. This file simply stores the node numbers of all monitored ports and signals, one per line. This is sufficient since changing the PMG, and thus the functionality of the model, makes the old monitors inappropriate. The saved node numbers can be later read and the monitors for the nodes enabled.

The last file created by the simulation environment is the sim.wav file that stores the ports and signals to load into the Synopsys Waveform Viewer. This file is loaded by the Waveform Viewer automatically when the sim.aw file is passed to the viewer with the
"-wif sim.ow" parameter. This lists monitors in a fashion similar to the sim.con file. All signals which are monitored in the model are included in this file. An example sim.wav file is shown in Figure 64.

| Wave LATCH8_TEST_BENCH/R1/REG[0:7] |
| Wave LATCH8_TEST_BENCH/R1/ENBLD    |
| Wave LATCH8_TEST_BENCH/DO[0:7]     |
| Wave LATCH8_TEST_BENCH/NDS2        |
| Wave LATCH8_TEST_BENCH/DS1         |
| Wave LATCH8_TEST_BENCH/CLK         |
| Wave LATCH8_TEST_BENCH/DATA[0:7]   |
| Wave LATCH8_TEST_BENCH/R1/LATCH_14/VAR[0:7] |

Figure 64. Sim.wav File for Latch8 Model
Section 4- Compilation Procedure

The following Makefile will compile the new version of the Modeler's Assistant:

\[\text{OBJS = supernodes.o primitives.o statement.o modules.o expr.o generics.o ports.o signals.o variables.o modasTop.o vcad.o menus.o units.o misc.o sim.o}\]

\[\text{CFLAGS = -g -sb}\]

\[\text{a.out: $(OBJS)}\]
\[\text{cc -g -sb -L/usr/lib -L/usr/openwin/lib -L/home/synopsys/sparc/X11/usr/lib $(OBJS)}\]
\[\text{-Bstatic -lXaw -lXmu -lXext -lX1 -lX11 -lm -o modas}\]

\[\text{$(OBJS)$: xinclude.h vhdl.h externs.h macros.}\]

The above Makefile will produce the code and symbol tables needed to use the Sun Debugger and Source Browser. The executable would be smaller without the -g option.
VITA


David M. Dailey Jr.