A Parametrized CAD Tool for VHDL Model Development with X Windows

by

Balraj Singh

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

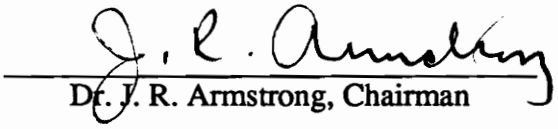
in partial fulfillment of the requirements for the degree of

Master of Science

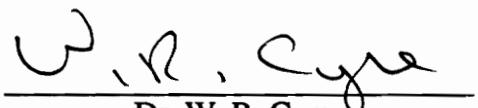
in

Electrical Engineering

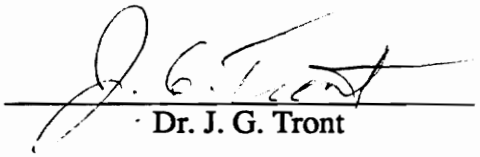
APPROVED



Dr. J. R. Armstrong, Chairman



Dr. W. R. Cyre



Dr. J. G. Tront

July, 1990

Blacksburg, Virginia

A Parametrized CAD Tool for VHDL Model Development with X Windows

by

Balraj Singh

Dr. James R. Armstrong, Chairman

Electrical Engineering

(ABSTRACT)

This thesis describes the development of a graphical CAD tool for VHDL model development. The tool was developed for the X Windows environment. The graphical representation used is the *process model graph* [1,4]. The process model graph is input interactively and the tool generates the corresponding VHDL code. The design style is restricted to behavioral models. A new scheme was formulated for the development and use of reusable code in the form of *primitives*. A default set of primitives as presented in [5] was also developed. The tool can also attach to any VHDL analyzer available on the system and analyze the developed code. This tool is designed for use by system modelers and should simplify the process of model development, thus improving productivity.

Acknowledgments

I thank my advisor Dr. James Armstrong for his guidance and support. Studying under him has been a very rewarding experience.

I thank Dr. Walling Cyre and Dr. Joe Tront for serving on the committee to review my thesis. I am also grateful to Dr. Tront for having given me invaluable opportunities to learn.

I thank my friends and colleagues for their support and encouragement.

I thank my parents for being loving and supportive through all my endeavors, for serving as the finest role models, and for teaching me the value of hard work. I thank my brother for being my best friend.

Table of Contents

List of Illustrations

Chapter 1

Introduction

For the last several years, the complexity of integrated circuits has been increasing at an astonishing rate. Designing such large and complex integrated circuits with millions of transistors requires extensive use of computer aids. One of the most important computer aids that has contributed to this ongoing technological revolution is computer simulation for system testing and verification. With an increasing number of vendors that offered simulation systems, came the need for a versatile means of accurately describing the hardware to be simulated. Netlists were ruled out since they were inadequate in their support of higher level abstractions. Higher level languages such as C or Pascal were unsuitable since they lacked the necessary constructs to model a concurrent hardware system. Out of these shortcomings arose several specialized hardware description languages such as AHPL [9]. These specialized languages soon evolved into the VHSIC Hardware Description Language (VHDL), which has since been endorsed by the IEEE as a standard [2].

VHDL has proven to be an extremely important design and simulation tool. It has become the hardware modeling language of choice for a large percentage of system modelers. VHDL has also been found to be very useful in the documentation of digital circuits. A restricted modeling style of VHDL has been found to be a good input to a high level synthesis program. Efforts are now being made to use VHDL to model various other kinds of systems like mixed analog and digital, and other systems that have no obvious correspondence to electronic hardware but can still be accurately modeled because of the ability of VHDL to do behavioral modeling.

However, the price for the richness and versatility of VHDL, is that the language is verbose and difficult to learn. When reading a program, the amount of information presented to the reader is so great that often simple programs appear complicated. To overcome the difficulty created by these two problems, the *process model graph* was developed [1, 4]. Since the nature of the systems that are modeled by VHDL, for example a digital hardware system, is essentially concurrent, as opposed to being sequential, these systems can be best expressed in a concurrent graphical representation. A common example of a concurrent graphical representation is a circuit diagram. The process model graph is one such concurrent graphical representation tailor made for use with VHDL programs. The process model graph is discussed in detail in section 2.3.

1.1 Previous Work

A software tool, the Modeler's Assistant, was developed at Virginia Tech that took as input a process model graph and produced as output VHDL code [4]. This tool was developed for the IBM PC environment and used MetaWINDOW ((c) 1986 Metagraphics Software Corporation) for all the graphics. This tool was enhanced to include a set of built in primitive processes to take advantage of the reusable nature of VHDL code [5].

There were several limitations and oversights in this version of the Modeler's Assistant, also, the number of bugs in the software were so large that effective usage of the software was completely precluded.

Among the limitations of the PC version of the Modeler's Assistant were the following:

1. Bus resolution functions were not available in the system. Lack of a bus resolution function seriously restricts the kinds of models that can be built.

2. Multiple fan out from one process port was not permitted. This meant that a signal cannot be used as an input in more than one process. This again induces severe limitations on the kinds of systems that can be modeled. Any attempt to model a real system would produce code that was very unnatural since without multiple fan out each process would need to write to several copies of the same signal for each process that uses that signal.

3. In the PC version of the Modeler's Assistant, VHDL statements were created by picking from menus, a sequence of low level tokens that represented elementary VHDL constructs. There were several problems with the low level tokens. The system of picking low level tokens was extremely counter-intuitive and the user was bound to make many mistakes. But once the user made an error there was no way to recover since the software did not have any error checking mechanisms built into it. If the user made an error, the system would simply lock up forcing the user to reboot the machine with the result that all his work would be lost. The low level token system also had several bugs, for example, many of the menu items in the low level token menu were not attached to any code, thus producing unforced errors with the outcome that all the previous work was lost. There were also some conceptual problems with low level tokens. The first problem was that there was no way to see a statement that was built using low level tokens. Hence specifying the functionality of even a medium size process required a pencil and paper to keep track of the statements added. As a consequence of not being able to see the statements, they could not be corrected or modified at a later date. This meant that to specify the functionality of a process one would have to get it right the very first time.

4. The problems with the low level tokens were alleviated to some extent by the primitives library. However, there were some problems with the primitives library too. The first problem was that the primitives library was built into the software which made it impossible for the user to add or delete a primitive from the library. To add or delete a primitive one would have to reprogram the software with or without the code for the concerned primitive and then compile the entire code again. This of course meant that every user would need to have access to the source code. Once a primitive process was added to the model it was impossible to even modify it slightly. For example if a negative edge triggered counter was required instead of a positive edge triggered counter, the only way to get it would be to use an inverter process primitive with the positive edge triggered counter process primitive. Of course when a particular process required was not in the primitives library, that process would have to be built with the unreliable low level tokens.

There were several bugs in the software. Some of the obvious bugs with disastrous consequences are described in the following paragraphs.

Some of the menu items had erroneous code attached to them. This meant that picking those menu items almost always resulted in the system getting locked up.

When trying to delete a process from a model, the model would become completely deformed. Usually half the processes would disappear and the signals

would point to ports in invisible processes. The VHDL code produced would be unidentifiable.

The software never checked with the user whether he or she wished to save their work before moving to a level where their work would become irrecoverable. This caused trouble the most number of times because accidentally picking a wrong menu item meant that without warning several hours of work could be lost.

There were other bugs where menu items had no code attached to them. To name a few, Delete Signal, Delete Process Port, Delete Variable and Delete Constant had no code written for them. All the Delete menu items become particularly important if the program is to be used effectively.

The PC environment induces some limitations on the user as well as the programmer. A very significant limitation is that the application cannot be ported to any other platform. This is because the application uses graphics, and graphics programs written for the PC are very device dependent and thus cannot be ported to a different machine with different device drivers. Also since the IBM PC is not a particularly powerful machine, there is a limit induced on the degree to which the application can be developed. Other shortcomings result from MetaWINDOW not being a very sophisticated graphics package. For example, with MetaWINDOW it is very difficult to pop up a screen window, or to create a window to edit or display text. Both of these would be very desirable building blocks to construct powerful applications. For example, in the PC version of the Modeler's Assistant, to see the VHDL code generated, one has to quit the application and type the VHDL text file from

the DOS prompt. It would however be far more useful if the application could pop up a window and display the generated code. Finally since most VHDL toolsets are written for bigger machines, like UNIX workstations, it would be an erudite decision to also have the Modeler's Assistant running on similar machines.

1.2 Contributions and Scope of Solution

The Modeler's Assistant was completely rewritten for a UNIX system with graphics implemented using the X Window System ((c) 1988 Massachusetts Institute of Technology and Digital Equipment Corporation). Carried over from the PC version of the Modeler's Assistant are only the look and feel, and the core data structure which stores the process model graph information.

The basic look and feel of the PC version of the Modeler's Assistant was found adequate because it had a drawing area, an on screen menu, a prompt line and an input box. These were embellished in the present version with highly functional scroll bars, pop-up text display windows, pop-up text editing windows and pop-up list windows. The core data structure, the details of which are presented in section 4.2, was also found to be adequate to store all cases of the process model graph input by the user.

The X Window System was chosen as the means to implement the graphics for several reasons. Firstly X Windows has become the de facto industry standard for

graphics. All workstation vendors provide support for X windows. Secondly programs written for X windows, can be ported freely from one platform to any other, and since X windows is so wide spread such a program can be run virtually on all machines. X windows is also a very powerful medium for creating advanced graphics. A brief discussion of X windows is presented in section 2.1. The Modeler's Assistant was written for Version 11, Release 4 of the X Window System.

Since graphics with X windows are considerably different from graphics with MetaWINDOW, all the drawing and display routines had to be rewritten for X windows. Also since programming with the Xt toolkit, a toolkit available with X windows, enforces an object oriented program structure, the entire flow of control of the program had to be rewritten in order to conform to this programming style.

Conceptually the Modeler's Assistant went through major changes. It was found that using low level tokens to specify process statements was not time efficient and prone to errors. It also severely restricted the statements that could be constructed. In order to make low level tokens more versatile, a very large set of cases would need to be accounted for, which would further complicate their usage and cause them to be still more prone to error. Low level tokens were done away with and instead the functionality of the process is now input textually. A special text editing window was created for this. This system of textually specifying functionality is discussed further in section 4.4.

The system of addition and usage of reusable code in the form of process primitives was completely redone. Now it is possible to add and delete process

primitives from the primitives library. It is also possible to produce a process primitive starting from just the VHDL code of a process. A library containing the set of primitives presented in [5] was built for the new system of primitives and is available with the software. This new system of process primitives is discussed in section 3.7.

A useful feature for a tool like the Modeler's Assistant would be for it to be able to analyze the VHDL code generated from within the application. Since the Modeler's Assistant was now running in a multiprocessing environment for which VHDL analyzers were available, this became possible. It involved some UNIX systems programming that spawns a new process, runs the analyzer, takes the output and displays it in a window. The technique used in running the analyzer is presented in section 4.6.

The bus resolution function was added in the form of two new resolved data types. These data types have the bus resolution function in their type declarations. Only process I/O ports of one of these data types will allow more than one inputs. Multiple fan out was also implemented. Details of the multiple fan out and the bus resolution function are presented in section 3.4.

The present version of the Modeler's Assistant is designed for a particular modeling style. The modeling style is restricted to behavioral modeling with processes. A typical VHDL program written in this modeling style would be in one file, with a single entity and one architectural body. The architecture would contain a number of processes. The Modeler's Assistant does not support block statements. Signals can only be of one of the following data types: Bit, MVL, Boolean, Integer, Real,

Bit_Vector, MVL_Vector, TSL and TSL_Vector, where MVL and MVL_Vector are the four valued multiple valued logic data types and TSL and TSL_Vector are resolved MVL and MVL_Vector data types respectively. These data types are declared in the package VHDLCAD (see Appendix A). The bus resolution function, also in the package VHDLCAD, can be changed as per the requirements of the user. Variables and constants can be of any of the above data types except TSL and TSL_Vector. Generics can only be of one of the following types: Integer, Real, Boolean and Time.

Any kind of model not precluded by these restrictions can be built with the Modeler's Assistant.

Chapter 2

Background

This thesis describes the development of a VHDL CAD tool the Modeler's Assistant for X Windows. In order to understand the details of the design issues presented in the later chapters, it is important to have some knowledge of the X Window System, the motivation behind graphical modeling, and the graphical modeling scheme used in the Modeler's Assistant, the process model graph. In this chapter, each of these areas are discussed in some detail.

2.1 Programming with X windows

The X Window System is an industry standard graphics system that lets programmers develop device independent code for user interfaces [12,13,14,15,16]. This device independent nature of X programs allows them to be freely ported to any

other platform running the X Window System. X is now available on most UNIX workstations and also many personal computers. There are also several hardware products that are specifically designed to support X.

The architecture of the X Window System is based on the client-server model. The server is responsible for all input and output devices like the display, the keyboard and the mouse. All outputs to the display are controlled by the server. Similarly all inputs from the keyboard or the mouse go first to the server. The X server is a single process that runs typically on a workstation or a personal computer with a graphics display. Some vendors are now offering dedicated X terminals that have all or part of the X server implemented in the hardware.

The client, sometimes called the application, uses the facilities provided by the X server. The client communicates with the server using a clearly defined protocol known as the X Protocol. The X protocol is designed to communicate all the information necessary to operate a window system over a single asynchronous bidirectional bit stream. Below the X protocol any kind of bidirectional lower level network protocol may be used. When the client and the server are on the same machine, the two can communicate over interprocess communication channels, shared memory, UNIX domain sockets, or System V streams. When the client and server are on different machines, one of the many network protocols supported by X, including TCP/IP, DECnet, and Chaos, can be used. An individual client can also connect to multiple servers.

The X architecture hides most of the device dependent pieces inside the server and away from the client. This allows client code to become portable. A client can talk to an X server on any machine as long as they use the X protocol.

The X protocol specifies four types of messages, requests, replies, events and errors, that can be sent over the communication channel. A request is sent by the client to the server and replies, events and errors are sent by the server to the client. A request can carry a wide variety of information such as the specification for drawing a line or the specification to change the color value of a cell. Replies are returned by the server in response to certain kinds of requests, such as querying request. An event is sent to the client containing information about a device action or a side effect of a previous request. A typical event is a key press on the keyboard. An error is like an event but is handled differently by the client.

X clients ordinarily do not use the network protocol directly, but rather work through an interface called a client programming library. The C language client programming library is known as Xlib. Xlib offers various high level routines such as routines that open a connection to the server, or routines to draw simple graphics like circles and lines. These routines contain the appropriate protocol requests. Protocol replies, events and errors are dealt with partially by Xlib, but the cases that can not be handled by Xlib fall through to the client which has to have the code to take care of such cases. A typical event that cannot be handled by Xlib is an expose event in which the server informs the client that some part of the client window that was covered has become exposed. To handle such a case the client needs to be programmed to redraw the portion of the window that has become exposed.

Built above Xlib are various toolkits that offer specialized routines for more complex graphics. A standard toolkit that comes with X, is the Xt toolkit and the widget set that goes with the Xt toolkit, the Athena Widget set. The Xt toolkit routines, the Athena widgets and Xlib routines were all used in the programming of the Modeler's Assistant.

2.2 Graphical Modeling

A graphical representation is a powerful method of clearly depicting a concurrent system. This is because a graphical representation offers a very clear visual model of the structure of the design in terms of interconnections between physical partitions and the interconnections form the core of any concurrent system. The physical partitions are essentially input to output relationships whose inputs come along some interconnections and whose outputs are propagated away along other interconnections.

Whereas text can be very powerful when specifying the details of the input to output relationships of the physical partitions of the system, text is a very inadequate means of representing concurrent systems. A typical case is the circuit diagram. The clarity with which a circuit diagram presents the essentials of a circuit cannot be matched by a textual description. However, a textual description of one of the constituent pieces of the circuit diagram, which may be in the form of a formula, is the most concise description of the particular device.

The Modeler's Assistant was developed with a view to exploit the power of both graphical and textual representation. The details of the design of the Modeler's Assistant is presented in chapter 3 and chapter 4.

VHDL as a language was developed because of the need for a higher level language that could conveniently represent digital hardware, which is essentially a concurrent system. Hence models built with VHDL would tend to have a concurrent nature. It can thus be concluded that a typical VHDL program would have a strong correspondence with some kind of concurrent graphical representation. One such representation is the process model graph which is described in detail in the next section.

2.3 Process Model Graph

In designing the VHDL language, the authors of the language have added the one important feature not found in other higher level languages, the ability to represent a concurrent system. At the same time since the language could be used to model the system at a high level of abstraction, enough richness had to be ingrained in the language so that very abstract models may also be successfully created. As a result of being a rich language VHDL can now be used to model many kinds of concurrent systems. Hence when choosing a graphical representation, a reasonably generic representation had to be found. The process model graph is one such representation [1, 4].

The architecture body of any behavioral VHDL model consists of a set of processes. These processes all run concurrently. Within the processes is code that runs sequentially. This immediately corresponds with the graphical representation of a concurrent system. The process model graph uses this property of the architecture of behavioral models. It contains a set of graphical tokens that represent processes, and arrows that connect these tokens representing signals. The details of the usage of the process model graph are presented below.

A process is represented as a single circle as shown in figure 1. The small circles on the boundary of the circle that represents the process are the process ports. A process port is simply a temporary signal that the process either writes to or reads from. The process port may be empty or filled. A filled process port means that that process port is in the sensitivity list of the process, whereas an empty process port is not. A filled small square inside the process represents a constant used by that process. An empty small square represents a variable.

In figure 1, PROC is the name of the process. The process has three process ports P1, P2 and P3. Process ports P1 and P2 are in the sensitivity list of process PROC. VAR is a variable in the process and CNST is a constant of the process.

One or more processes together form the process model graph. Two process ports can be connected together by a signal. The direction of the arrow indicates the process that writes to the signal and the process in which the signal is read. The process at the head of the arrow uses the signal as input and the process at the tail of the arrow

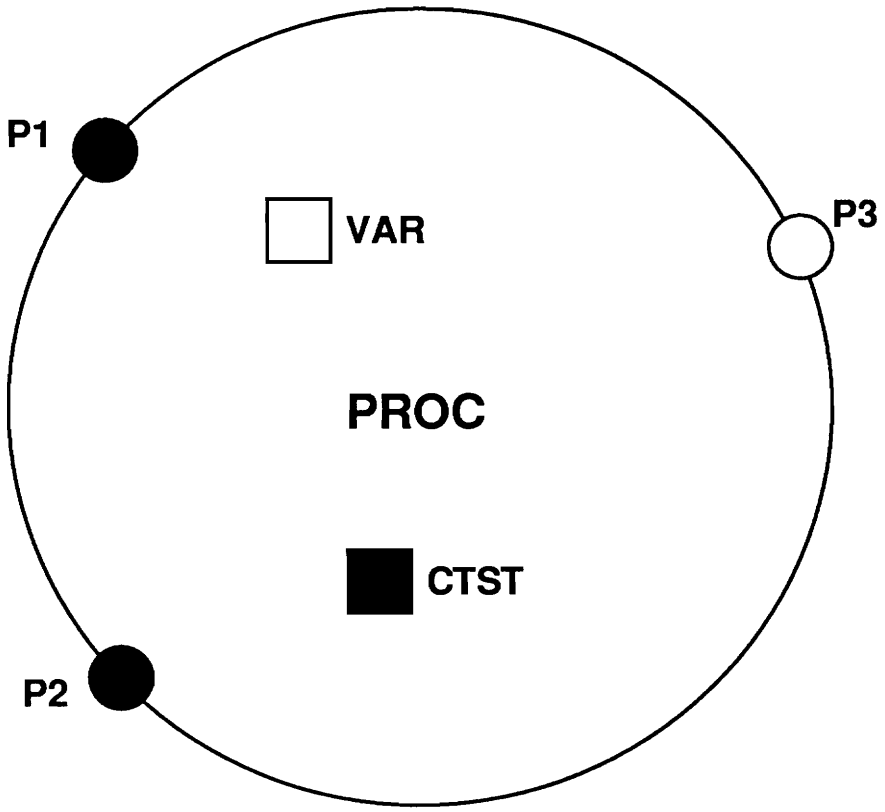


Figure 1. Graphical Representation of a Single Process

writes to the signal. The width of the line that represents the signal is proportional to the size of the array that it represents. Once a process port is connected to a signal, the name of the process port is overridden by the name of the signal at all further instances of the process port. Any process port not connected to a signal is assumed to be an entity port. Two unconnected process ports of the same name and type are assumed to be the same entity port.

Figure 2 shows a process model graph. The process model graph contains three processes PROC1, PROC2 and PROC3. There are two signals SIG1 and SIG2. The entity ports are INP and OUTP. Since there are two unconnected process ports with the name INP, they are considered to be the same and only one entry is made in the entity port list.

A process model graph of the VHDL code in figure 3 is in figure 4. Since a process model graph contains less information than the VHDL code that it represents, information needs to be added to the process model graph in order for it to produce a unique VHDL program. The process model graph along with this kind of information is collected by the Modeler's Assistant, where it is used to generate VHDL code. The details of the kind of information required and the method used to collect it is presented in chapter 3 and chapter 4.

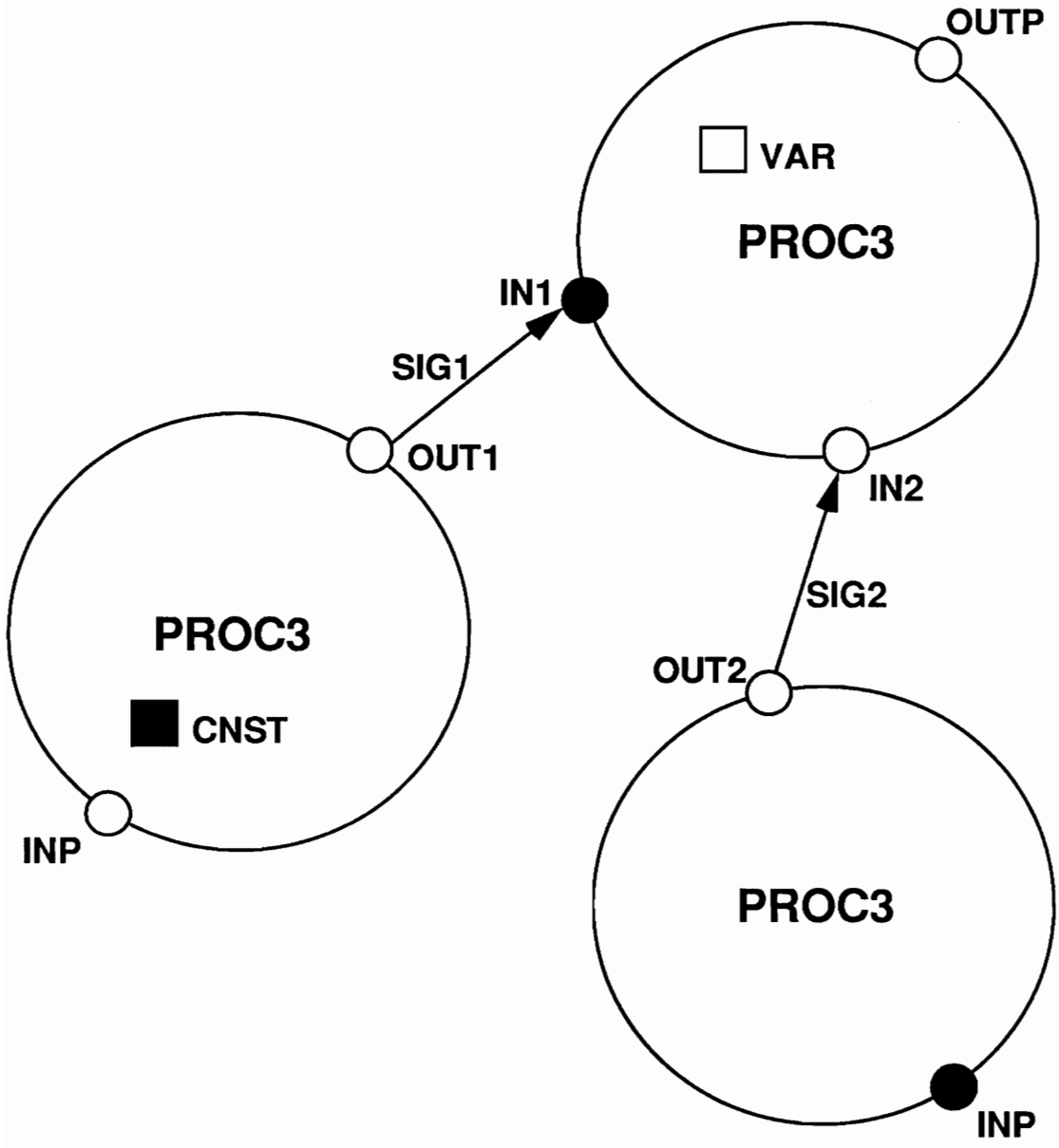


Figure 2. Process Model Graph with Three Processes

```
entity TEST is
        port (TRIG: in BIT; OPUT: out BIT);
end TEST;

architecture BEHAVIOR of COUNTER is
        signal SIG: BIT;
begin
        PROC_1: process(TRIG)
        begin
                SIG <= TRIG after 100 ns;
        end process PROC_1;

        PROC_2: process(SIG)
        begin
                OPUT <= SIG after 100 ns;
        end process PROC_2;
end BEHAVIOR;
```

Figure 3. Sample VHDL code

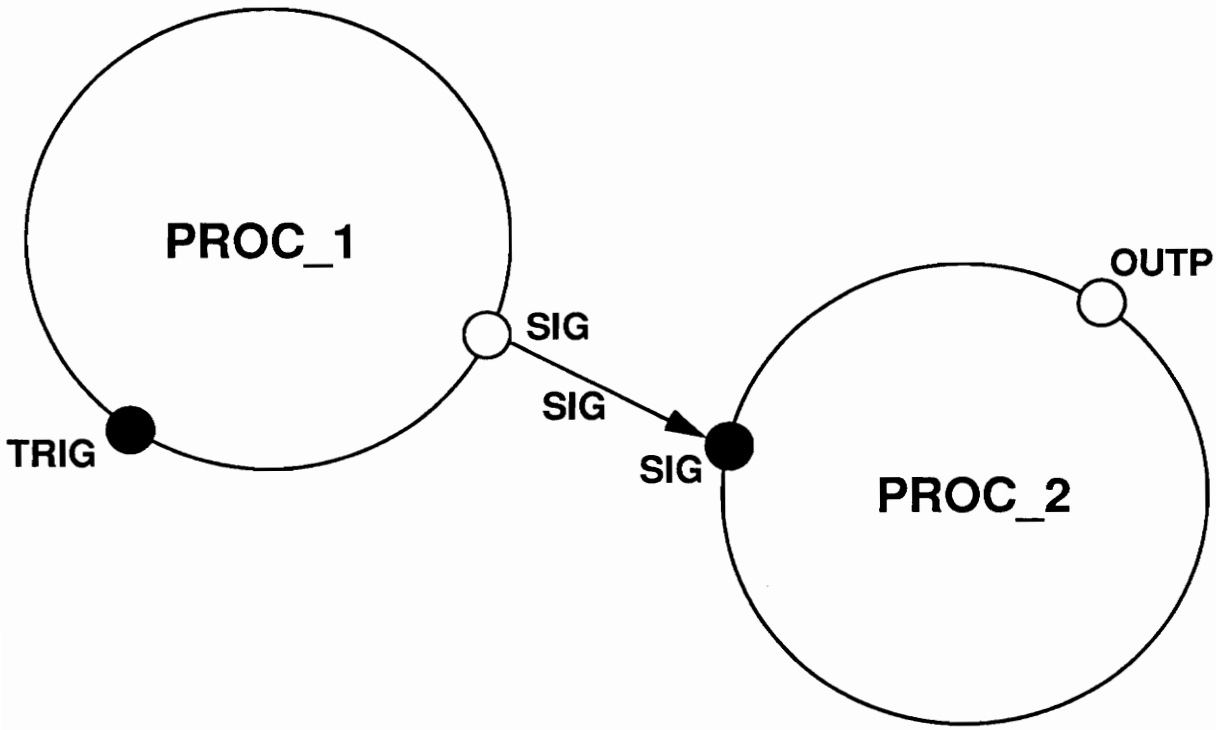


Figure 4. Process Model Graph of VHDL code in figure 3

Chapter 3

Design Specifications

Prior to beginning the construction of a large software tool it is important to clearly define the specifications of the project in terms of detailed functioning of the various parts of the tool. These specifications may change several times during the course of building the tool because of new experience that is continuously acquired. In this chapter the specifications that were implemented in the final design of the Modeler's Assistant are presented.

3.1 Information Required

The Modeler's Assistant was to be used to generate a behavioral model of a system. The VHDL model to be generated was to be a single entity, single architecture model, with the architecture body containing only processes. Figures 3 and 4 in the

previous chapter show a typical VHDL behavioral model built with processes and the process model graph of the model. Obviously the process model graph does not contain all the information present in the VHDL program. The minimum amount of information required in addition to the process model graph needs to be determined. By comparing the VHDL code and the process model graph it can be seen that the process model graph does not contain the functionality of the process, that is the VHDL code that appears between the *begin* and *end process* statements, the mode or type of the entity ports, the entity generics, and the types of the signals, variables and constants.

Parts of this information need to be input textually, and the rest graphically. By using the premise that the concurrent part of the model be input graphically and the rest textually, it can be seen that only the functionality of the process should be input textually. The rest of the information would need to be input graphically. By parsing the text of the functionality, the mode of the process ports can be determined. This can be done because a process port of mode *out* would have a signal assignment statement made to it, whereas a process port of mode *in* would occur somewhere in the program but would not have a signal assignment statement made to it. An entity port that is of mode *in* in some process and of mode *out* in other process would be of mode *inout*.

To build a complete process model graph, the information displayed in the process model graph, as well as the generics, and the types of the process ports, variables and constants need to be input interactively by the user. The best time to ask the user for types of the process ports, variables and constants would be when the user first adds these structures to the process model graph. The generics, and their types could be added at the same time as the process ports, variables and constants.

Information about the type of a signal need not be queried from the user since it would be of the same type as the process port that the signal is connected to. Details of the technique used to obtain the above information is presented in the following sections.

3.2 Building a Process

The first step towards building a process model graph is to build all the processes. The first piece of information required while building a process is the name of the process. This name will be used as the process label in the VHDL code generated. Having gotten the name of the process, a circle representing the process should be drawn on the screen.

Next a menu should offer to add process ports, variables, constants or generics. When a process port is chosen to be added, a menu with the standard signal types should be presented from which the user can choose the specific type of the process port. The types that were to be offered were Bit, MVL, TSL, Integer, Real, Boolean, Bit_Vector, MVL_Vector and TSL_Vector. MVL and MVL_Vector are the four valued multiple valued logic data types. TSL and TSL_Vector are bus resolved MVL and MVL_Vector. All four of these declarations, as well as the bus resolution function are in the package VHDLCAD. See appendix A for the package VHDLCAD. The choice of a resolved type for a process port becomes important when adding signals. Once the type of the process port is chosen the user should be able to place the process port at any position on the boundary of the circle representing the process. The location of the process port should be marked by a small circle. There should be a menu item

that allows the sensitivity of the process port to be toggled. A process port that is to be in the sensitivity list of the process should be marked by a filled small circle.

Similarly, to add variables and constants, the name of the variable or constant should be input following which the type of the variable or constant should be input. The types for the variables and constants should include Bit, MVL, Integer, Real, Boolean, Bit_Vector and MVL_Vector. The location chosen for the variable should be marked by a small unfilled square. The location for the constant should be marked by a small filled square.

To add a generic the user should be able to simply pick the menu item to add generics, and in response he should be asked for the name and type of the generic. The types of generics offered are Integer, Real, Boolean and Time. Although the generic does not have a graphical token associated with it, the presence of a generic in a process should become apparent by popping up a specify window.

There should also be provisions for deletion or modification of process ports, variables, constants and generics. The modifications to be allowed are change of name, or type of the relevant structure. The position of the process ports, variables and constants on the screen should also be changeable.

3.3 Specifying Functionality of the Process

Once a process is built by the procedure outlined in the previous section, it represents a complete process except for the VHDL code that lies between the *begin* and *end process* statements. This code, which is the functionality of the process should be specified textually. A text editor window should be popped up in response to the user picking the menu item to specify functionality. Within this window should also be displayed information regarding all the process ports, variables, constants and generics. The window should contain a full screen editor with which the user can type in the details of the functionality of the process. It should also be possible to import a previously created file containing the relevant information into the text editor window. The names used in specifying the functionality of the process should be the same as those used while defining the process graphic. If the functionality of the process was previously specified, then the editor window should contain the previous specification so that it may be edited.

After the functionality is fully specified, the pop-up window should be closed and the text of the functionality should be parsed to determine the mode of each of the process ports. The mode of process port is *out* if there is a signal assignment statement made to the process port, *in* if the process port is used without a signal assignment statement made to it, and *inout* if both of the above are true.

At this stage the all details of the individual processes are specified. All that remains is to connect them with signals to form a complete process model graph.

3.4 Building a Process Model Graph

To build a process model graph, referred to for brevity as a *unit* in the Modeler's Assistant, each of the previously built processes should be invoked and placed on the screen. The user should be given the choice of the location of the process. The user should also be able to move the process on the screen.

Once the processes are placed on the screen, the user should be able to add signals between process ports on different processes. However before any signal is added the software should ensure that both process ports are of the same type. If the process port types match, the user should be prompted for the name of the signal. This name should replace all the corresponding process port names where ever they appear, including in the text of the functionality of the process.

Before drawing the signal, the modes of the two process ports being connected should be ascertained and the arrow head on the signal should point towards the process port of mode *in*. If the process port where the signal starts is of mode *in* or if the process port where the signal terminates is of mode *out* then an error should be flagged immediately. If the process ports are not bus resolved and if the destination process port is of mode *inout*, then too an error should be flagged. If the process ports are of one of the bus resolved types, TSL or TSL_Vector, then signals are allowed to have destination process ports of mode *inout*.

Several signals can originate at the same source process port, as long as each of them do not violate the rules laid out above. If a signal originates from a process port that already has another signal originating from it, then the new signal takes the name of the first signal, and together they represent a single signal. More than one signal cannot have the same destination process port unless the process ports are of one of the bus resolved data types. By sticking to these rules, dynamic errors at the time of simulation of the model, saying that signals have multiple sources, are avoided. By correctly adding all the signals, the entire process model graph would be constructed.

3.5 Generating VHDL Code

Once the process model graph is completely built, VHDL code for the model can be generated. In the entity description, the name of the entity should be the same as the name specified for the process model graph. The entity port list should contain the process ports that remain unconnected to any signals. If two unconnected ports have the same name and are both of the same data type, then they represent the same entity port and only one entry is to be made in the entity port description. This is also the rule that applies to generics. If two generics in different processes have the same name and type then they represent the same generic and only one entry is to be made in the entity generic list.

The architecture is always to be named BEHAVIOR. The signal list is to contain all the signals in the process model graph. Of course, only one entry is to be made for multiple fan out or multiple fan in signals. Within the architecture body each

of the processes in the process model graph should be listed. In the process body the functionality of the process specified by the user is to be transcribed. However, every occurrence of the name of a process port is to be replaced by the name of the signal it is connected to, unless the process port is not connected to any signal. In this way the VHDL code corresponding to the process model graph can be generated.

Once the VHDL code is generated a window should pop up and display the newly generated VHDL program. This would be useful because the user could trace errors in the process model graph by looking for abnormalities in the VHDL code.

3.6 Analyzing VHDL Code

A design goal set out for the Modeler's Assistant was to be able to analyze the VHDL code generated by it from within the program. This should be done by taking from the user the command line to be run to use the analyzer that user would like to use. This information should be set as a UNIX environment variable. The program should now generate the VHDL code and from within the program start the analyzer. The output of the analyzer is to be taken by the software and displayed in a window. By doing this one could ensure that the models generated were syntactically correct without having to leave the program.

3.7 System of Process Primitives

A very important design goal was to have a versatile system of primitives to exploit the reusable nature of VHDL code, particularly the code that forms a process. The system should allow the user to produce his or her own set of process primitives. This was determined to be an important feature that should be incorporated in the primitives system because it would then be possible for a design group to develop their own set of primitive processes. It should also be possible to place these process primitives in a design library where they can be accessed by an entire design group.

The nature of the primitive process had to first be decided. The function of the process primitive was to create a process which could be used in the construction of a process model graph. Hence the process primitive had to have all the information to create a complete process, including the functionality of the process. At the same time the process primitive had to have enough information so that it could prompt the user for specific names to replace the generic names in the primitive. The generic names in the primitive would be the names of the processes ports and the names of the generics. Hence to create a primitive, one would need a completely functional process and the prompts to be used while prompting for the various process port names and generic names.

The method used to collect all the above information to create a primitive was for the user to first create a process in which he would ensure that there are no syntactic or semantic errors. This could be done by invoking the process in a dummy unit and running the analyzer on it. Next by picking a menu option he should be allowed to

specify the prompts to be used for each of the process port names and the generic names. The software should ensure that the prompt information provided by the user contained no errors. It should check to make sure that a prompt was specified for each of the process ports and generics. In this way the process primitive could be easily created by a user.

At the time of creating the process model graph the user should be given the option of using a process primitive to add a process. This could be done by picking a menu option for primitives, in response to which a window should pop up containing a list of all the process primitives in the primitive library. The user should then be able to pick a primitive and a generic copy of the primitive should be placed on the screen. Next the user should be queried for specific names for the process, the process ports and the generics. Once all the information is input the screen should be updated to reflect the new names. The user should now be able to treat the process created like a regular process. He should be able to modify all the attributes of this process including the functionality of the process.

The details of the issues involved in the implementation are presented in the following chapter.

Chapter 4

Design Issues

The Modeler's Assistant was written in the C programming language for the UNIX environment, with graphics implemented using X windows. The overall programming strategy as well as certain specific details are discussed in this chapter. In particular, the system architecture, the data structures used and their usage, the construction of graphics, the text parsing algorithm used, the VHDL source code generation algorithm, the methodology implemented to analyze the VHDL code, and implementation details involved in creating the system of primitives, are discussed.

4.1 System Architecture

Figure 5 shows a simplified view of the various components of the architecture of the Modeler's Assistant. These components, and their function in the rest of the

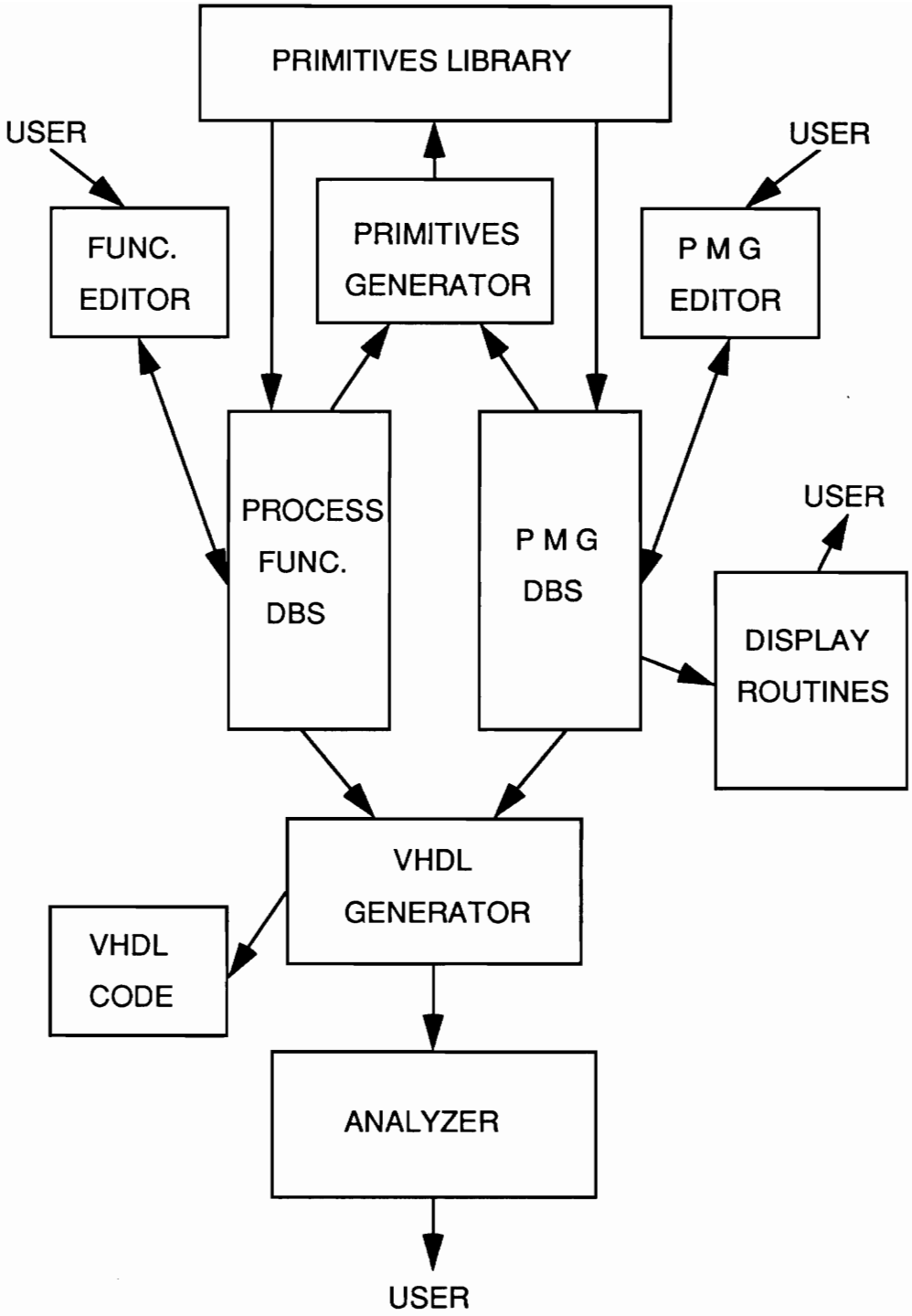


Figure 5. Block Diagram of System

system, are discussed individually in more detail in the following section. It is important to note that the arrows in the figure represent the direction of flow of data as opposed to the flow of control.

The central element of the system are two databases containing the information required to generate all the graphics as well as the information required to produce the corresponding VHDL code. The process model graph database contains the details of the geometry of the process model graph in terms of the coordinates of the various graphical constructs like the processes, process ports, signals, variables, and constants. It also contains details like the types of the signals, constants, variables and generics, the mode of the signals, information on whether the signal is in the sensitivity list of the process, and also the names of the elements of the process model graph. The process model graph editor permits the user to manipulate the information in this database. With the process model graph editor the user can add or delete process or signal information from the process model graph database. The user is also provided the ability to change the geometry of the process model graph, for example the user can change the location of a process, or he can change the position of a variable in a process. The user can also modify the characteristics of the process model graph. For example, the user can change the data type of one of the variables or signals.

The process functionality database contains the text of the functionality of all the processes currently in the process model graph. While the process model graph database resides in the RAM memory of the machine, the process functionality database is kept in temporary files on the machine's permanent disk. This approach was adopted because the memory requirements of the software became smaller and at the same time

the program was no longer bound by set limits on the size of the model that could be created. Yet the price paid in increased access time of the information did not deteriorate the performance simply because the information in this database was used very infrequently. Calls to this database were made only at the time of specification of the functionality and later at the time of generation of the VHDL code, and even these infrequent calls involved reading or writing only a single block of data, as opposed to a continuous exchange of data. Information in this database can be modified by using the process functionality editor. This is essentially a text editor, which is used to edit the text of the process functionality, and a text parser that extracts the mode information from the text and passes it to the process model graph database. The process functionality editor can also import a file from the disk in which the user had previously entered the process functionality.

Whenever the information in the process model graph database is modified by the process model graph editor, display routines are called to update the graphics on the screen. The display routines take as an argument the location of the updated information in the database, and change the display on the screen to reflect the new information. For example, when the user adds a process port to a process, the process model graph editor interactively accepts all the relevant information from the user, and inserts it in the process model graph database. The information always includes at least the name and the screen coordinates of the newly created graphic. It then calls a graphics routine that draws process ports and informs it of the location of the new process port in the database. The graphics routine then draws the process port at the proper location on the screen.

The primitive generator creates the process primitives and places them in the primitives library. The primitive generator takes the process model graph information and the text of the functionality from the databases, adds the prompt messages provided by the user, and puts all the information into a file which it places in the primitives library. The primitives library is an ordinary directory in the system where all the process primitive files created by the primitive generator are placed.

The VHDL generator takes the information from the process model graph database and the text functionality database and converts it into VHDL code. This code is then sent to a file on the disk. If the analyzer is invoked from the program, then the same file serves as input to the analyzer. The output of the analyzer is piped back to the Modeler's Assistant and displayed in a pop-up display window.

This forms the gist of the system architecture of the Modeler's Assistant.

4.2 Data Structures

The Modeler's Assistant stores a variety of information. For example a process model graph constructed with the Modeler's Assistant can contain a number of processes, with each process containing a varying number of process ports, variables, constants and generics. The process model graph will also have a number of signals. For each of these graphical constructs several diverse pieces of information need to be stored. For example, with a process port, its type needs to be stored, or with a process, all process ports in that process need to be stored. Yet there are some kinds of

information that all the constructs require. All the constructs need to store a character string which would contain their name. They also need to store a pair of X and Y coordinates that define their location on the screen.

Since the number of processes or process ports or variables or any other structure is not fixed, the obvious data structure to store all this information is a linked list. This is because a linked list can be of an indefinite length and information can easily be added or deleted from it. The linked list should be constructed with unit structures, where each of the unit structures contains information on individual graphical constructs. A single unit structure that could contain the kind of information mentioned previously for any of the graphical construct, would need a field that stored the coordinates, a field that stored the name of the construct and fields that contained information about the kind of construct being stored, that is, whether the construct is a process, a variable, a constant and so on. There would also need to be a small set of fields that could be used to store all the pointers required to form the list. These pointers would contain the addresses of other constructs that were associated with it. For example, the pointers in a structure storing a signal would contain the address of the next signal in the list, the address of the structure storing the source process port of that signal and the address of the structure storing the destination process port of that signal. This along with the name in the name field, and data saying that the structure contained a signal in the type field, would completely define the signal.

The C code that defines such a structure, called a_node in the Modeler's Assistant, is as follows:

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

The C code for the structure rect is:

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

Here the field name stores the name of the construct, the length of which is limited to thirty two characters. The field type contains the type of the construct, and the array of pointers ptr[6], can store the addresses of other structures. Six pointers were found to be adequate since more than six pointers were not required by any of the constructs. The field rect is itself a structure that contains the coordinates of the upper left and lower right corner of the construct as it appears on the screen.

These structures link to each other and form long complex chains that store the entire process model graph. A complete process model graph is stored in two lists. The first is a list of the processes and the second is a list of the signals. The addresses of the heads of both these lists are stored in the array of pointers in the top structure that contains the name of the entire process model graph. Hence starting from this top

structure, any of the processes or signals can be reached by traversing the appropriate list.

Figure 7 shows the state of the linkage of the data structures when it stores the process model graph shown in figure 6. The list containing the signals is a simple list, that is it does not have any sub-lists. On the other hand each of the processes in the process list have four sub-lists originating from them. These sub-lists include a list of process ports, a list of variables, a list of constants and a list of generics. Since there is no graphic associated with the generics, the generics in figure 7 do not appear in the process model graph.

4.3 Construction of Graphics

Using X windows for the graphics imposes some unusual conditions on the design process. Because of the client server architecture of X windows, a single bidirectional bit stream needs to exist between the client and server along which the client can send instructions to the server and the server can return replies, events and errors. In a typical X application the user would take a particular action after looking at the information presented on the screen. Consider the case where the user clicks the mouse button on a menu item, in response to which the application is supposed to draw a circle on the screen. This information is taken by the server and is treated like an event, which needs to be sent to the client along the bit stream. For the client to receive this information, the client should be executing code that continuously looks for data to come from the bit stream. Upon receiving the data, the client recognizes the action to

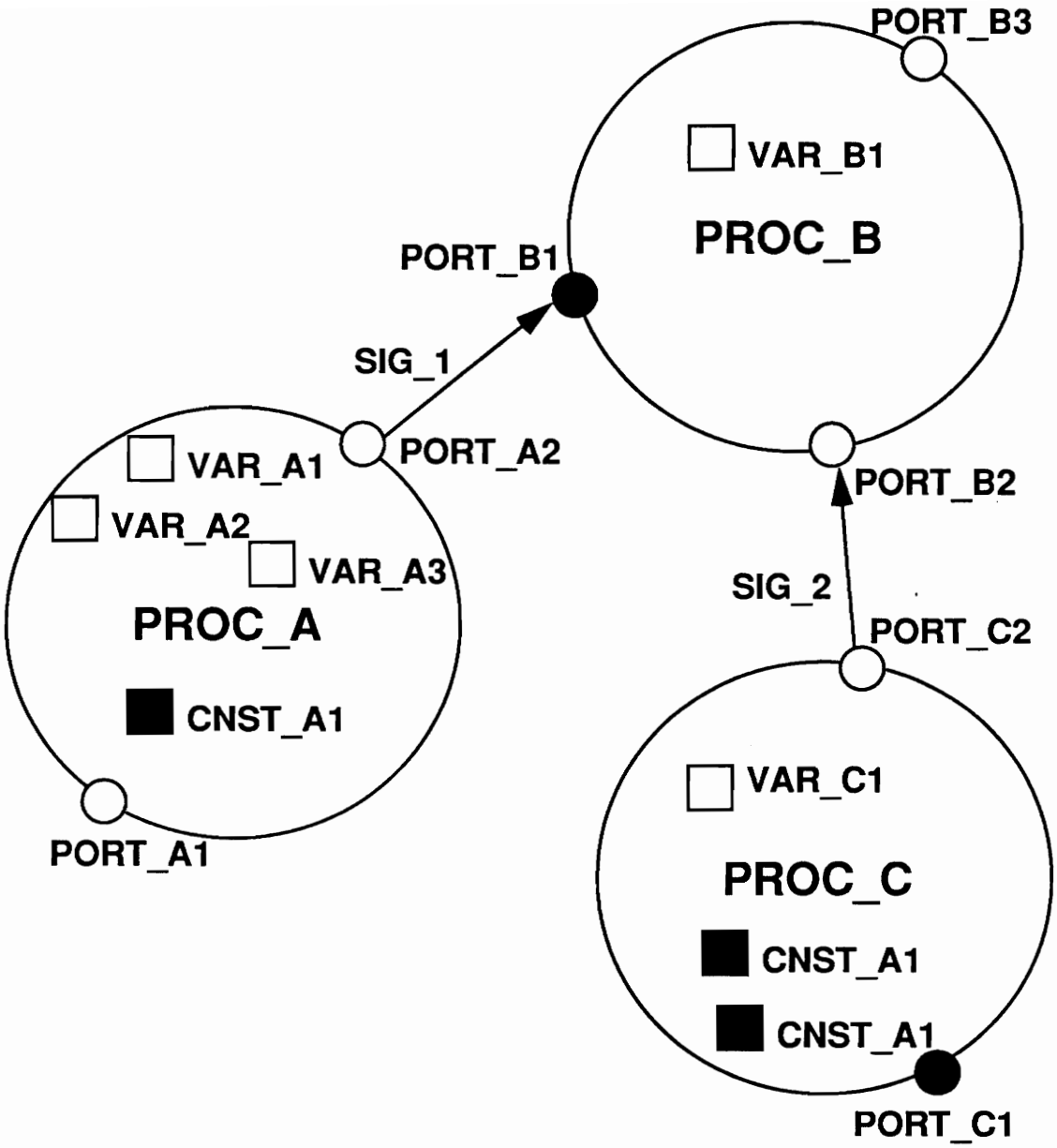


Figure 6. Sample Process Model Graph

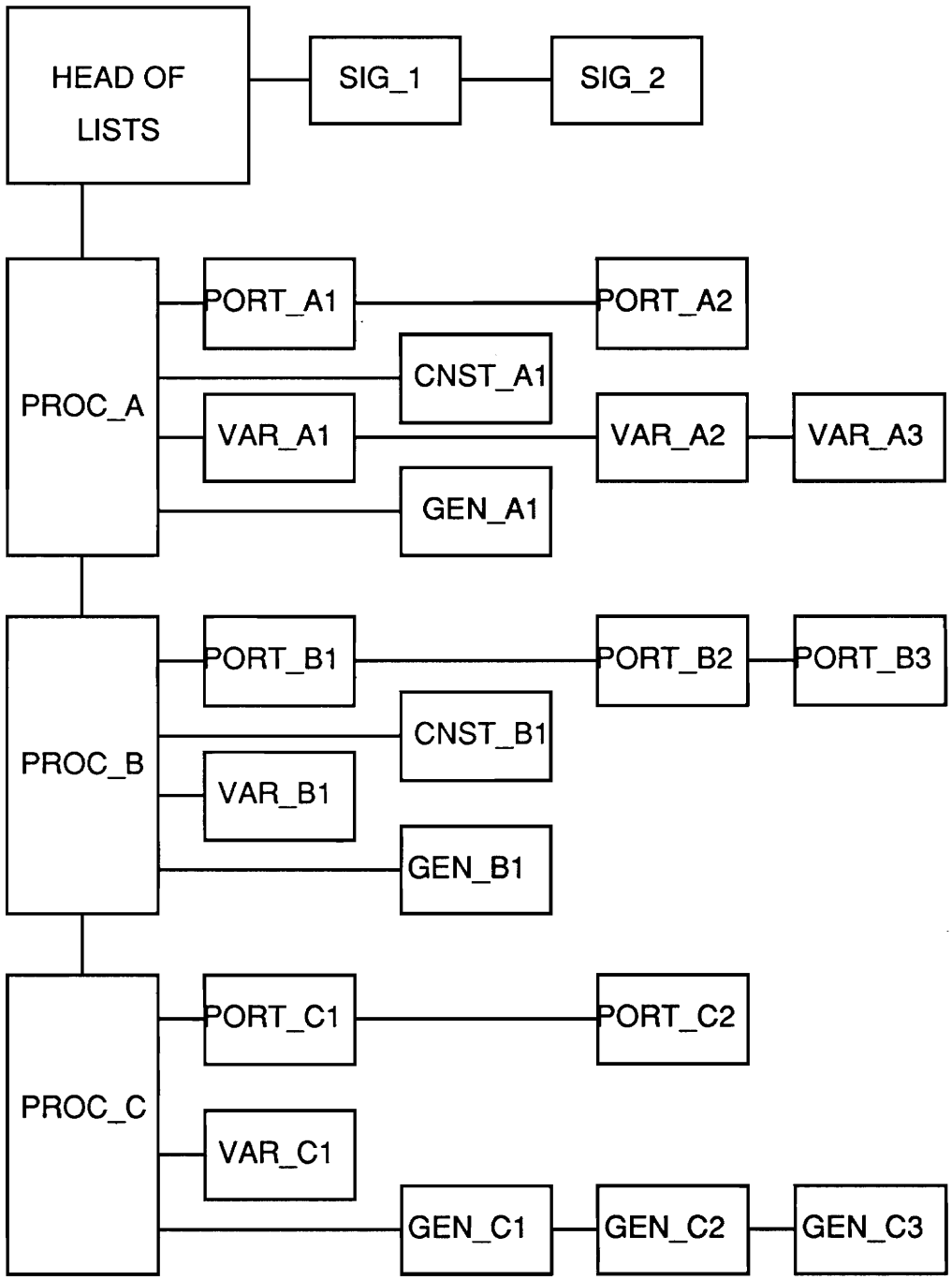


Figure 7. Structure of linked list to store the PMG of figure 6.

be taken which is to draw a circle. As soon as the application finishes drawing the circle it would need to return to the bit stream and look for more data. This means that the client code should be an infinite loop where the client spends most of its time waiting for information from the server and is interrupted only briefly to do specific tasks.

Since there are a number of possible tasks that an application may be required to do, the only feasible approach to writing such code is to have an infinite loop that waits for information from the server, and upon getting the information interprets it and calls an appropriate routine. The routine called should do its task and fall all the way through to the end and return to the main infinite loop where the application can once again wait for more information from the server. The concept is exactly the same as the concept of an interrupt service routine.

It is important to note that the routine called cannot stop at any point and ask for more information since it can only get information from the server, and information from the server can only be accepted in the main infinite loop which can be reached only when the routine ends. To stop and ask for information would thus produce a deadlock.

Hence creating an X application involves writing a large number of routines that cover all the cases of information that can be sent by the server. At the same time each routine must be written so that it has all the data that it requires at the start of the routine. Hence an X application cannot be designed using the traditional flow charts. A routine in an X application is more like a finite state machine that does different tasks

depending on the state of the application, and on completion of the task, updates the state. This was the programming methodology that had to be adopted when coding the Modeler's Assistant. When the Modeler's Assistant is first started, it sends information to the server to draw all the graphics in the window. Once the window is set up, the application simply waits. Each menu item on the screen has a routine associated with it. These routines are all designed to do a set sequence of tasks and fall through to the end so that control returns to the main infinite loop.

Since the application is running in a multi-tasking environment, the code that waits for data on the bit stream must be blocking in nature, as opposed to code that polls the bit stream for new information. Blocking code simply tells the operating system to inform it when new data becomes available on the bit stream and then becomes dormant, creating zero load on the system.

When the client has to draw an object in its window, a routine in the client program would send an appropriate instruction either directly along the bit stream to the server, or to a request buffer where several requests accumulate and are flushed to the server together. It was found that by buffering requests and sending them all at once, the speed of execution of programs improved tremendously [12]. Output intensive applications run as much as thirty times slower without the benefit of buffering. Consequently buffering is always used while building X applications. In some extraneous cases, for example when the software is being debugged, buffering can be turned off although it is not recommended.

Buffering is implemented by having all the routines write to the request buffer. Once the routines have finished making their requests and fallen through to the end, the buffer is flushed. The line of code that flushes the buffer is usually put in the infinite main loop, right before the code to wait for data from the server.

An important event that needs to be programmed for is the expose event. The expose event is sent by the server to the client when part of the client's window that was covered becomes exposed. The client then has to refresh the exposed portion of the window with the appropriate data. When programming the Modeler's Assistant, the expose event was dealt with using a technique similar to backing store. All the drawing to the window was actually done on a drawing area not mapped onto the screen. This drawing area was a dedicated part of the memory designated for use by the Modeler's Assistant, and would not change except under instruction from the program. Whenever there was an expose event, the exposed area would be updated by copying it from the off screen drawing area onto the window on the screen. Although this method creates a greater demand on the system memory, it was adopted because it was an efficient method and was not prone to error.

The actual graphics were created mainly with the Xt toolkit, although the drawing of the process model graph was done with lower level Xlib calls. The menus were created with the list widget. The list widget was found to have a quick response time and was the most feasible for the application setup. All the display windows, and the editor window were all implemented with the text widget. The scrollbars were added to the drawing area by using the viewport widget.

4.4 Text Parser

In the Modeler's Assistant the functionality of the processes is specified textually. This text of the functionality is used to determine the modes of the process ports. At the time of VHDL source code generation, this text is modified to replace process port names with signal names. A routine that looks for a specified string of characters in a long string of characters was written and used for both of the above functions.

To determine the mode of a process port, the process port was considered of mode *out* if there was a signal assignment statement made to that process port or to a slice of that process port. If the process port existed in the text without a signal assignment statement it was considered to be of mode *in*. If both of the above were true, then the mode was *inout*. Any characters in a line after the special characters "--" were completely ignored since those characters were assumed to be part of a comment. A process port was assumed to have a signal assignment statement if the characters "<=" were found immediately after the process port name in the text.

4.5 VHDL Source Code Generation

The algorithm for generating the VHDL source code takes its inputs from the process model graph database and the process functionality database and sends its output to a disk file. The algorithm starts at the top structure in the linked list that

contains the name of the process model graph. First it writes the entity name and the appropriate reserved words to start the entity declaration. It then starts writing the generic list in the entity declaration. To find the generics, the algorithm traverses the list to each process and then down the sub-list of generics. Before writing a generic, it checks if there already exists a generic with the same name and type. If there is no such generic it writes the generic name.

After the generics the algorithm starts writing the port list. Again the algorithm traverses the list along the processes and down the sub-list of process ports. It looks for process ports that are not cross linked with any signal. If a process port of the same name and type was not previously found, then this process port would be added to the entity port list. If a process by the same name and of the same type had previously occurred, then the mode information of the new process port would be combined with that of the old. For example, if the first time a process port occurs, it is of mode *in*, and the second time of mode *out*, then its mode in the entity description would be *inout*.

The algorithm then begins writing the architecture. The signals are added in a fashion similar to the one described above. After writing all the signals, it puts down each of the processes individually. It first writes the variables and constants and then goes to the process functionality database. It extracts the functionality of the process in question, replaces the process port names by the signal names and puts the new functionality in the VHDL source code file. The algorithm continues till the VHDL code for all the processes are put down.

4.6 Linking to an Analyzer

Since the Modeler's Assistant is a VHDL model development tool, the process of developing the final correct model would involve several iterations where the user would create a model, test it, go back and modify the model, test it again, and so on. It was felt that it should be made possible to analyze the generated VHDL code from within the Modeler's Assistant. This was achieved by doing some UNIX systems programming.

The user was given the flexibility to specify any analyzer to be used by setting a predefined environment variable. The analyzer specified was to be run, which meant that a new process had to be started. This was done by making the Modeler's Assistant spawn off a child process using the UNIX system call `fork()`. In the child process the analyzer was invoked. This was done with the `execv()` system call. At this stage Modeler's Assistant process and the analyzer process were both active. Now the output from the analyzer process had to be sent to the Modeler's Assistant where it was to be displayed. Since both of these were two completely independent processes, they could be made to communicate only by going through the operating system. The communication channel chosen was a UNIX pipe, which was created with the system call `pipe()`. The standard output of the analyzer process was tied to the pipe, and at the other end of the pipe the Modeler's Assistant would wait for new data which it would display in a pop-up window.

The Modeler's Assistant had to now wait at the end of two data channels, one from the server, and the other from the analyzer process. Both of these waiting

statements had to be blocking statements in order to avoid unnecessary overhead. But it is not possible for a blocking process to wait for data on two different channels. So in order to wait for data on two channels, a time out was implemented. Now the process would wait on one channel, and if no data was received for a specified period of time, it would time out and start waiting on the next channel. In this way the Modeler's Assistant was linked to an analyzer, and models could now be verified to be correct from within the program.

4.7 System of Primitives

The building of a primitive involves first building a complete and error free process. Then the user specifies the prompts to be used for each of the process ports and generics in the primitive. The process and the prompts together form the primitive. The system of generating primitives takes its inputs from the two databases and the user and integrates this information in a single primitive file to be placed in the primitive library.

Since a primitive would be used many times by several users, it needs to be completely error free. To ensure that the primitive is indeed error free, first the VHDL code of the primitive should be analyzed using the built in access to the analyzer. This would ensure that the VHDL code was syntactically and semantically correct, which would imply that the process functionality as well as the process model graph of the process were correct. The only other place where there could be an error is with the specification of the prompts. If prompts were not specified for all the process ports and

generics, or if too many prompts were specified, then the primitive would function incorrectly. So the Modeler's Assistant checks if a prompt is specified for each of the processes and generics, and there also if there are extra prompts.

At the time of the usage of the primitive, a generic shell of the process that represents the primitive is placed on the screen. Once the user has provided information for all the prompts, the shell is updated and it becomes a process, that can be treated like any other process in the process model graph.

Chapter 5

Future Work

The Modeler's Assistant can be enhanced in many directions. Some of the improvements would be large contributions, whereas others would be minor modifications.

The part of the program where the functionality of the process is specified can be used as a starting point for some large additions. The text parser can be made very sophisticated. It can be used to extract much more information than simply the modes of the process ports. For example, it can be used to extract variables and constants, and their types from the VHDL text.

Algorithms can be developed that would allow the process functionality to be input, not in VHDL, but in some other language. For example, a natural language processing system can be developed that accepts the functionality specified in English

and translates it into VHDL code. Such a system would be tremendously useful because it would allow a person with no knowledge of VHDL to be able to create sophisticated VHDL models.

Another area where the Modeler's Assistant can be enhanced is with the addition of structural modeling. Presently only behavioral models can be developed. This is because the design permits only two levels of hierarchy, the process level and the unit level. To introduce structural modeling, a multiple level hierarchy would need to be developed. This could take the behavioral model of the present system and build a structural model above it.

Several other smaller enhancements that can be made. More data types can be offered for the process ports, variables, constants and generics. User defined data types can also be introduced. Another minor enhancement that can be made is to have the system save information in files, not as data, but rather as text. Then models created on one machine can be used on a different machine. This cannot be done now because different machines have different word lengths, and they represent the data differently.

These are some of the modifications and enhancements that can be made to the Modeler's Assistant in the future.

Bibliography

1. James R. Armstrong, "Chip Level Modeling with VHDL," Prentice Hall, New Jersey, 1989.

2. "IEEE Standard VHDL Language Reference Manual," IEEE, New York, 1988.

3. Moe Shahdad, et. al., "VHSIC Hardware Description Language," *IEEE Computer*, Vol 18, No. 2., pp.. 94-103.

4. David G. Burnette, "A Graphical Representation for VHDL Models," *Master's Thesis*, Virginia Polytechnic Institute and State University, 1988.

5. Chakravarthy S. Kosaraju, "A Set of Behavioral Modeling Primitives," *Master's Thesis*, Virginia Polytechnic Institute and State University, 1990.

6. James R. Armstrong, "Chip Level Modeling with HDLs," *IEEE Design and Test of Computers*, Vol. 5, No. 1, pp. 8-18, Feb 1988.

7. James R. Armstrong, "Chip-Level Modeling and Simulation," *Simulation*, pp. 141-148, Oct. 1983.

8. David R. Coelho, "The VHDL Handbook," Kluwer Academic Publishers, Boston, 1989.

9. Fredrick J. Hill and Gerald R. Peterson, "Digital Systems: Hardware Organization and Design," John Wiley and Sons, New York, 1978.

10. Roger Lipsett, et. al., "VHDL: Hardware Description and Design," Kluwer Academic Publishers, Boston, 1989.

11. Chakravarthy S. Kosaraju and James R. Armstrong, "A Set of Behavioral Modeling Primitives," *IEEE Southeastcon*, pp. 610-613, 1990.

12. Oliver Jones, "Introduction to the X Window System," Prentice Hall, New Jersey, 1989.

13. Douglas A. Young, "The X Window System, Programming and Application with Xt," Prentice Hall, New Jersey, 1990.

14. Joel McCormack, Paul Asente, Ralph R. Swick, "X Toolkit Intrinsics - C Language Interface." 1990.

15. Chris D. Peterson, "Athena Widget Set - C Language Interface." 1990.

16. O'Reilly & Associates, Inc., "The Definitive Guides to the X Window System," O'Reilly & Associates, Sebastopol, California, 1990.

17. David A. Curry, "Using C on the UNIX System," O'Reilly & Associates, Sebastopol, California, 1989.

18. Brian W. Kernighan, Dennis M. Ritchie, "The C Programming Language," Prentice Hall, New Jersey, 1977.

19. Bjarne Stroustrup, "The C++ Programming Language," Addison-Wesley, Reading, Massachusetts, 1987.

20. Brian W. Kernighan, Rob Pike, "The UNIX Programming Environment," Prentice Hall, New Jersey, 1984.

Appendix A. Package VHDLCAD

The following is the package VHDLCAD, which is referenced by all the VHDL files generated by the Modeler's Assistant. This package contains the type declarations for a multiple valued logic data type. It also has the the type declaration of the bus resolved data type. The bus resolution function is also defined in this package. This package is available with the Modeler's Assistant.

```
package VHDLCAD is

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

function BIT_TO_TSL(INPUT: BIT) return TSL;
function TSL_TO_BIT(INPUT: TSL) return BIT;
function BV_TO_TSL(INPUT: BIT_VECTOR) return TSL_VECTOR;
function TSL_TO_BV(INPUT: TSL_VECTOR) return BIT_VECTOR;

function BIT_TO_MVL(INPUT: BIT) return MVL;
function MVL_TO_BIT(INPUT: MVL) return BIT;
function BV_TO_MVL(INPUT: BIT_VECTOR) return MVL_VECTOR;
function MVL_TO_BV(INPUT: MVL_VECTOR) return BIT_VECTOR;
function BV_TO_INT(INPUT: BIT_VECTOR) return INTEGER;

end VHDLCAD;


package body VHDLCAD is

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


function BIT_TO_TSL(INPUT: BIT) return TSL is
   begin
      case INPUT is
         when '0' => return '0';
         when '1' => return '1';
      end case;
   end BIT_TO_TSL;
```

```
function TSL_TO_BIT(INPUT: TSL) return BIT is
  constant TIE_OFF: BIT := '1';
  begin
    case INPUT is
      when 'X' => return TIE_OFF;
      when 'Z' => return TIE_OFF;
      when '0' => return '0';
      when '1' => return '1';
    end case;
  end TSL_TO_BIT;


function BV_TO_TSL(INPUT: BIT_VECTOR) return TSL_VECTOR is
  variable TEMP: TSL_VECTOR(INPUT'RANGE);
  begin
    for I in INPUT'low to INPUT'high loop
      TEMP(I) := BIT_TO_TSL(INPUT(I));
    end loop;
    return TEMP;
  end BV_TO_TSL;


function TSL_TO_BV(INPUT: TSL_VECTOR) return BIT_VECTOR is
  variable TEMP: BIT_VECTOR(INPUT'RANGE);
  begin
    for I in INPUT'low to INPUT'high loop
      TEMP(I) := TSL_TO_BIT(INPUT(I));
    end loop;
    return TEMP;
  end TSL_TO_BV;


function BIT_TO_MVL(INPUT: BIT) return MVL is
  begin
    case INPUT is
      when '0' => return '0';
      when '1' => return '1';
    end case;
  end BIT_TO_MVL;


function MVL_TO_BIT(INPUT: MVL) return BIT is
  constant TIE_OFF: BIT := '1';
  begin
    case INPUT is
      when 'X' => return TIE_OFF;
      when 'Z' => return TIE_OFF;
      when '0' => return '0';
      when '1' => return '1';
    end case;
  end MVL_TO_BIT;
```

```
function BV_TO_MVL(INPUT: BIT_VECTOR) return MVL_VECTOR is
  variable TEMP: MVL_VECTOR(INPUT'RANGE);
  begin
    for I in INPUT'low to INPUT'high loop
      TEMP(I) := BIT_TO_MVL(INPUT(I));
    end loop;
    return TEMP;
  end BV_TO_MVL;



function MVL_TO_BV(INPUT: MVL_VECTOR) return BIT_VECTOR is
  variable TEMP: BIT_VECTOR(INPUT'RANGE);
  begin
    for I in INPUT'low to INPUT'high loop
      TEMP(I) := MVL_TO_BIT(INPUT(I));
    end loop;
    return TEMP;
  end MVL_TO_BV;


function BV_TO_INT(INPUT : BIT_VECTOR) return INTEGER is
  variable SUM : INTEGER := 0;
  begin
    for N in INPUT'low to INPUT'high loop
      if INPUT(N) = '1' then
        SUM := SUM + (2**N);
      end if;
    end loop;
    return SUM;
  end BV_TO_INT;


end VHDLCAD;
```

Appendix B. Users Manual

This users manual describes installation procedures and the usage of the Modeler's Assistant. It is divided into four sections. The first section discusses the system requirements, the second section is an installation guide, the third section discusses the usage of the tool and the fourth section is a sample session with the Modeler's Assistant.

System Requirements

The Modeler's Assistant was developed on an Apollo DN 3500 running the Domain operating system version SR 10.2. The graphics were created using the X Window System version 11 revision 4 (X11 R.4). The Modeler's Assistant can be compiled on any UNIX workstation running X11 R.4. To compile the Modeler's Assistant requires the Xlib library, as well as the Xt toolkit library, which is a standard library that is part of the X11 R.4 distribution. The widget set used is the Andrew widget set developed at the Carnegie Mellon University. The Andrew widgets too are part of the X11 R.4 distribution available from MIT. The X11 R.4 distribution can be obtained by anonymous *ftp* from export.lcs.mit.edu (18.30.0.238). It is possible to change the widget from the Andrew widgets to Open Software Foundation's Motif

widget set. The changing of widget sets is straightforward but would involve modification of the source code.

Hence the requirements for running the Modeler's Assistant are a UNIX workstation running X11 R.4 and the Xaw, Xmu, Xext, Xt and the Xlib libraries.

Installation Guide

To install the Modeler's Assistant, place the files on the distribution media in a directory on the system that has all the requirements listed in the previous section. The distribution files include a *Makefile* and invoking the *make* utility would compile the software and produce an executable file, *vcad*. This file should be placed in a directory where it can be accessed by all the users. Next the X resource file *Vcad* (note capitalization) should be copied from the distribution directory to the directory */usr/lib/X11/app-defaults*. At this stage the software would be ready to use.

Usage of the Modeler's Assistant

The Modeler's Assistant can be invoked by typing *vcad*. This would place the main window of the Modeler's Assistant on the screen. The window includes an on-screen menu, a drawing area, a message and prompt display line, and an input box. The Modeler's Assistant can be manipulated by picking various items from the menu list. A

brief description of the most important menu items follows. The *Backup* menu item found in most menus is used to move back up in the menu tree.

MAIN

The *MAIN* menu contains three items, *Create*, *Edit*, *Debug* and *Quit*. *Create* and *Edit* are used to create new processes (also referred to as modules), or units, or to edit previously created modules and units. The *Debug* menu item is for debugging the software and will only be of use to someone involved in further developing the software. Picking *Quit* will quit the program and will make the main window disappear.

MAIN - CREATE / EDIT

The *CREATE* menu and the *EDIT* menu give two options each, to either create or edit a *Unit* or to create or edit a *Process*. Each of these produces a relevant menu. Picking *Unit* or *Process* in the Edit menu will make the software prompt the user for the name of the file that contains the information on the Unit or Process. Incorrect information will cause an error message to appear in the error/prompt display box.

MAIN - CREATE / EDIT - PROCESS

The *MODULE* (or *PROCESS*) menu contains *Add*, *Delete*, *Change*, *Move*, *Specify*, *VHDL Dump*, *Show VHDL*, *Save*, *Primitive* and *Refresh* menu items. *Add* allows the addition of process ports, variables, constants and generics. It also lets the user change the sensitivity of a process port. *Delete* and *Change* let the user delete or change any of the process ports, variables, constants and generics in the process. *Move* lets the user move the graphical tokens on the screen. Picking *Specify* pops up a window on the screen where the user can specify the functionality of the process. *VHDL Dump* dumps the VHDL code of the process in a file and *Show VHDL* pops a window on the screen and displays the VHDL code of the process on the screen. *Save* saves the process. The menu item *Primitive* is used to convert the process into a primitive process. *Refresh* refreshes the screen.

MAIN - CREATE / EDIT - PROCESS - ADD

The *ADD* menu option in the *Module* menu gives the user a menu with *Port*, *Sense*, *Variable*, *Constant* and *Generic* menu items. *Port*, *Variable*, *Constant* and *Generic* let the user add the respective constructs by first prompting the user for the name of the construct and then letting the user pick a data type from a menu. The data types offered for variables and constants are Bit, MVL, Boolean, Integer, Real, Bit_Vector and MVL_Vector. The data types offered for process ports include TSL and TSL_Vector in addition to the above list. The data types MVL, MVL_Vector, TSL and TSL_Vector are declared in the VHDLCAD package (see Appendix A). TSL and

TSL_Vector are bus resolved data types and signals of this type are the only ones that will permit a signal assignment from inside more than one process. The data types offered for generics are Boolean, Integer, Real and Time. *Port*, *Variable* and *Constant* also prompt the user for a location to place the construct on the screen. *Sense* lets the user toggle the sensitivity of a process port.

MAIN - CREATE / EDIT - PROCESS - SPECIFY

The *Specify* menu option is used to specify the functionality of the process. Picking it, results in a window with a text editor popping up on the screen where the VHDL code of the functionality of the process, that is the VHDL code that would come between the *begin* and *end process* statements, should be entered. Information in a text file can be imported into the editor by pressing Control-I (^I). This would pop up a small dialog box where the file name can be entered. The names of the process ports, variables, constants and generics used in the functionality should be the same as those defined earlier when creating the graphics. The text of the functionality is parsed to extract mode information for the process ports. Since this text is parsed only when the user closes the popped up window after completing the specification, the graphics of the process ports should be created before beginning the specification of functionality. As a safety rule, the specification of the functionality should always be the final step while creating a process. If a previously created process is being edited, the editor window will pop up containing the functionality specified for the original process.

MAIN - CREATE / EDIT - PROCESS - PRIMITIVE

The *Primitive* menu option is used to convert the process into a process primitive. A process primitive contains the entire process with generic names for the process ports and generics, and text lines that are to be displayed as prompts in the message box when prompting the user for the real names of the process ports and generics. Before picking the *Primitive* menu option, a complete and error free process should be created. The process can be checked by using the *Analyze* menu option in the *UNIT* menu. When the process is ready, picking the *Primitive* menu item pops up a window with a text editor where the prompts can be specified. The prompts can each only be one line long. The prompts should be typed after the name of the process port or generic, with a colon (:) used to separate the two. Each name-prompt combination should appear on a separate line, and there should be prompts specified for all the process ports and generics. Correctly specifying all the prompts will create a process primitive. This primitive is placed in the users default library. After the primitive is thoroughly tested, it can then be moved to a process primitive library directory where it can be accessed by the entire design group.

MAIN - CREATE / EDIT - UNIT

The *UNIT* menu contains *Add*, *Delete*, *Change*, *Move*, *VHDL Dump*, *Show VHDL*, *Analyze*, *Save* and *Refresh* menu items. The *Add* menu item lets the user add processes, signals and primitives to the unit. *Delete* and *Change* let the user modify the processes and signals in the unit. *Move* lets the user move processes on the screen, by

picking the process graphic and placing it at a different point on the screen. *Move* also moves any signals connected to processes. *VHDL Dump* lets the user dump the VHDL code for the entire process onto a file. *Show VHDL* is used to display the VHDL code generated in a window on the screen. *Analyze* lets the user analyze the VHDL code in produced by the Modeler's Assistant, provided the user has specified the analyzer to be used. The *Save* menu item saves the unit to a file which can later be edited. *Refresh* refreshes the graphics displayed on the screen.

MAIN - CREATE / EDIT - UNIT - ADD

The *ADD* menu contains three options, *Process*, *Signal* and *Primitive*. The *Process* option is used to add a process to the unit. It prompts the user for the name of the file containing the process and for the location on the screen to place the process. The *Signal* option lets the user add a signal between process ports on two processes. The user has to first choose the source process, then the process port in the process, then the destination process and finally the process port in the destination process. If required the user will be prompted for the name to be given to the signal. The signal is added only if it does not create an error. Some of the errors checked for include, type mismatch between the source process port and the destination process port, incorrect modes of process ports (for example source process port cannot be of mode *in*), and multiple inputs to a process port not of a bus resolved type. In a completed unit, process ports that are not attached to a signal are assumed to be entity ports, and two such process ports of the same name and data type are assumed to be the same entity

port. The *Primitive* menu item lets the user add a primitive process to the unit being created.

MAIN - CREATE / EDIT - UNIT - ADD - PRIMITIVE

The *Add-Primitive* menu option can be used to add a primitive from the primitives library to the unit. The primitives library is a directory containing the files of all the primitive processes. This directory is specified through an environmental variable "VCADPRIMDIR" which has to be set before starting the program. The environmental variable must contain the full path name of the primitives directory. For example if the primitives directory was */usr/vhdlcad/primitives*, the environment variable would be set as follows:

% setenv VCADPRIMDIR /usr/vhdlcad/primitives

If the process primitive directory is not specified, the default directory is taken to be the primitives directory.

Picking the *Primitive* menu option pops up a window with a menu containing the names of all the process primitives found in the primitives directory. Once a primitive is picked, the user will be prompted for the real names for all the process ports and generics, after which the primitive is placed on the screen and a new process is included in the unit.

MAIN - CREATE / EDIT - UNIT - ANALYZE

The *ANALYZE* menu option is used to analyze the VHDL code generated by the Modeler's Assistant. The Modeler's Assistant can use any analyzer available on the system. To specify the analyzer to be used, the user should set the UNIX environment variable "VCADANALYZER" to contain the full path name of the analyzer program, before calling up the *vcad* program. For example to use the Synopsys VHDL analyzer ((c) 1990 Synopsys Inc.) *vhdlan*, found in the directory */usr/synopsys/simulation/vhdl/bin* the environment variable should be set as follows:

% setenv VCADANALYZER /usr/synopsys/simulation/vhdl/bin/vhdlan

Now using the *Analyzer* menu option would pop up a window and the output of the analyzer would be displayed in it. The analyzer can be killed by pressing the *Cancel* button in the pop-up window. Before analyzing any VHDL code generated by the Modeler's Assistant, the package VHDLCAD must be analyzed, since it is referenced by all VHDL programs created by the Modeler's Assistant.

Sample Session with the Modeler's Assistant

The following is a sample session with the Modeler's Assistant where a simple model is created. Window dumps and screen dumps show all the steps involved in

creating the model. The model contains two processes, an inverter and an oscillator, with the output of the oscillator connected to the input of the inverter.

The Modeler's Assistant can be invoked by typing *vcad* at the UNIX shell prompt. Figure 8 shows the window that is first displayed. The first step in creating the entire process model graph (or unit), is to create the individual processes. Figure 9 shows the window when *Create* is picked in the *MAIN* menu. To create the model described above two processes, one for the oscillator and one for the inverter, would need to be created. On picking *Process* in the *CREATE* menu, the software displays a prompt to enter the process name where the name "OSCL" for the oscillator process is input, as shown in figure 10. Note that when a text input is required the input box gets activated.

A circle representing the process is displayed on the window as in figure 11. For this process to model an oscillator, it would need two process ports, an enable input and a clock output. To add a process port the menu option *Add* is picked, and the window would look like figure 12. To add a process port for the clock output, the menu item *Port* is picked and in response to the prompt to specify the name of the port, the name "CLK" is entered. Figure 13 shows the name of the port being entered. Next the data type of the port is to be picked from a menu. The menu of the data types is in figure 14. After picking the data type *Bit*, the location of the process port needs to be specified. Figures 15 and 16 show the location of the process port being specified. Since this process is to be a continuous oscillator, the output CLK, also needs to be in the sensitivity list of the oscillator process. To toggle the sensitivity, the menu item *Sense* is picked as shown in figure 17. Figure 18 shows the process port "CLK",

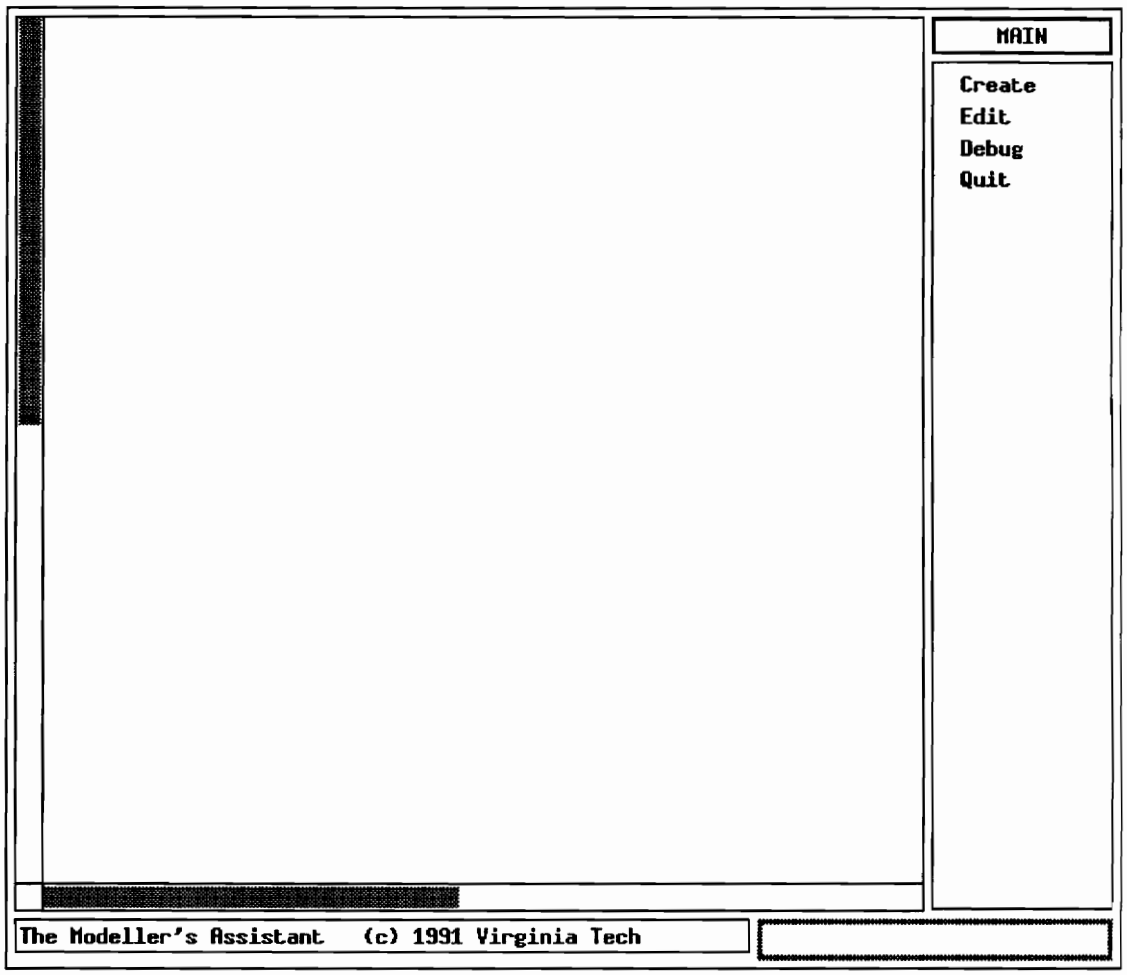


Figure 8. Starting window of the Modeler's Assistant

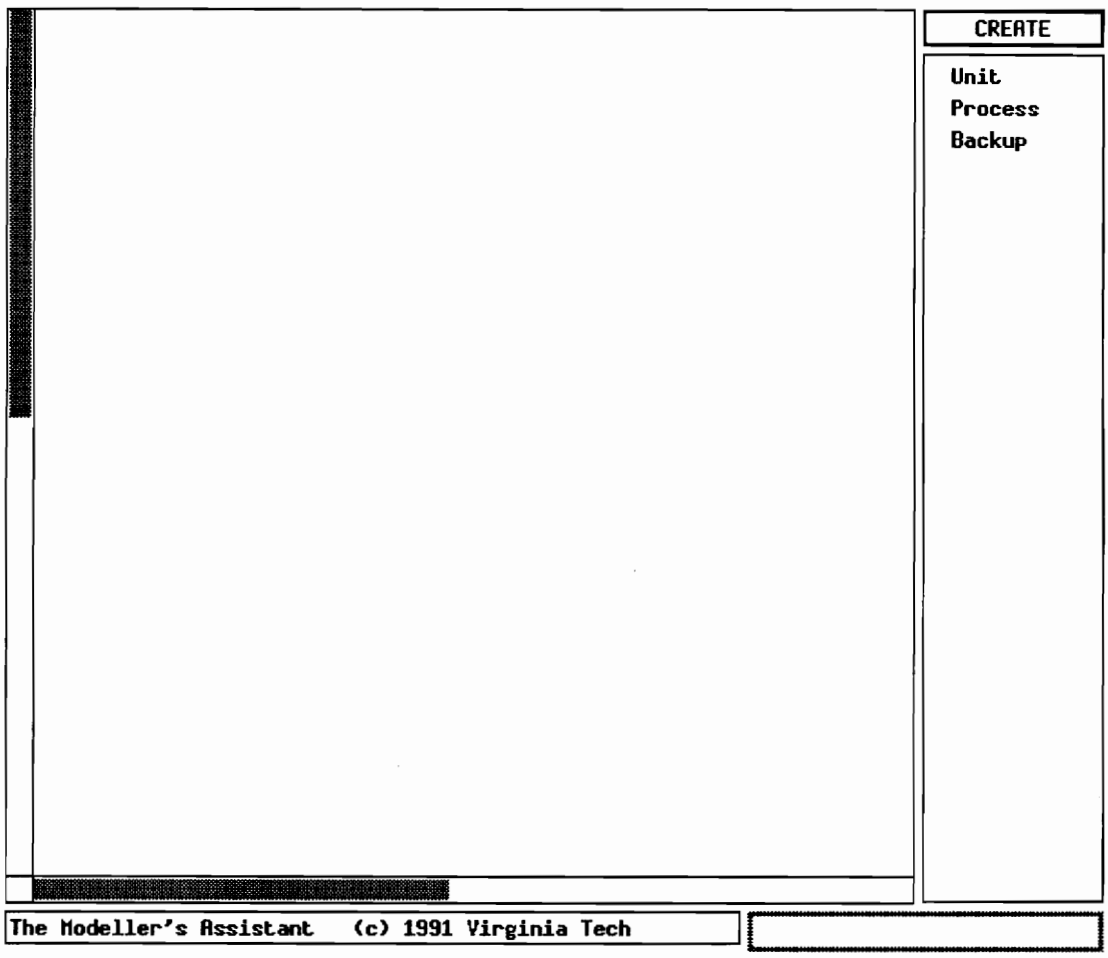


Figure 9. ***Create*** **menu window**

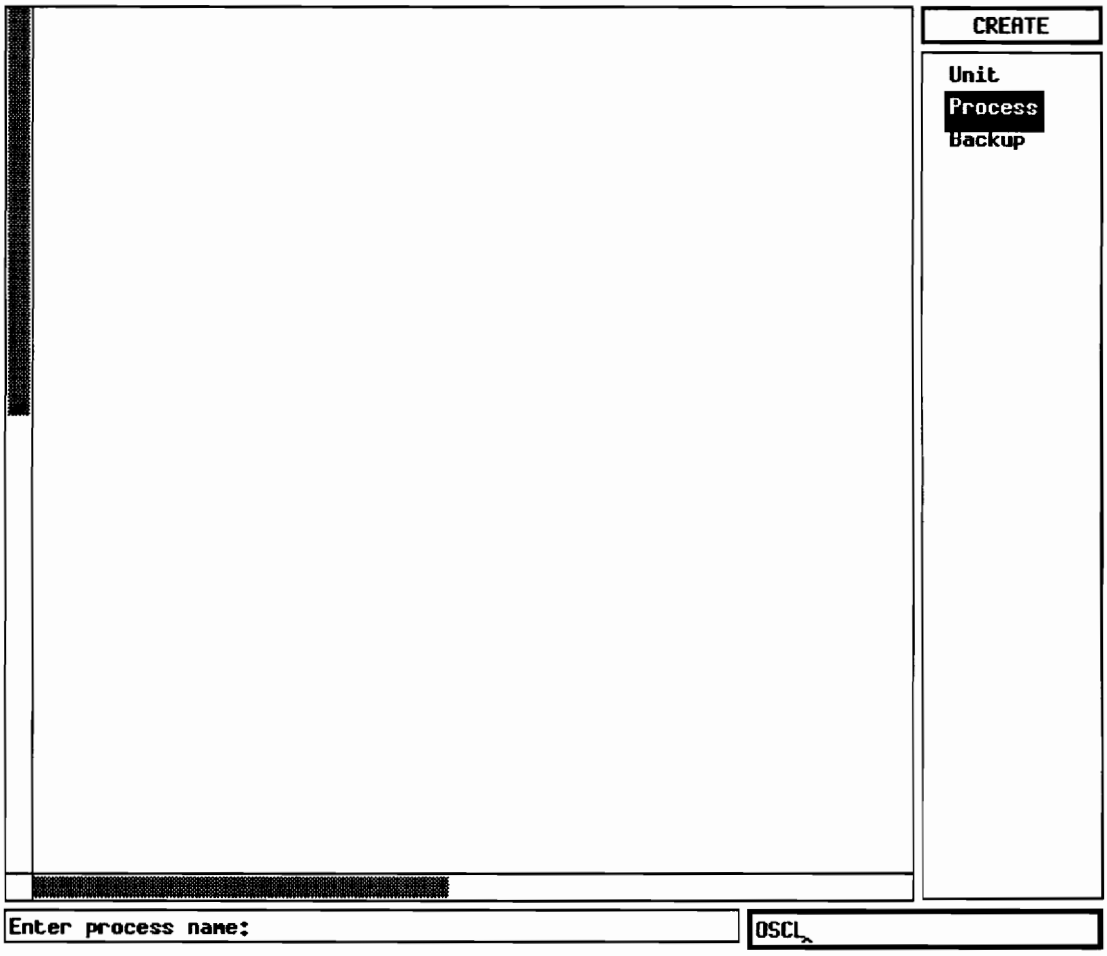


Figure 10. Starting creation of process "OSCL"

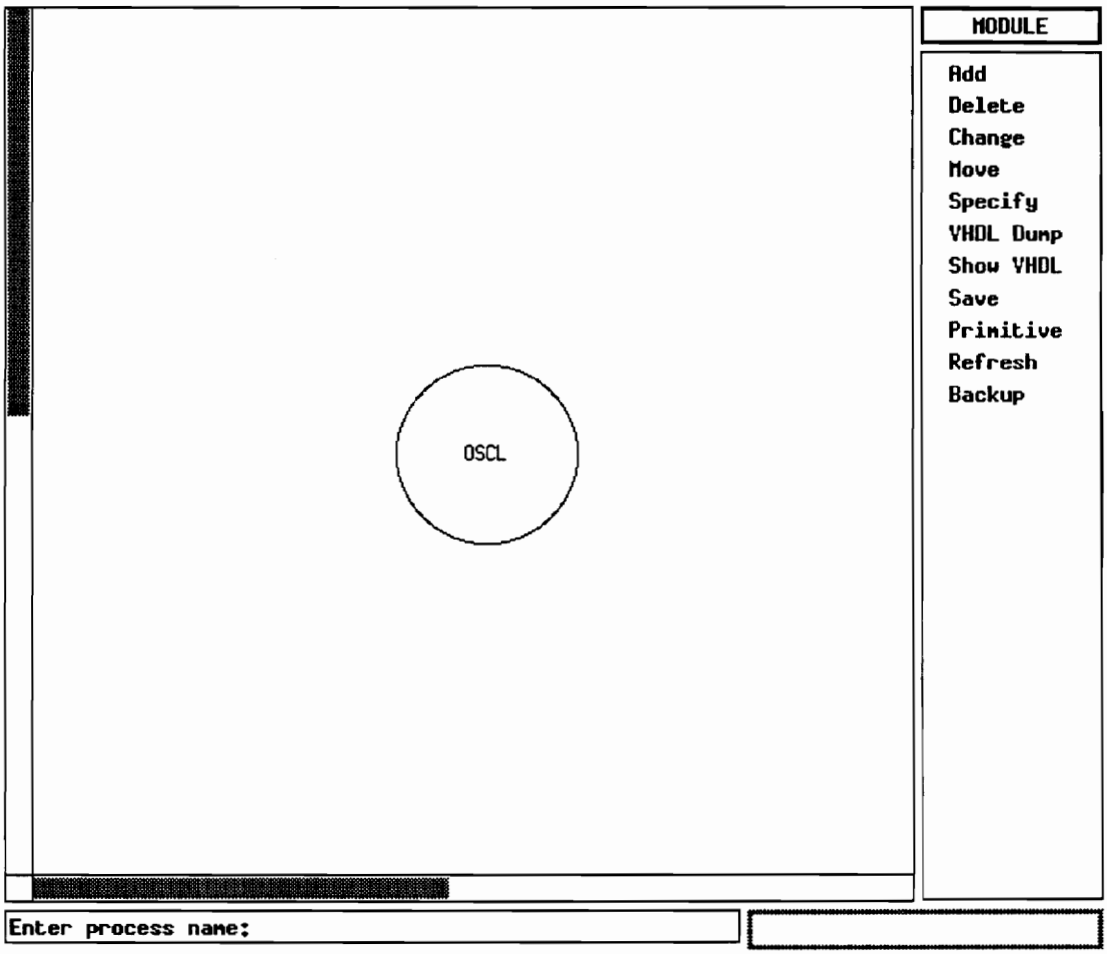


Figure 11. Starting representation of process "OSCL"

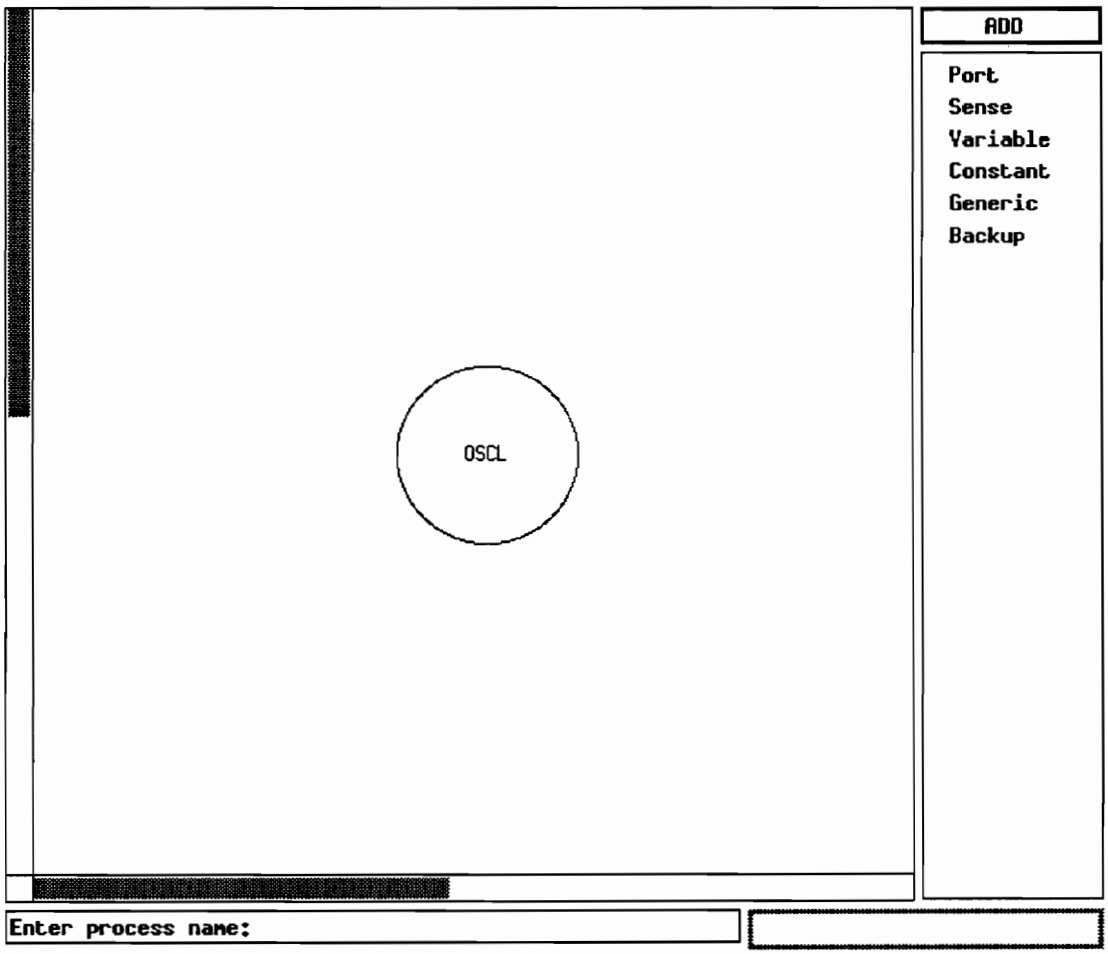


Figure 12. ***ADD*** **menu window**

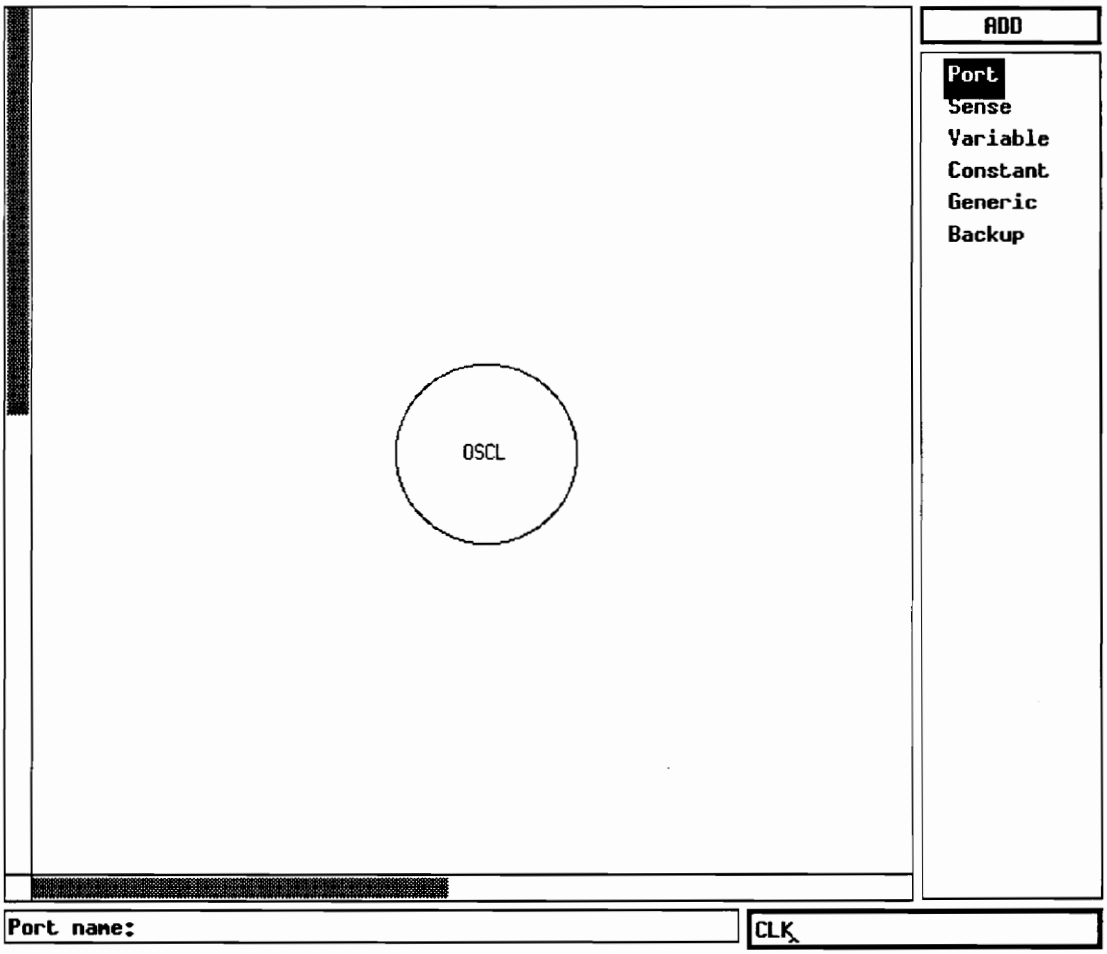


Figure 13. Adding process port "CLK"

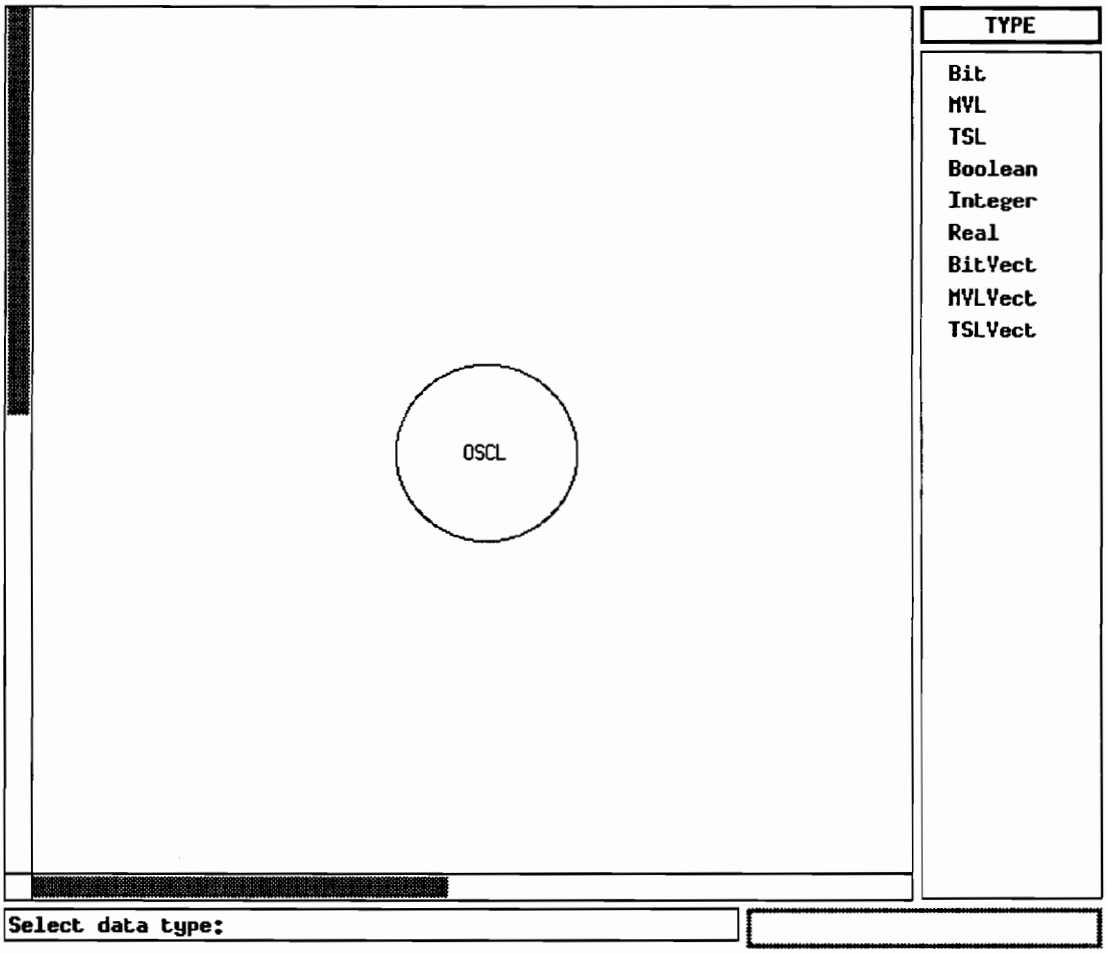


Figure 14. Selecting data type for process port "CLK"

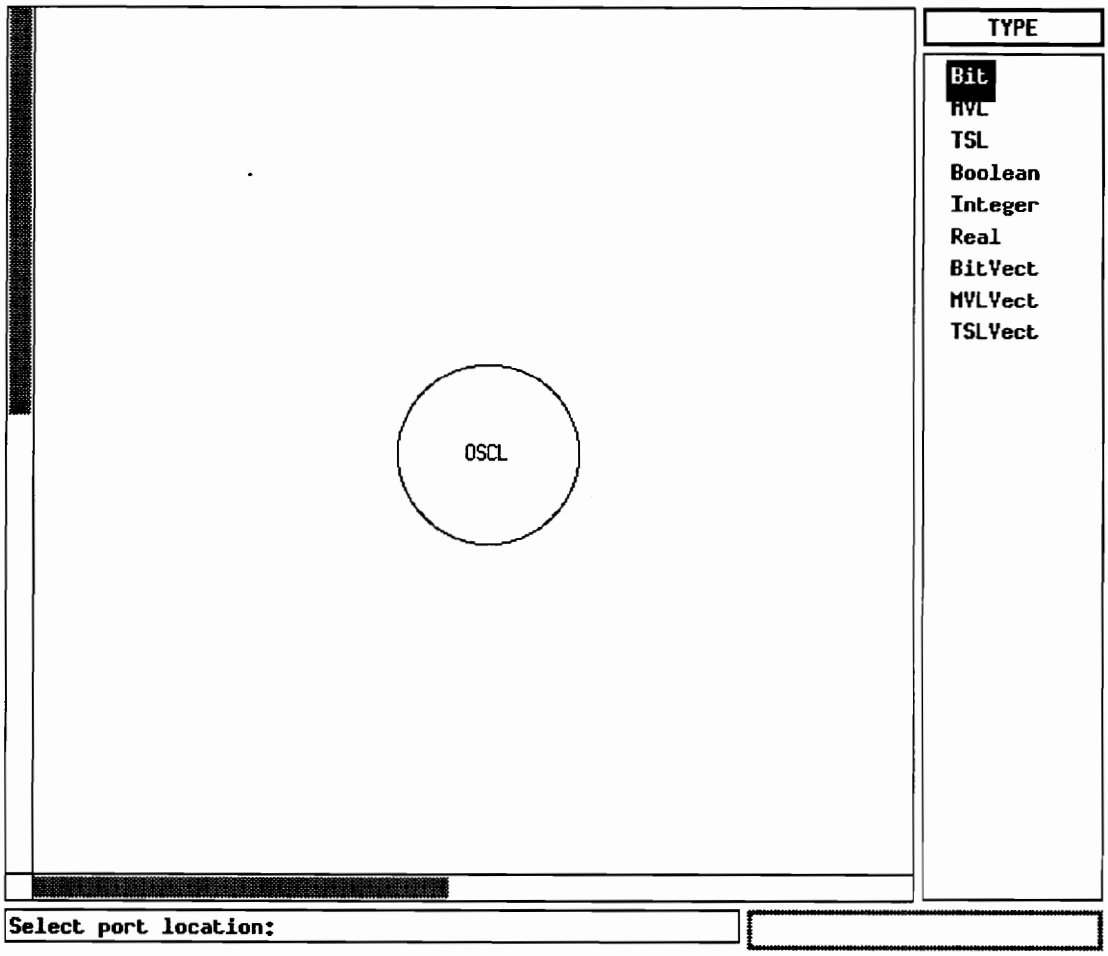


Figure 15. Selecting location of process port "CLK"

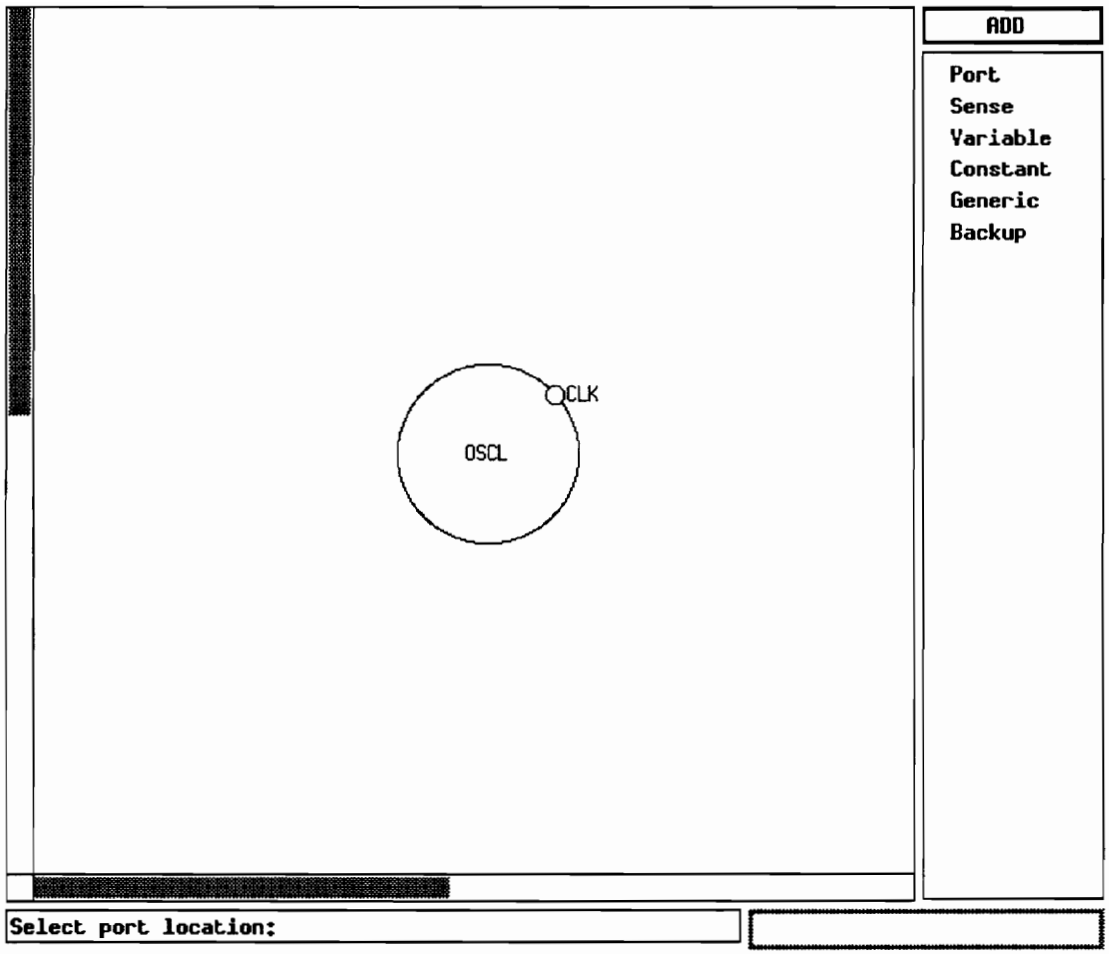


Figure 16. Process with process port "CLK"

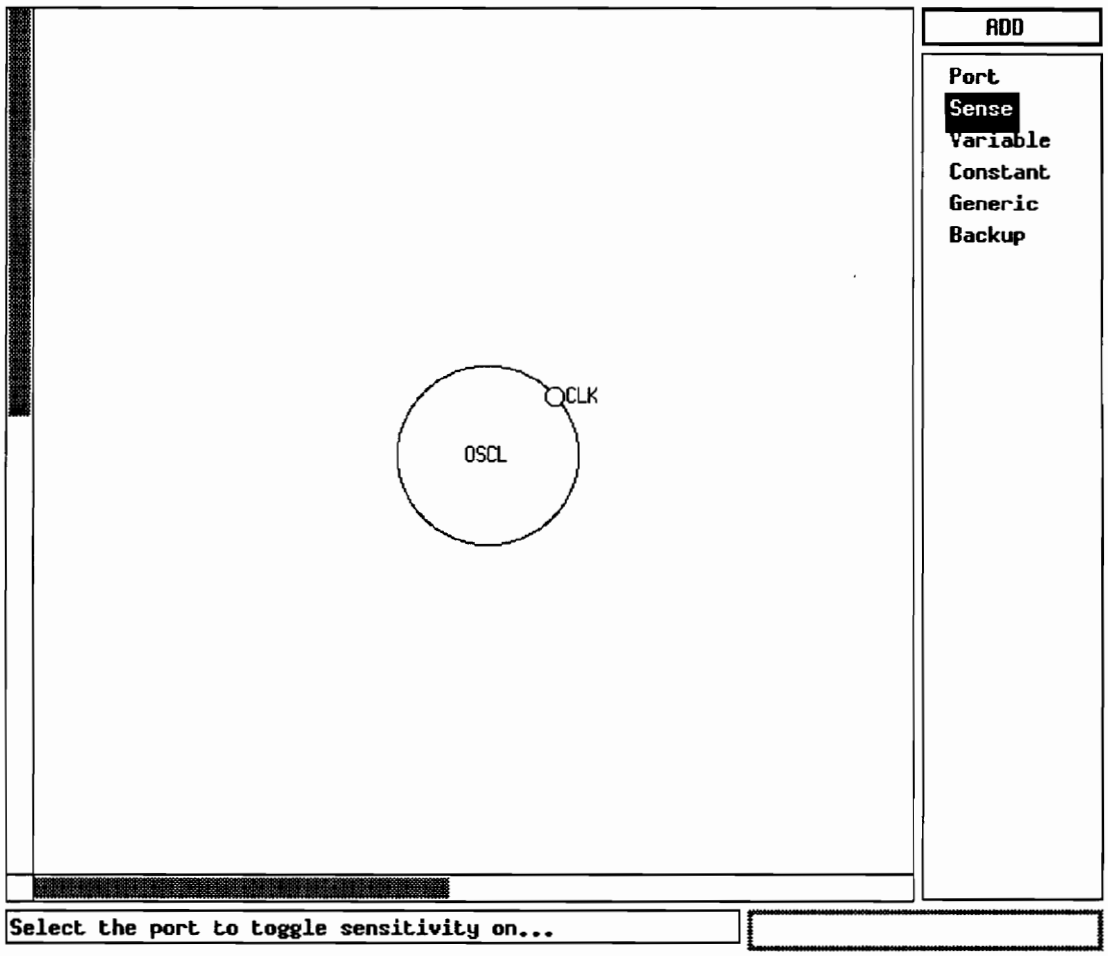


Figure 17. Toggling sensitivity of process ports

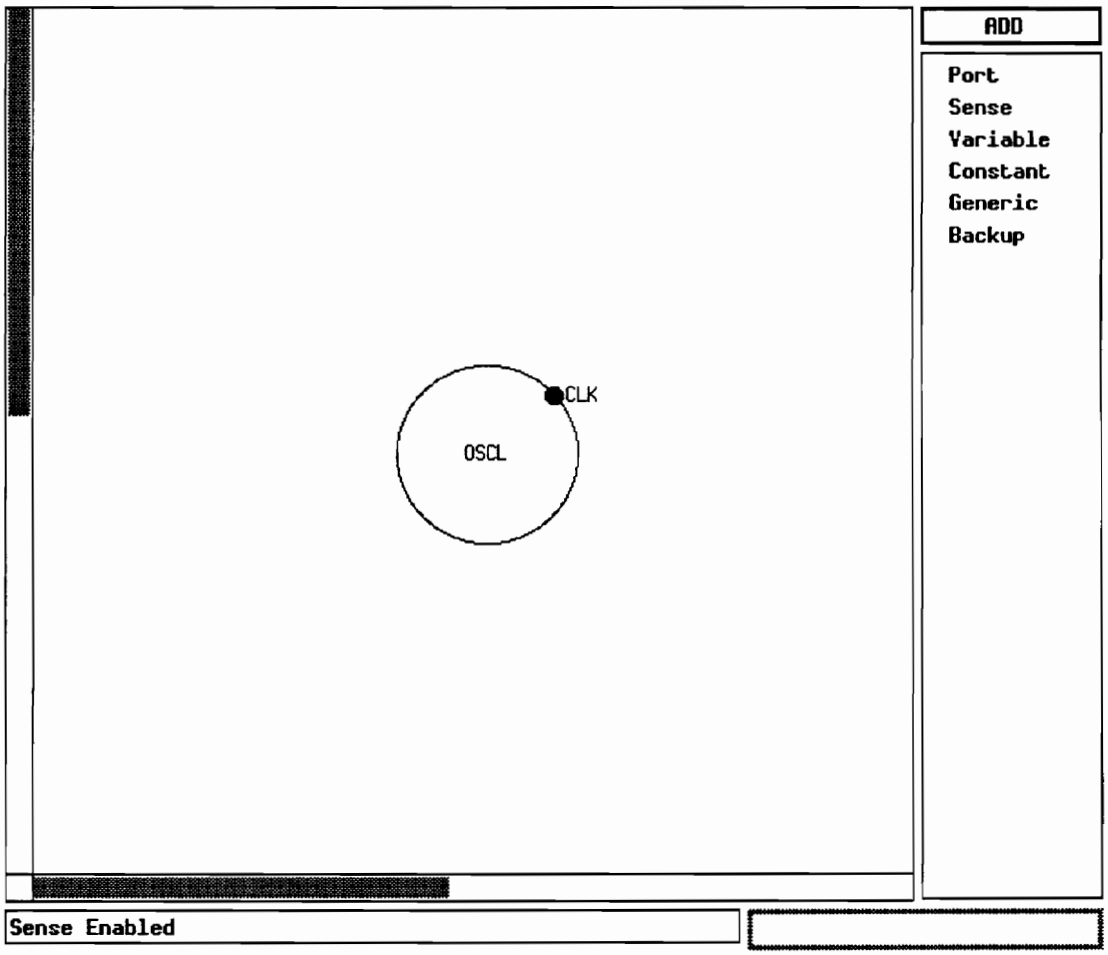


Figure 18. Process with sensitive process port "CLK"

represented with a filled circle implying that it is in the sensitivity list of the process. Similarly a second sensitive port of type *Bit* needs to be added that can trigger the oscillator. Figure 19 shows the process with both ports. Next a generic needs to be added that would specify the clock period of the oscillator. This is done as shown in figure 20, by picking the *Generic* menu item, entering the name of the generic and then picking the data type of the generic from a menu. The menu with the data types available for generics is shown in figure 21. Since the generic is to be the clock period of the oscillator, the data type *Time* is picked.

Now the functionality of the process needs to be specified. This is done by picking the *Specify* menu option in the *MODULE* menu and typing the VHDL functionality of the process. Figure 22 shows the functionality being specified. The functionality can also be specified by importing a previously created text file, as shown in figure 23. The dialog box is made to pop up by pressing Control-I in the editor window. After specifying the functionality, the process construction is complete.

Similarly a second process for the inverter process is created. As shown in figure 24, this process has two process ports and one generic. The processes are saved by picking the menu item *Save* in the *MODULE* menu, and specifying the file to save them to, as shown in figures 25 and 26. If the process "OSCL" was to be made a primitive, the *Primitive* menu would be picked and the prompts would be specified. The creation of a process primitive is shown in figure 27.

After both the processes are created, a unit is created by invoking the processes. To create a unit, the *Unit* menu option in the *CREATE* menu is picked and the name

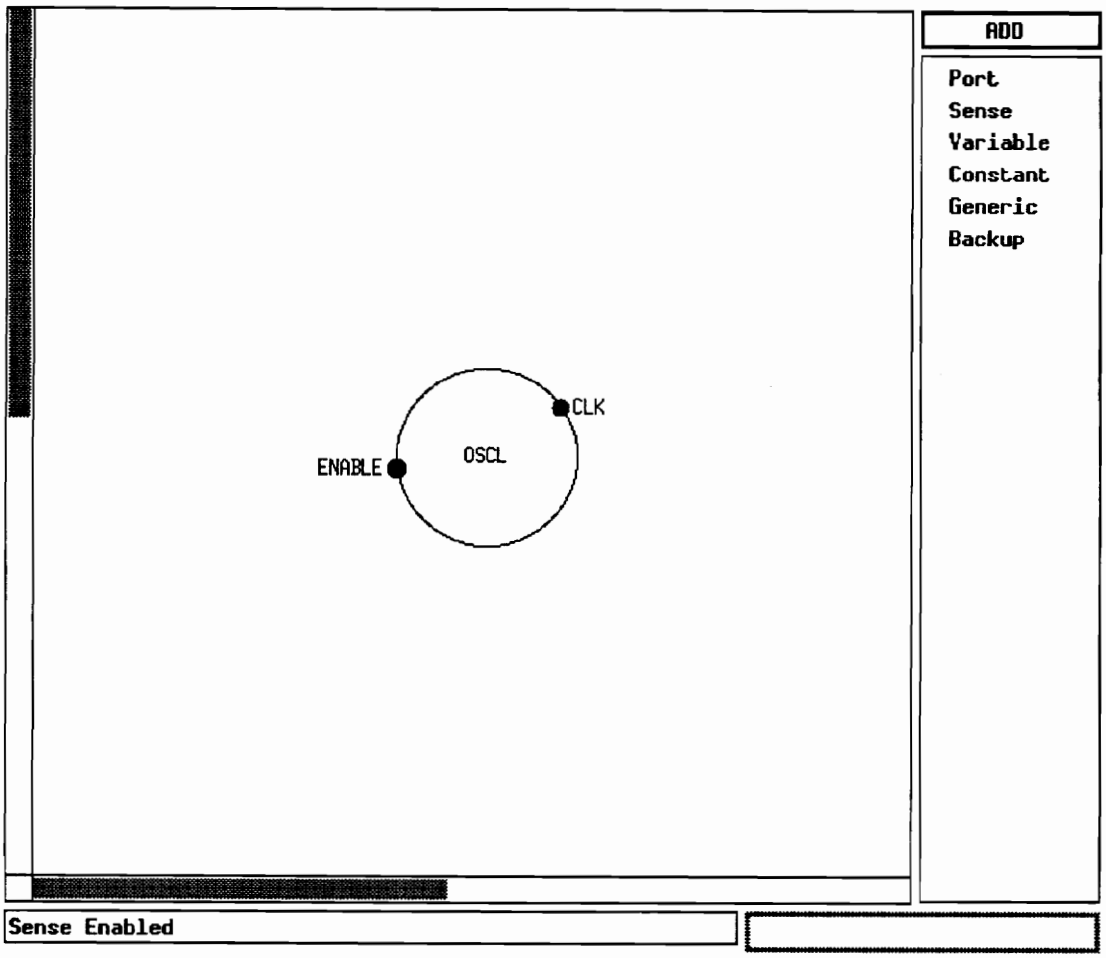


Figure 19. Process with process ports "ENABLE" and "CLK"

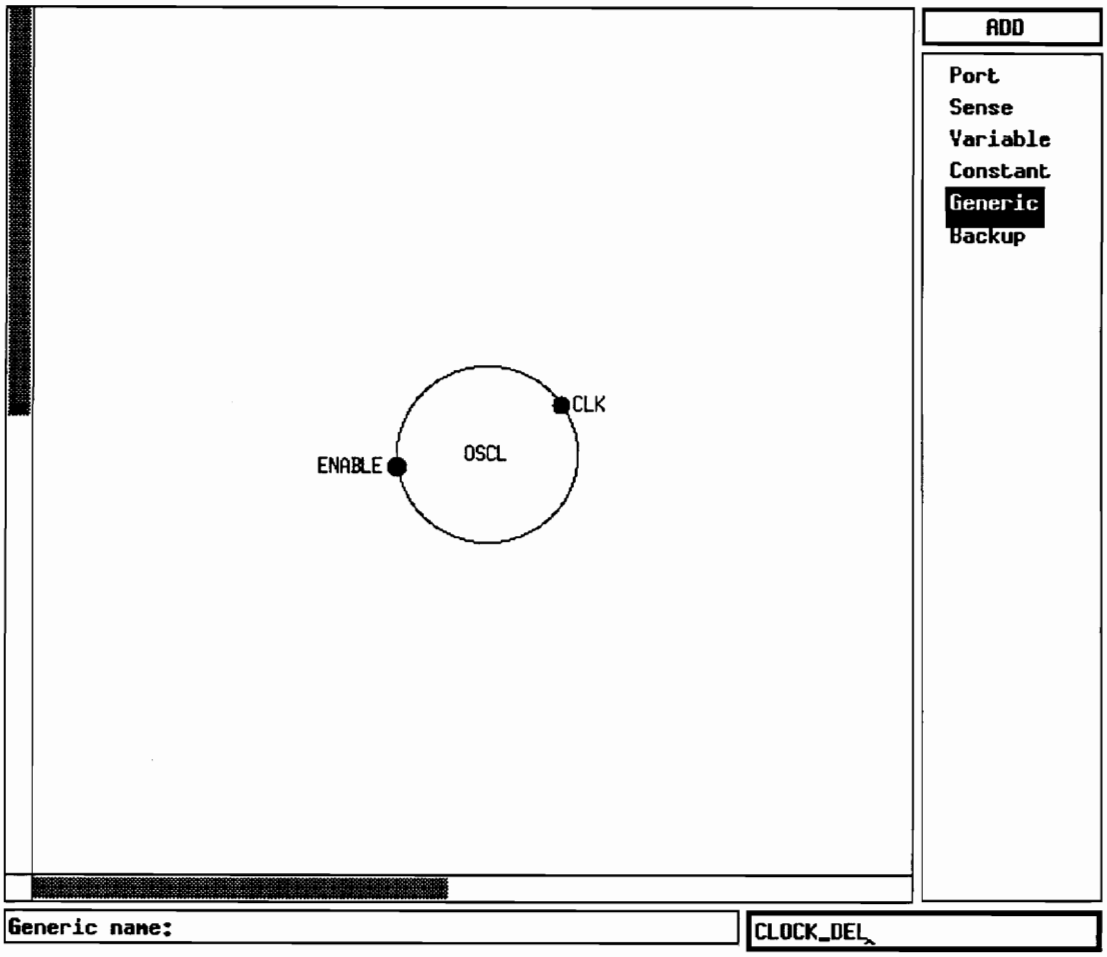


Figure 20. Adding generic "CLOCK_DEL"

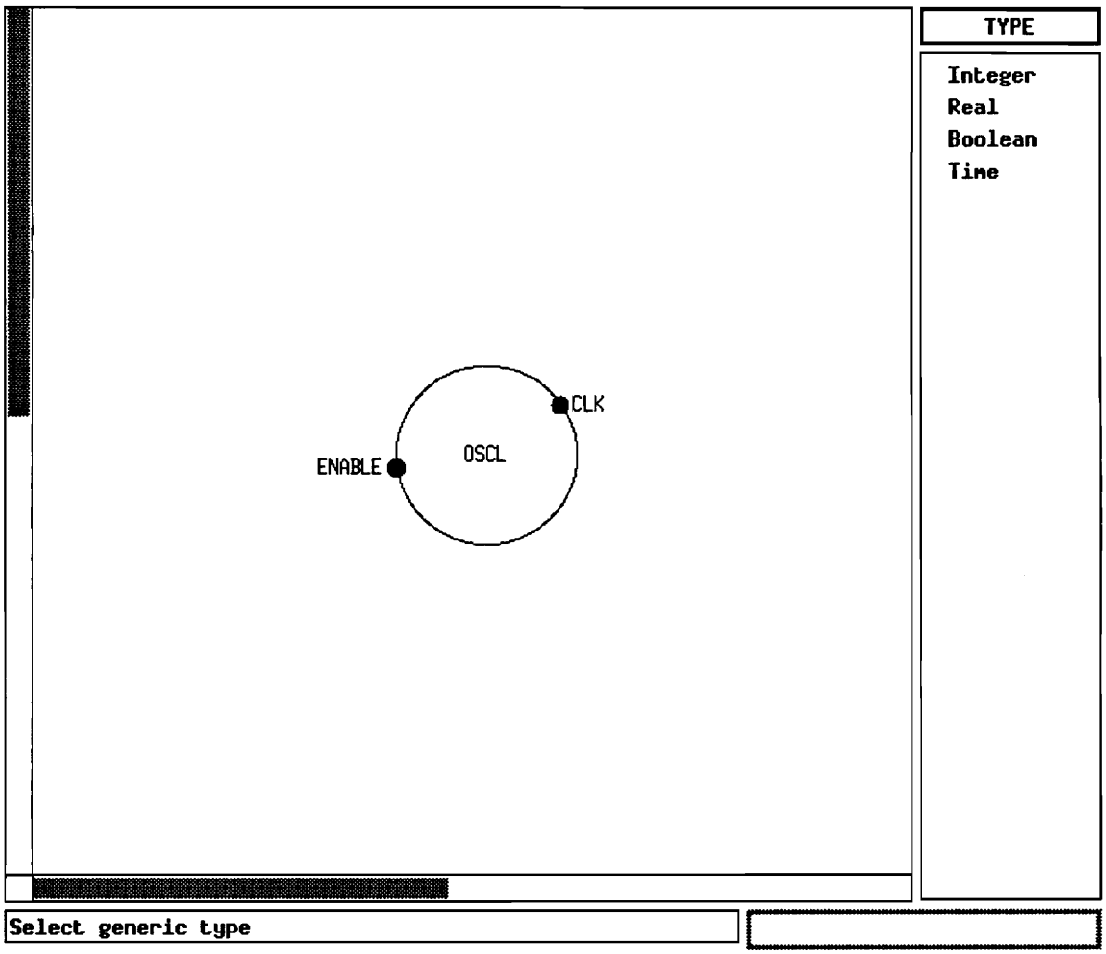


Figure 21. Selecting data type for generic

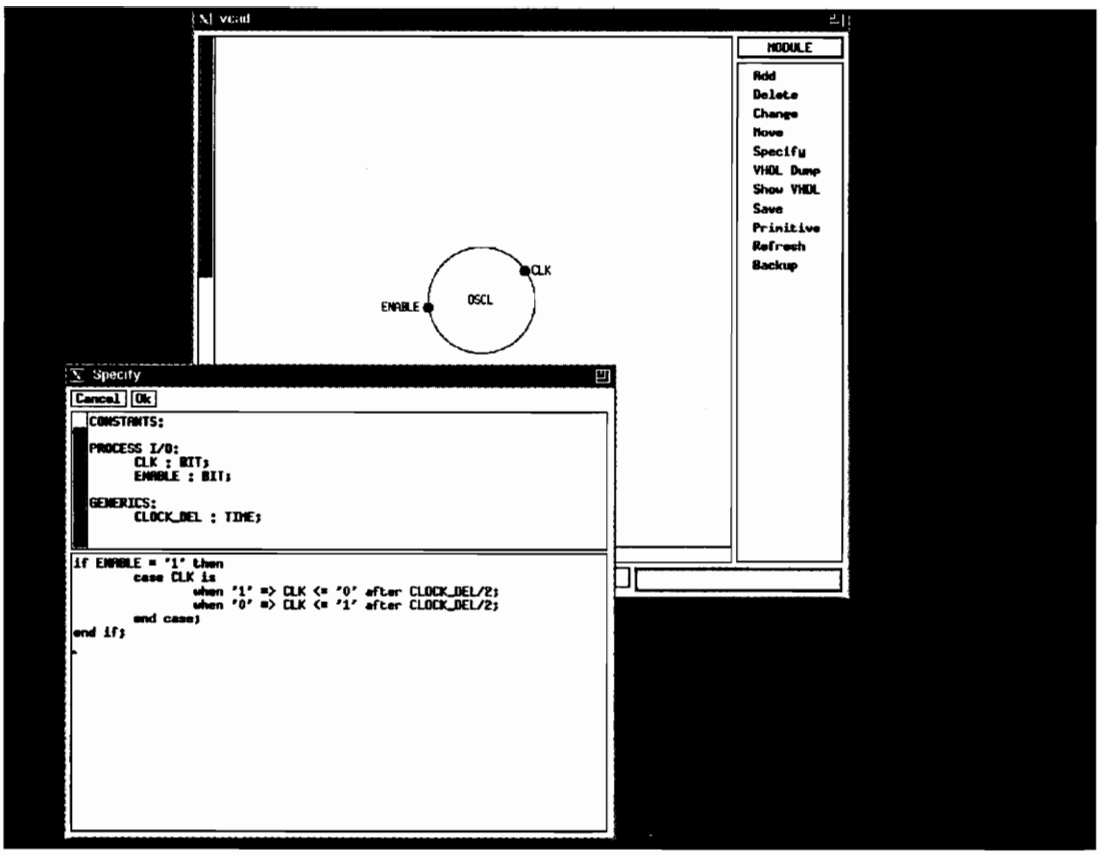


Figure 22. Specifying functionality of process "OSCL"

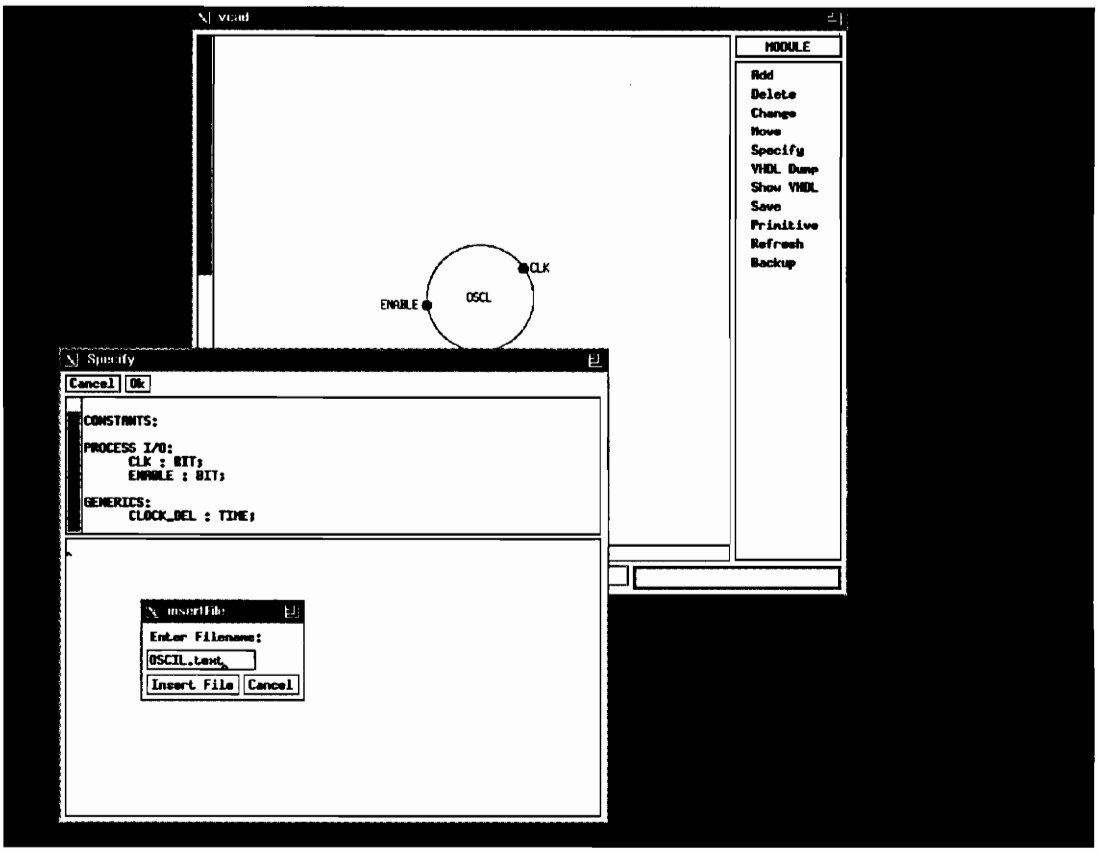


Figure 23. Importing text file of functionality

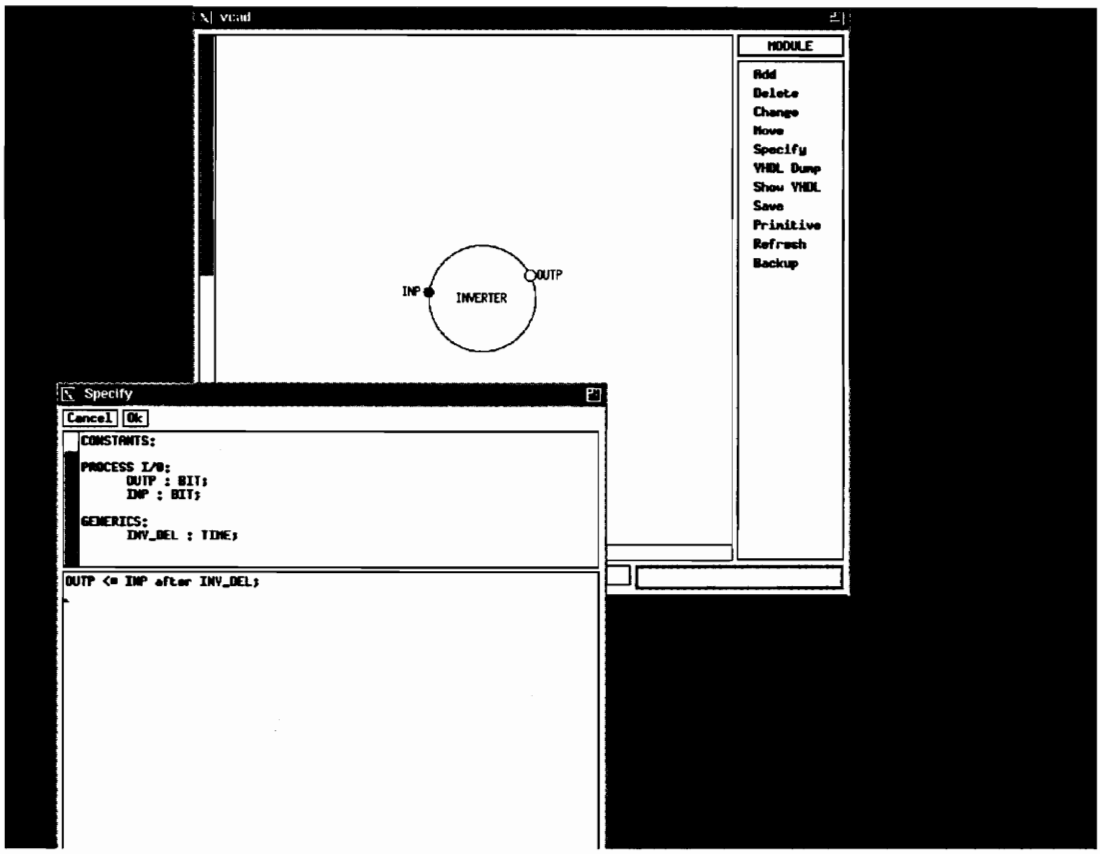


Figure 24. Building process "INVERTER"

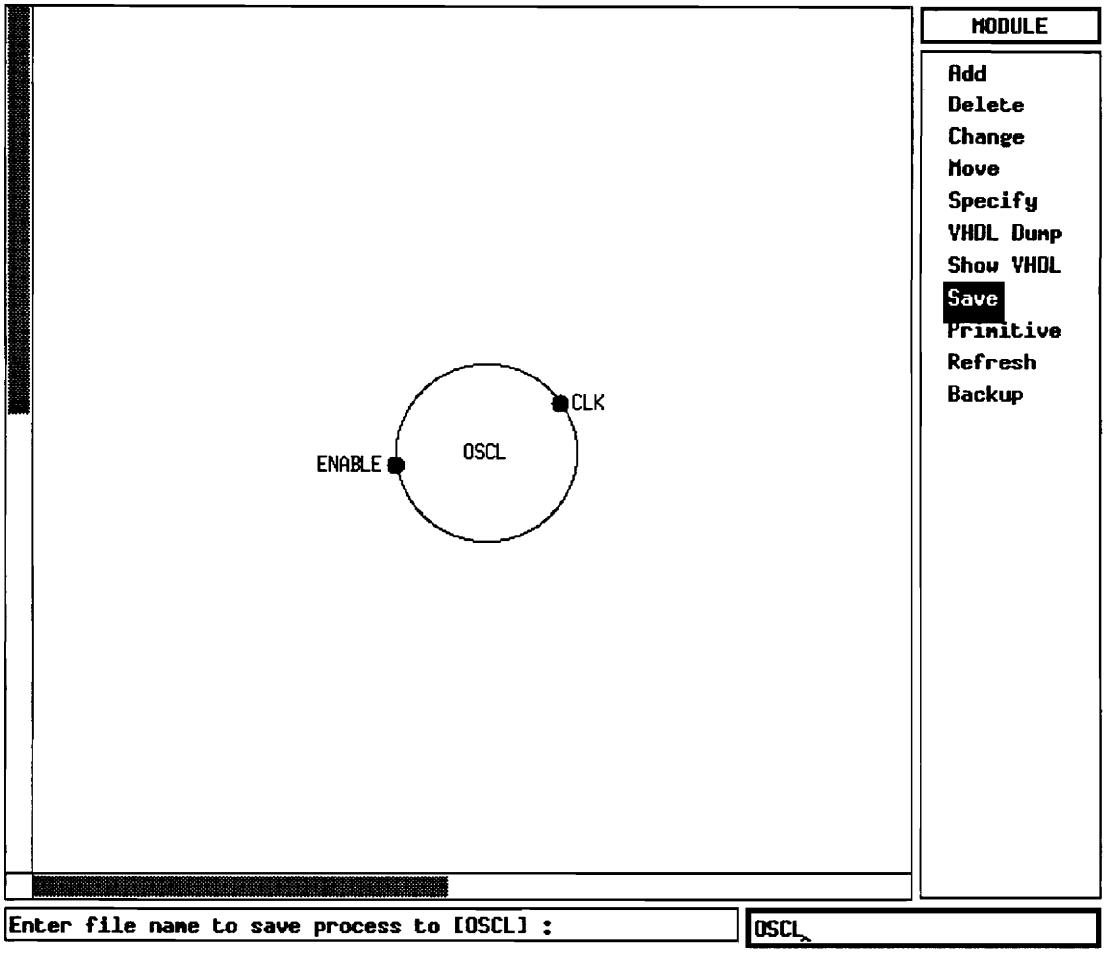


Figure 25. Saving process "OSCL"

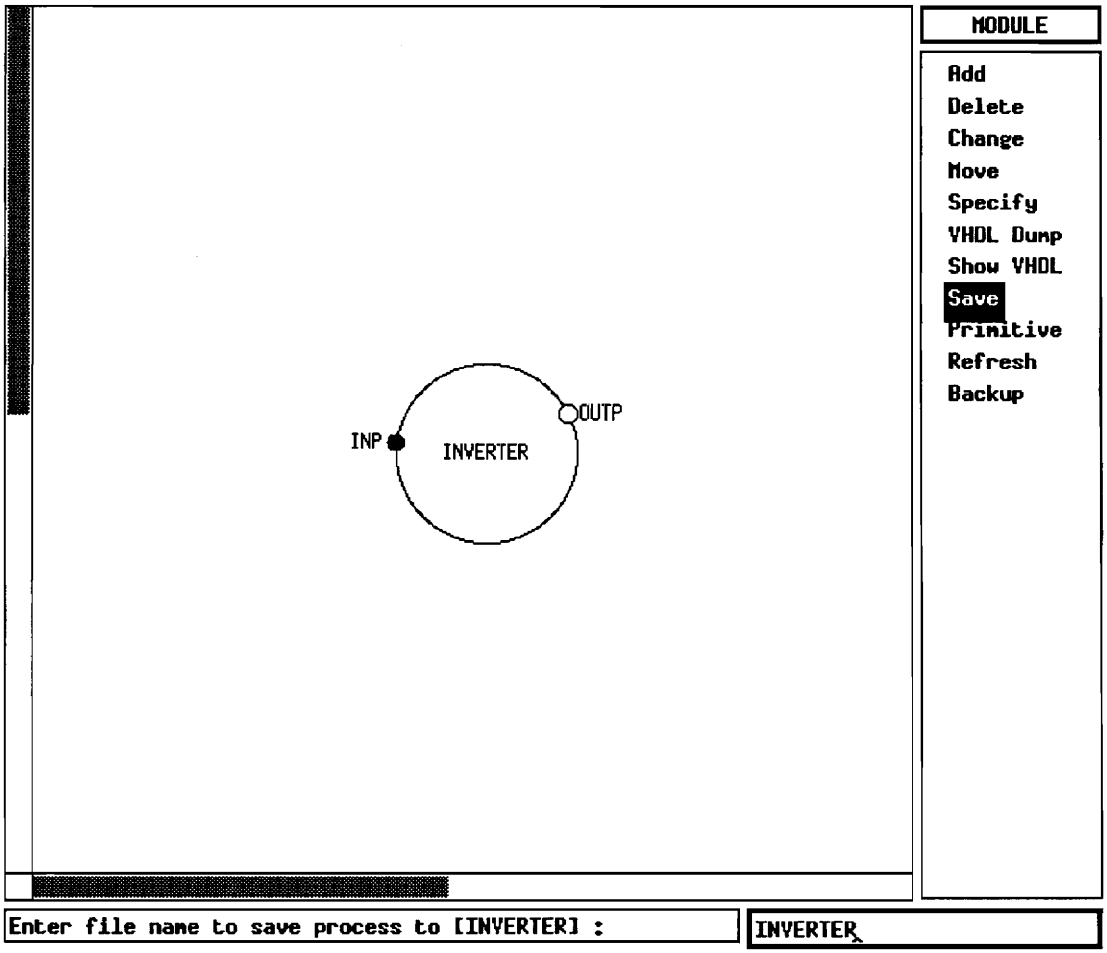


Figure 26. Saving process "INVERTER"

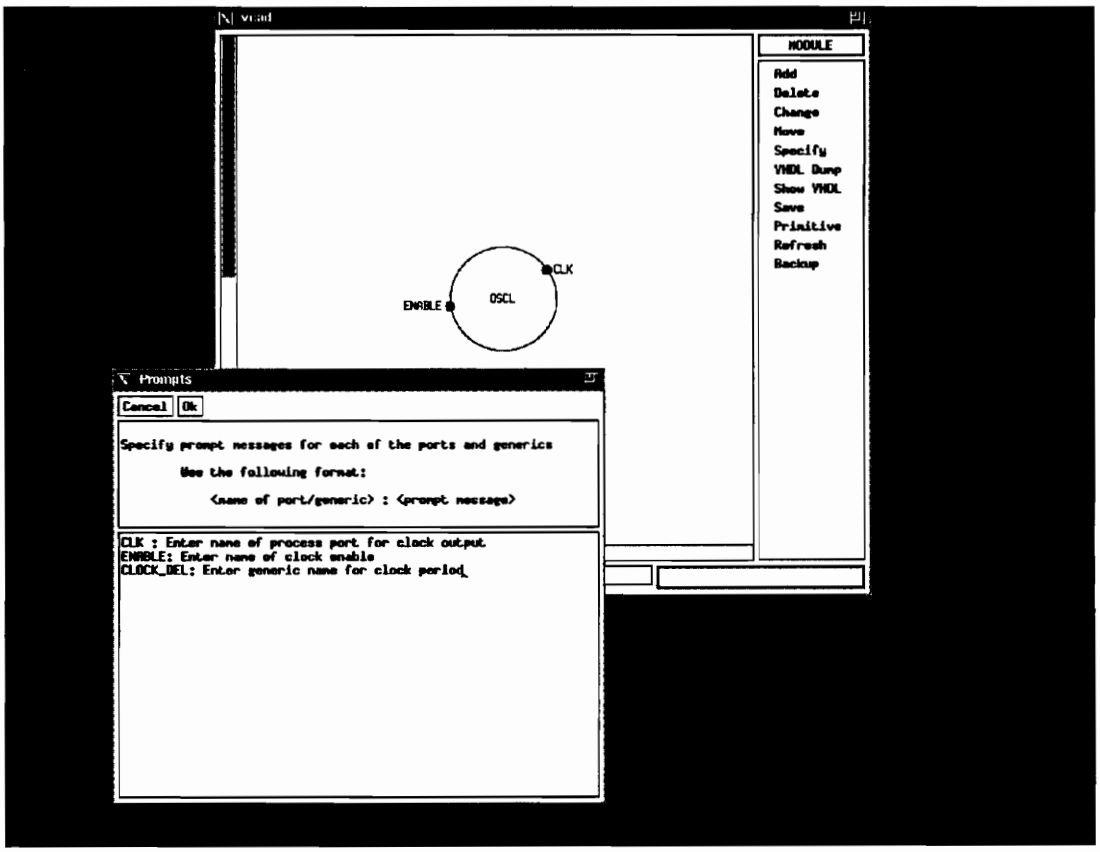


Figure 27. Converting process "OSCL" to a primitive

"OSCILLATOR" is given to the whole unit. Figure 28 shows the unit being initialized, and figure 29 shows the window with the *UNIT* menu. The two processes, "OSCL" and "INVERTER" need to be added. To add the processes, the *Add* menu option is chosen to reach the *ADD* menu. The window with the *ADD* menu is in figure 30. Figure 31 shows the process "OSCL" being added, and figure 32 shows the Modeler's Assistant prompting for the location of the process on the screen. Figure 33 shows the first process "OSCL", added to the unit. Similarly the second process "INVERTER", is added. Figure 34 shows both processes on the screen. Next a signal needs to be added between "CLK" and "INP". Figures 35 and 36 show the user being prompted for the source process and the process port in the process. The user is similarly prompted for the destination process and the process port in the destination process. Figure 37 shows the signal name being specified.

If the unit is to contain a primitive, clicking on *Primitive* menu option produces a pop up window that shows all the primitives presently in the primitives library. Figure 38 shows the primitive pop up window. Figure 39 is the complete unit. The VHDL code for this unit can be generated and sent to a disk file as shown in figure 40. The VHDL code can also be viewed on the screen by picking the *Show VHDL* menu option. Figure 41 shows the VHDL code being displayed. If the analyzer system is setup, the VHDL code produced can also be analyzed. Figure 42 shows the "OSCILLATOR" VHDL program being analyzed inside the Modeler's Assistant.

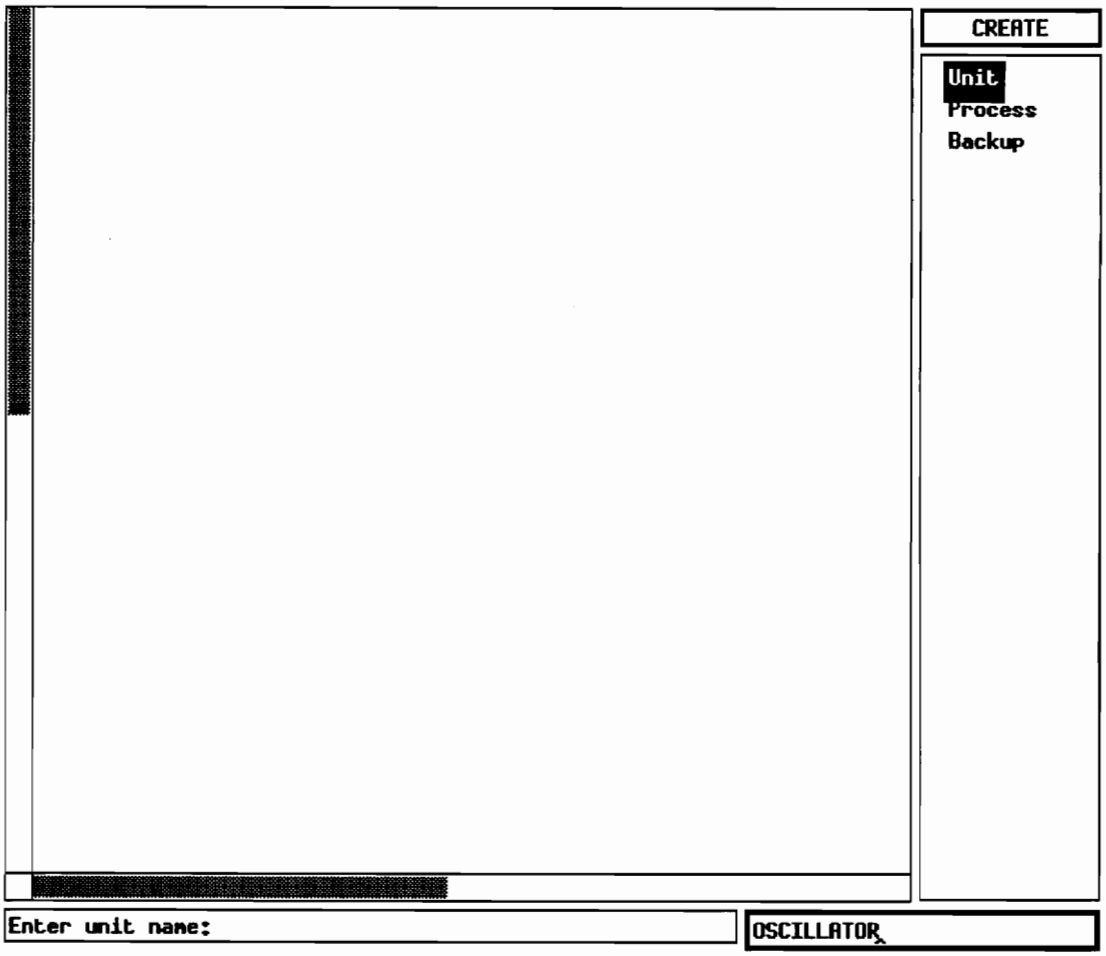


Figure 28. Creating unit "OSCILLATOR"

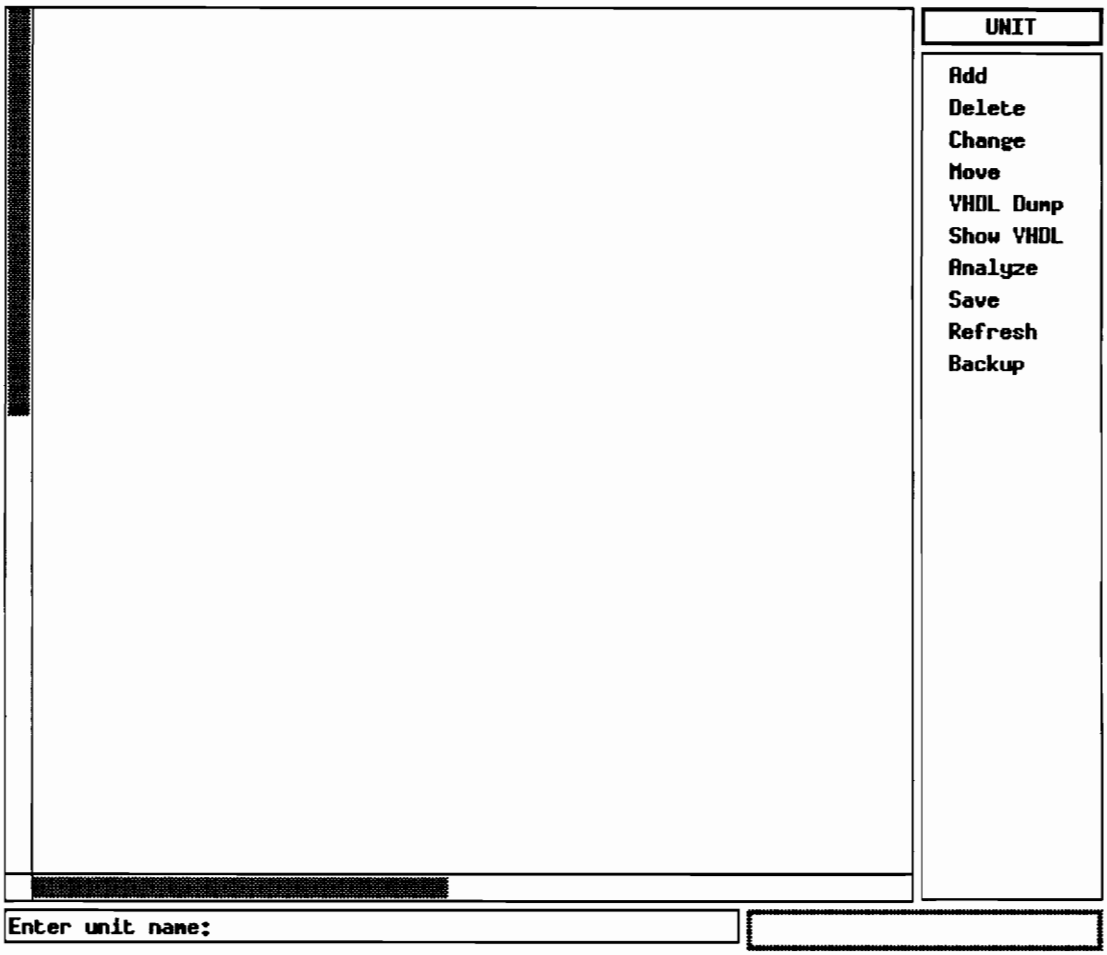


Figure 29. Window with *UNIT* menu

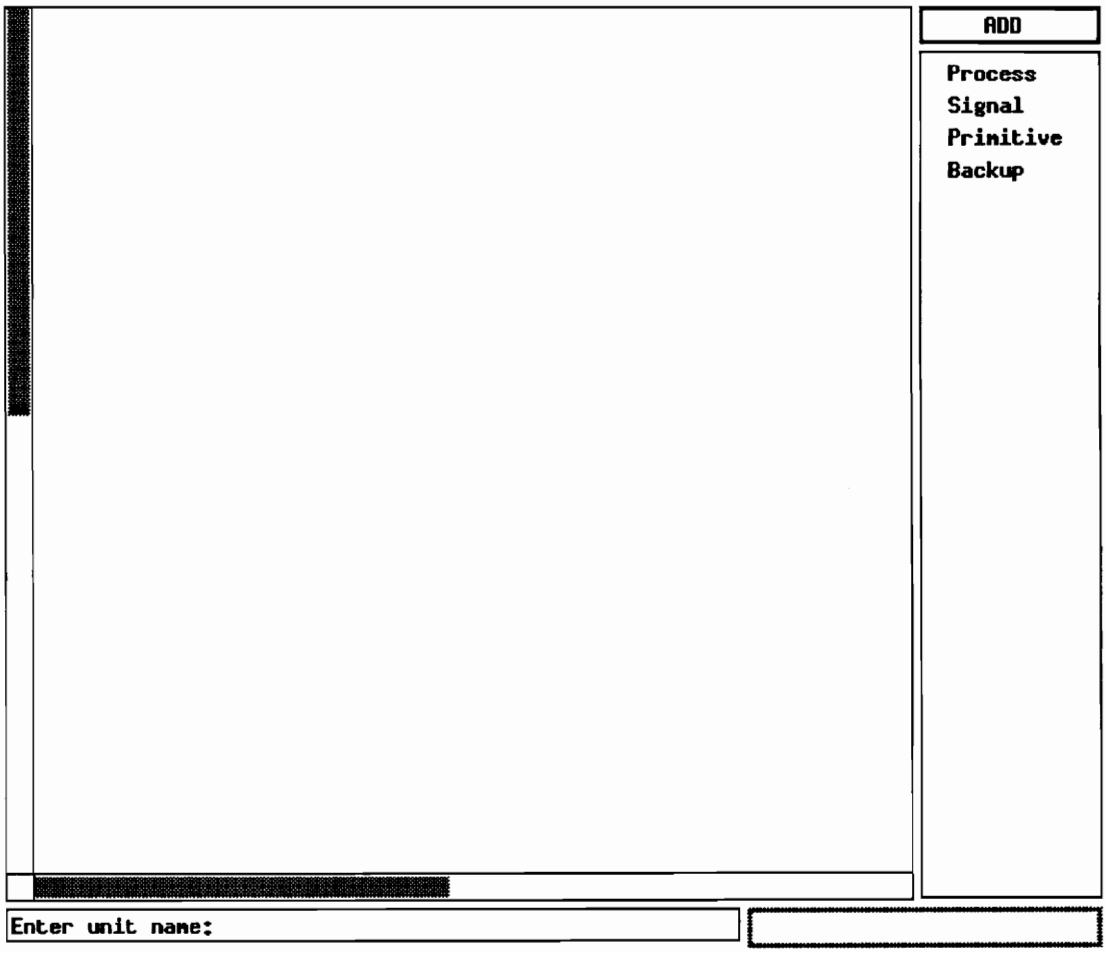


Figure 30. Window with *UNIT - ADD* menu

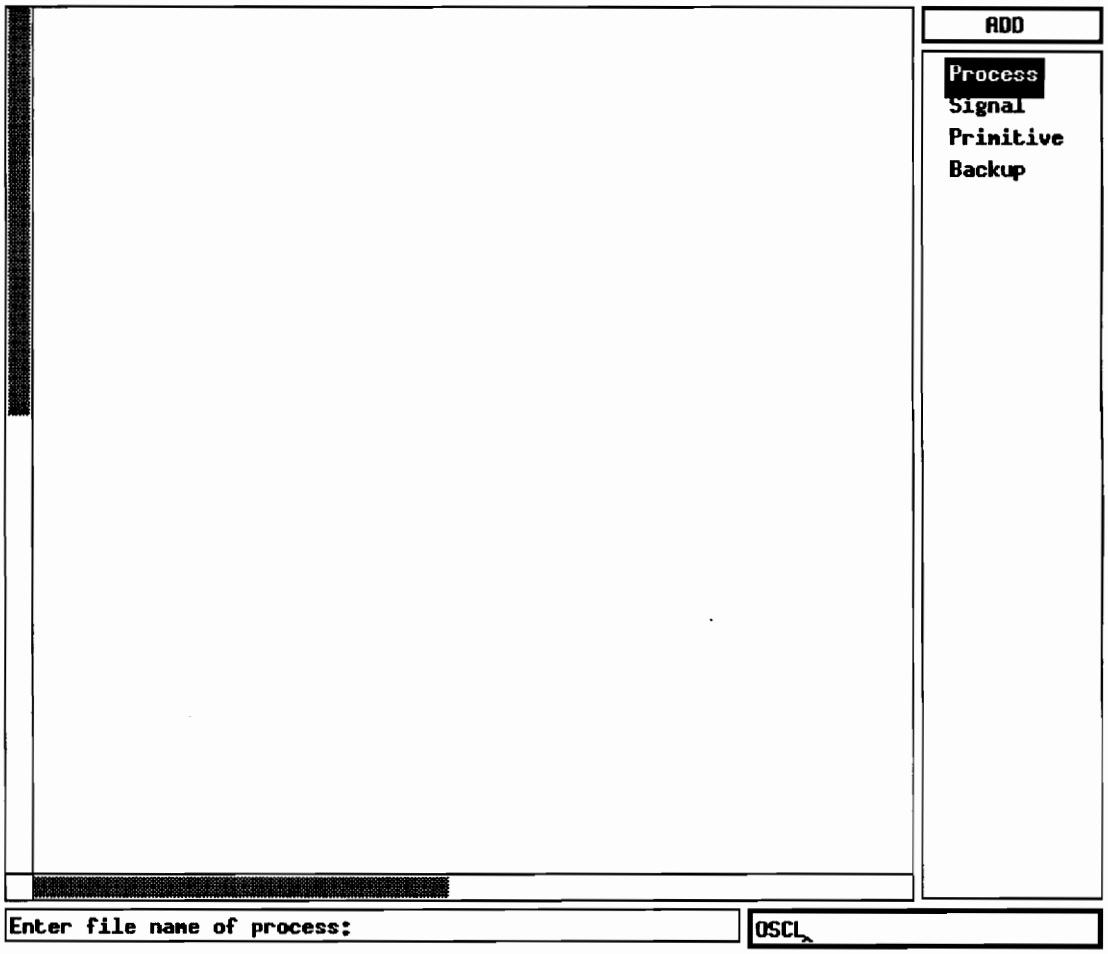


Figure 31. Adding process "OSCL"

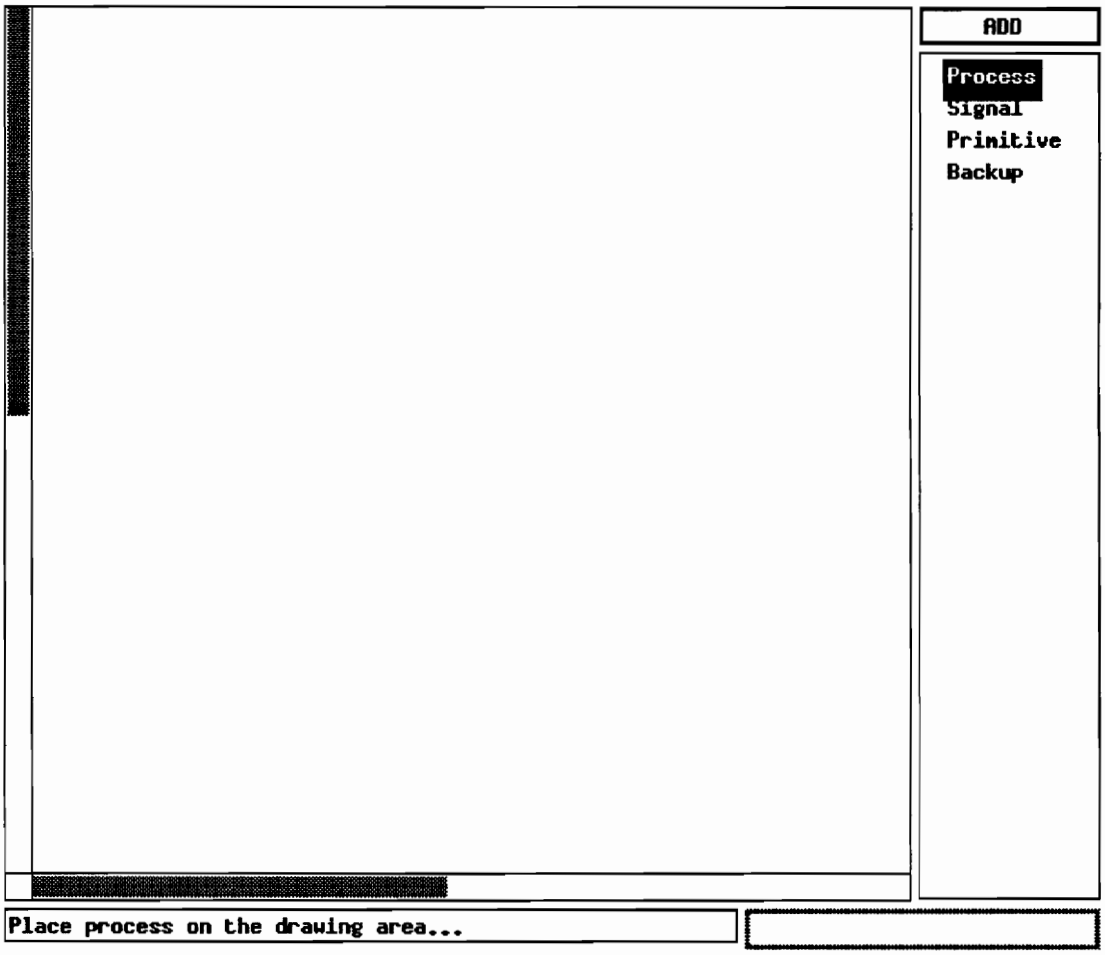


Figure 32. Prompt for location of process on screen

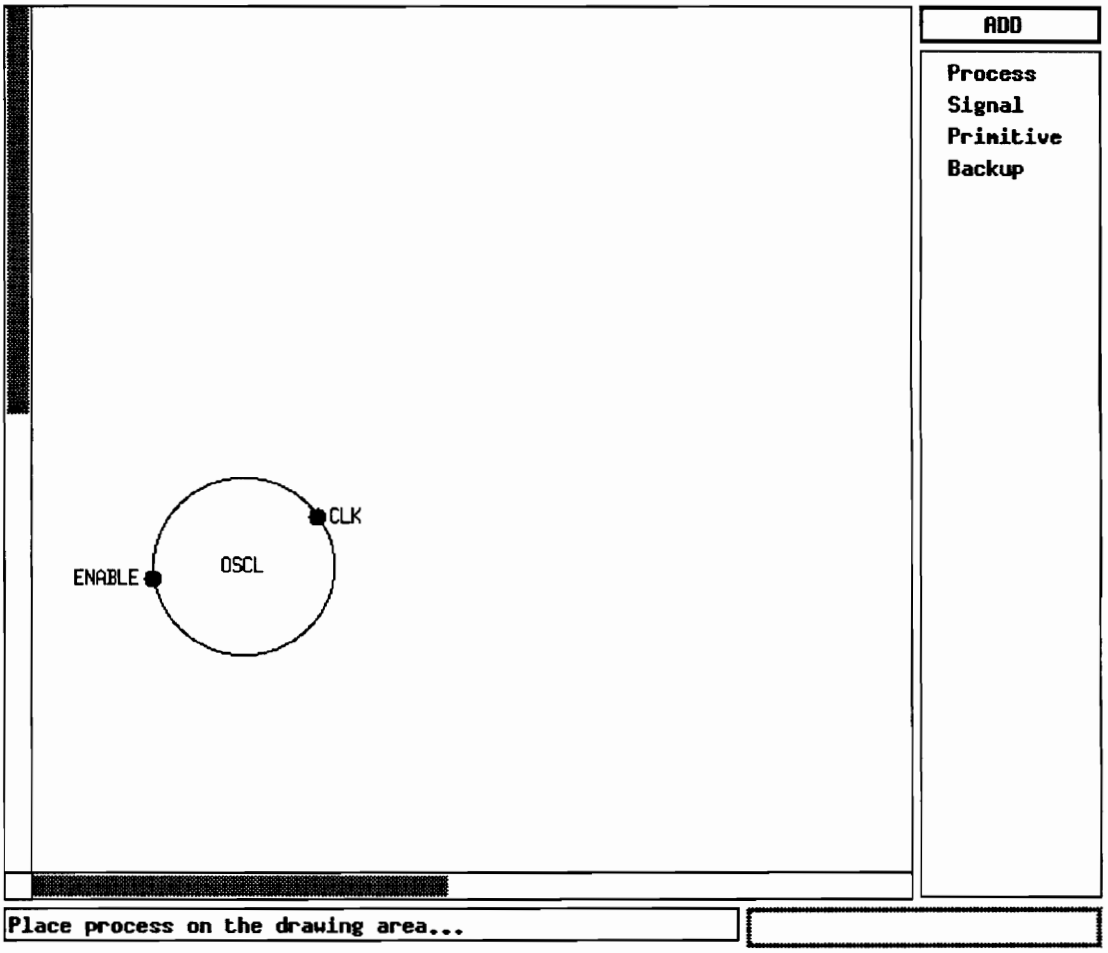


Figure 33. Unit with process "OSCL"

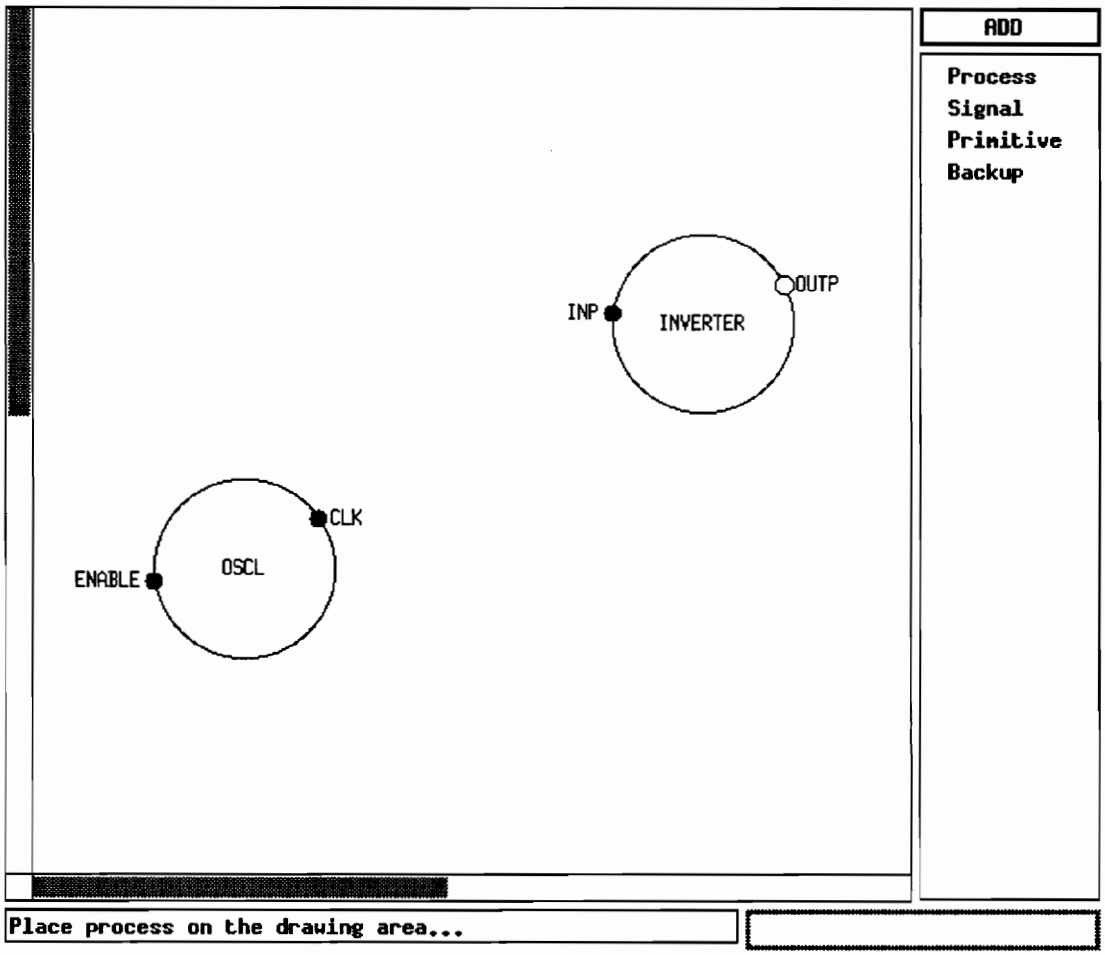


Figure 34. Unit with processes "OSCL" and "INVERTER"

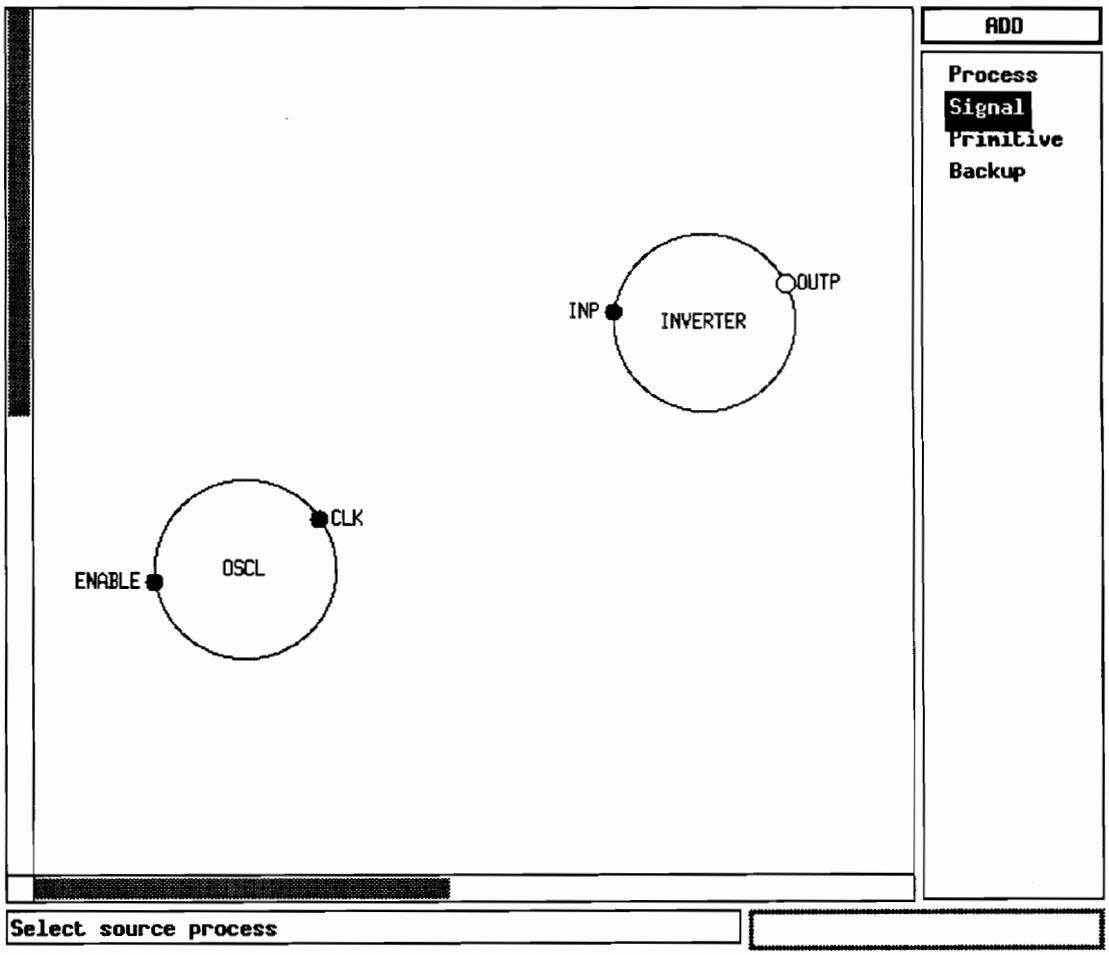


Figure 35. Adding signal - Selecting source proc

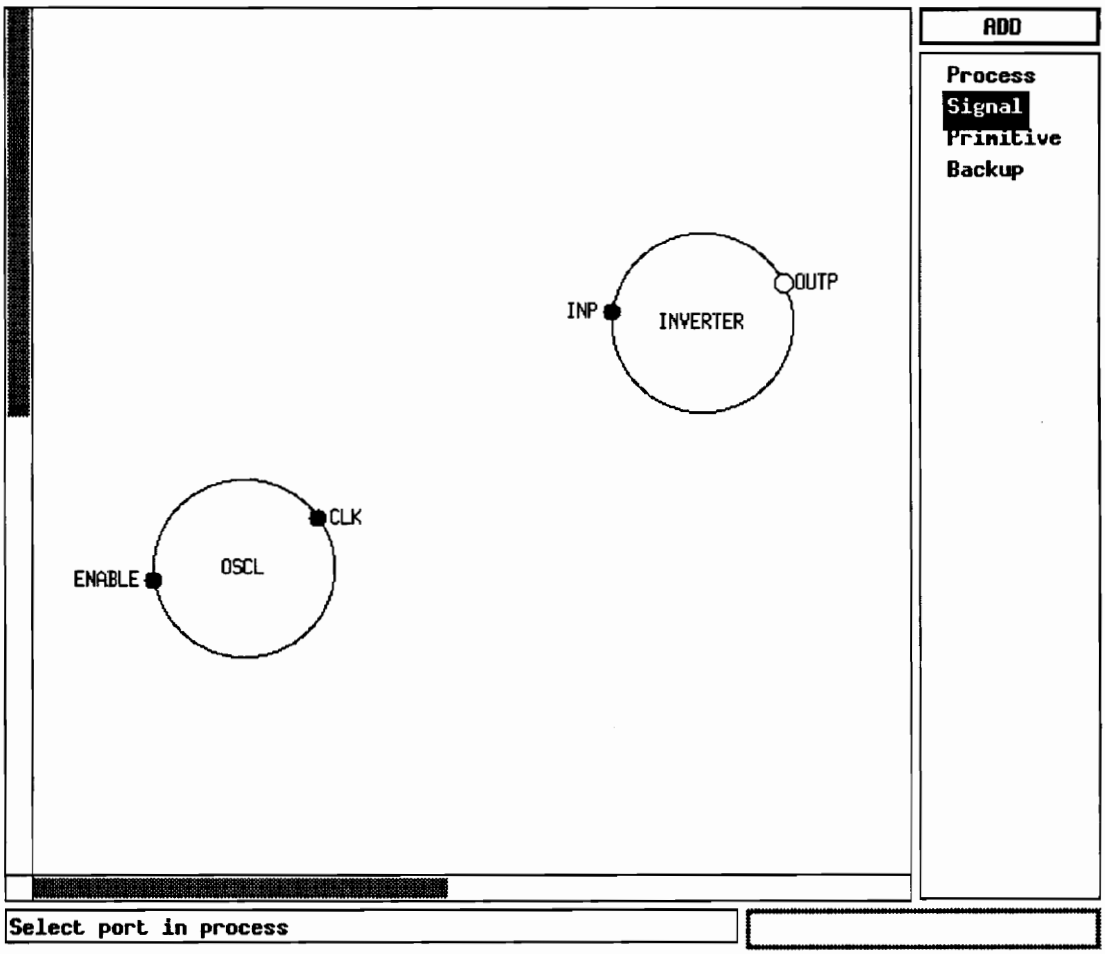


Figure 36. Adding signal - Specifying process port in process

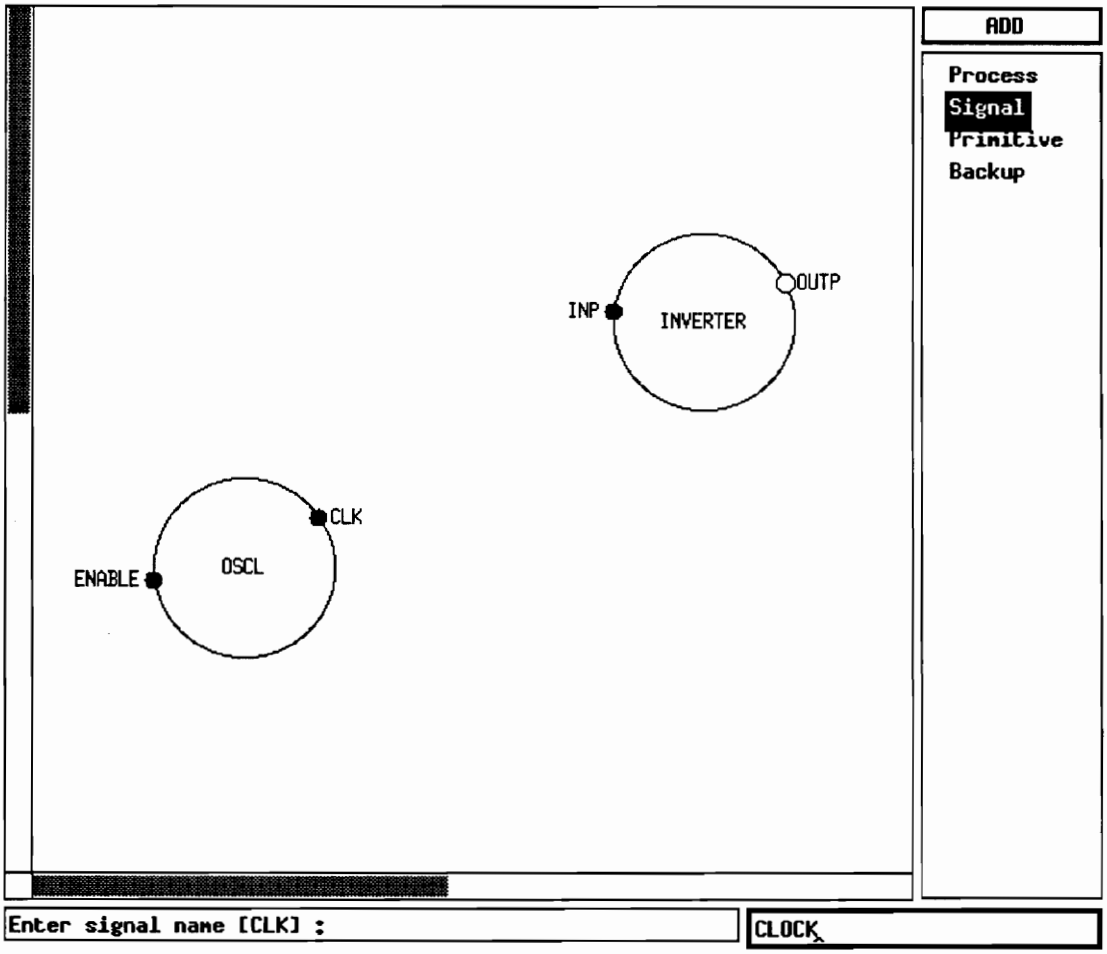


Figure 37. Specification of signal name

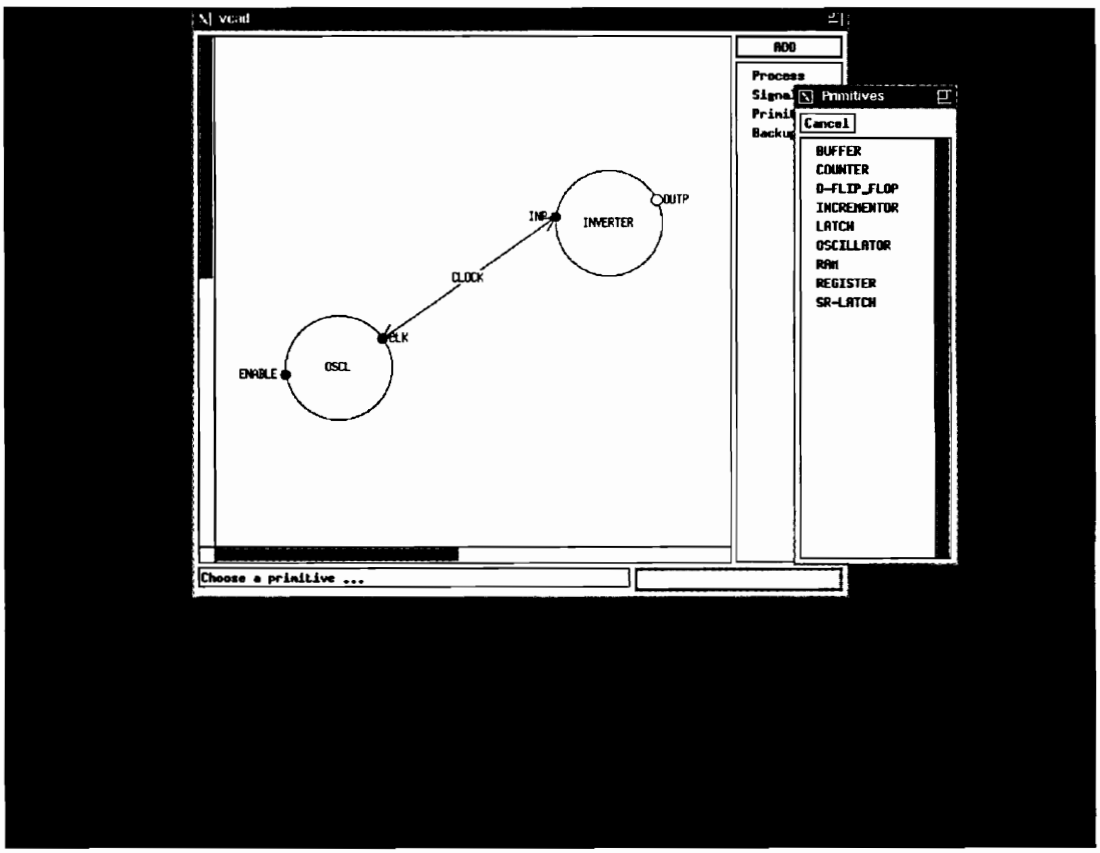


Figure 38. Adding a process from the primitives library

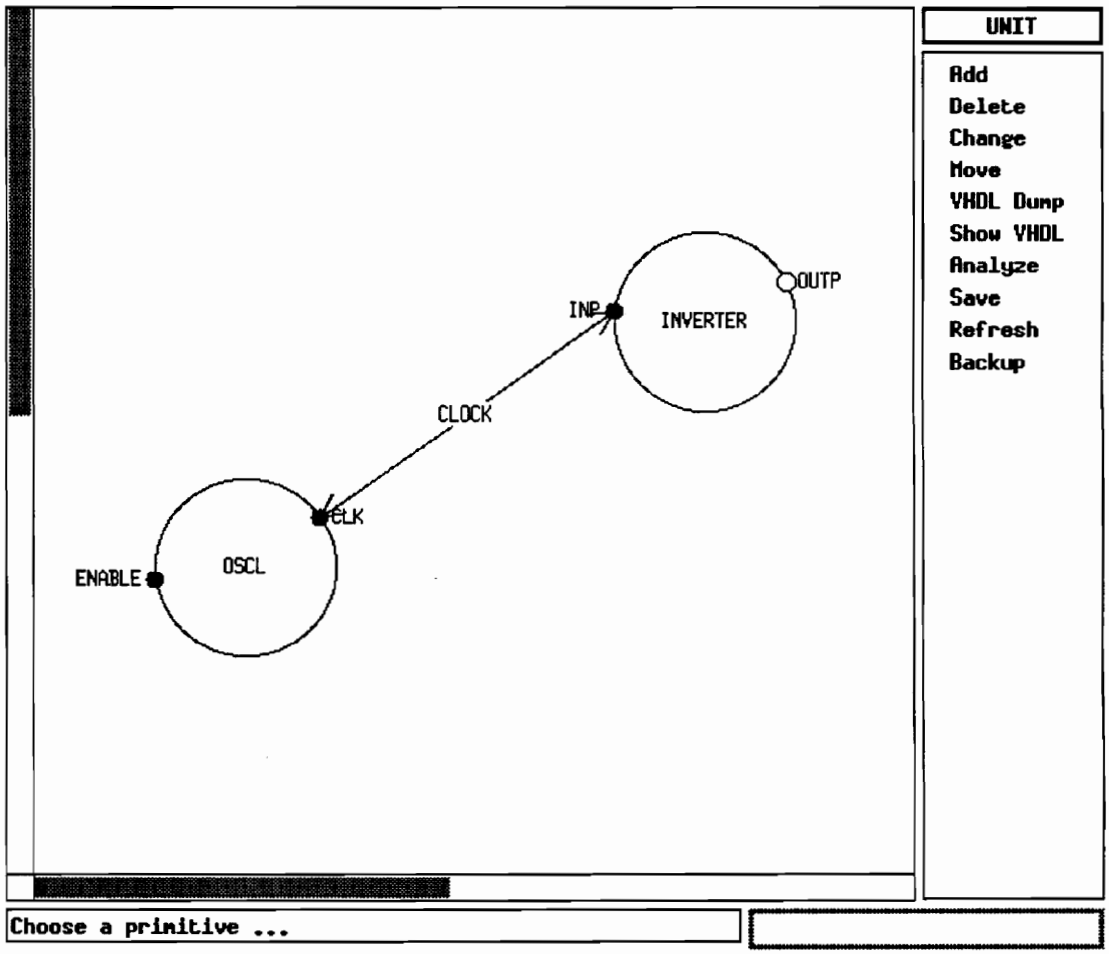


Figure 39. Final process model graph

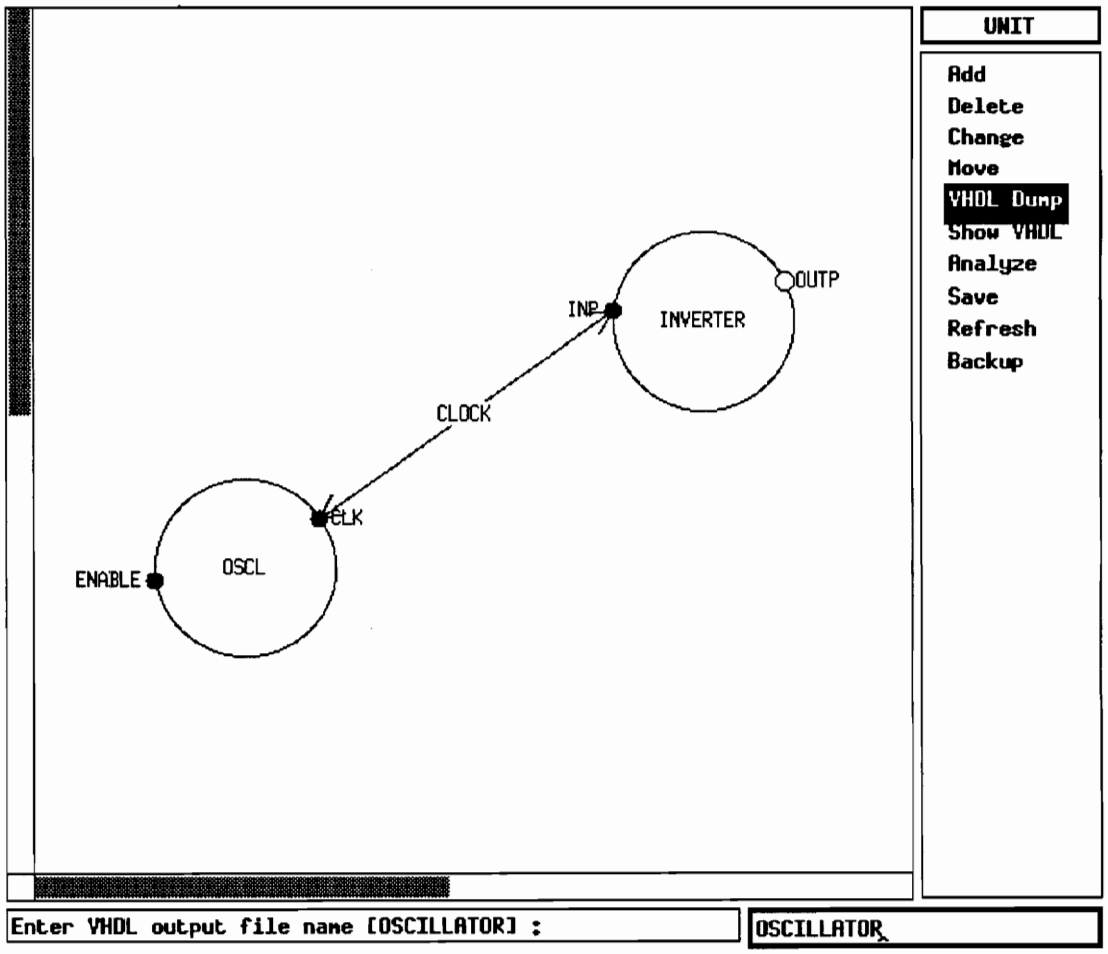


Figure 40. Dumping VHDL code to disk file

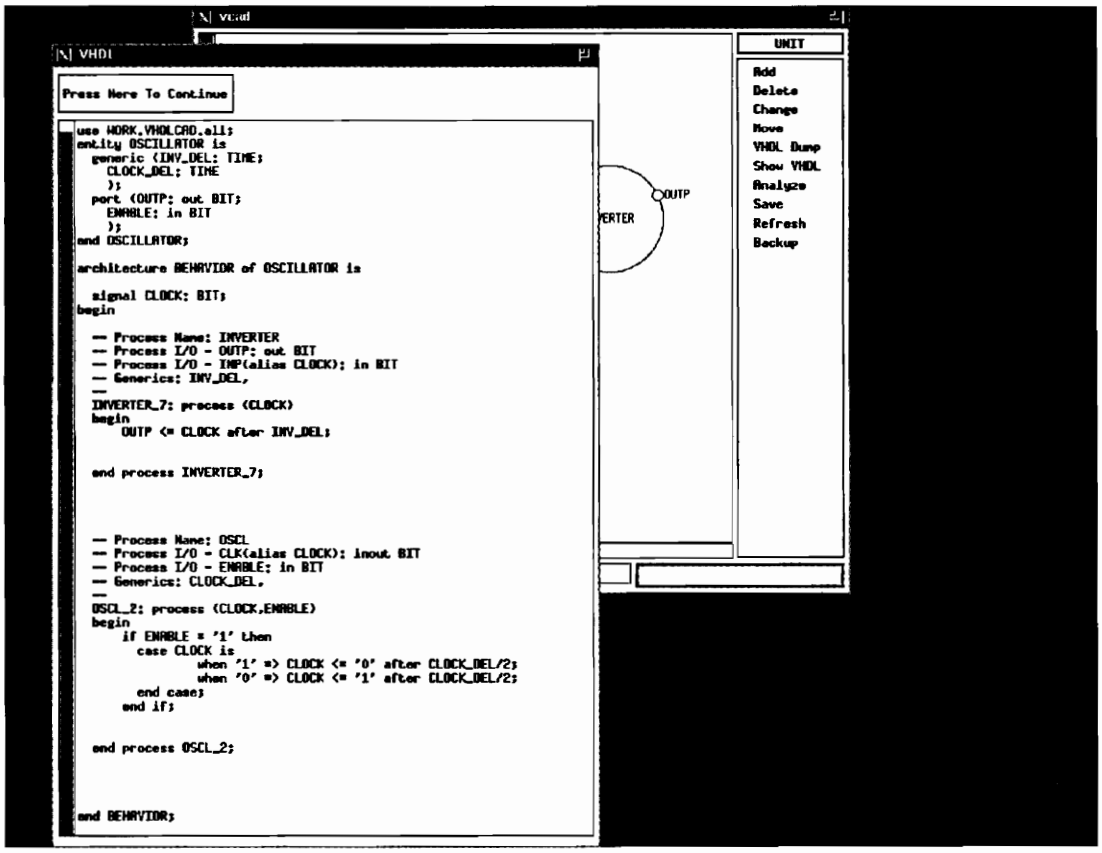


Figure 41. Displaying VHDL code in a window

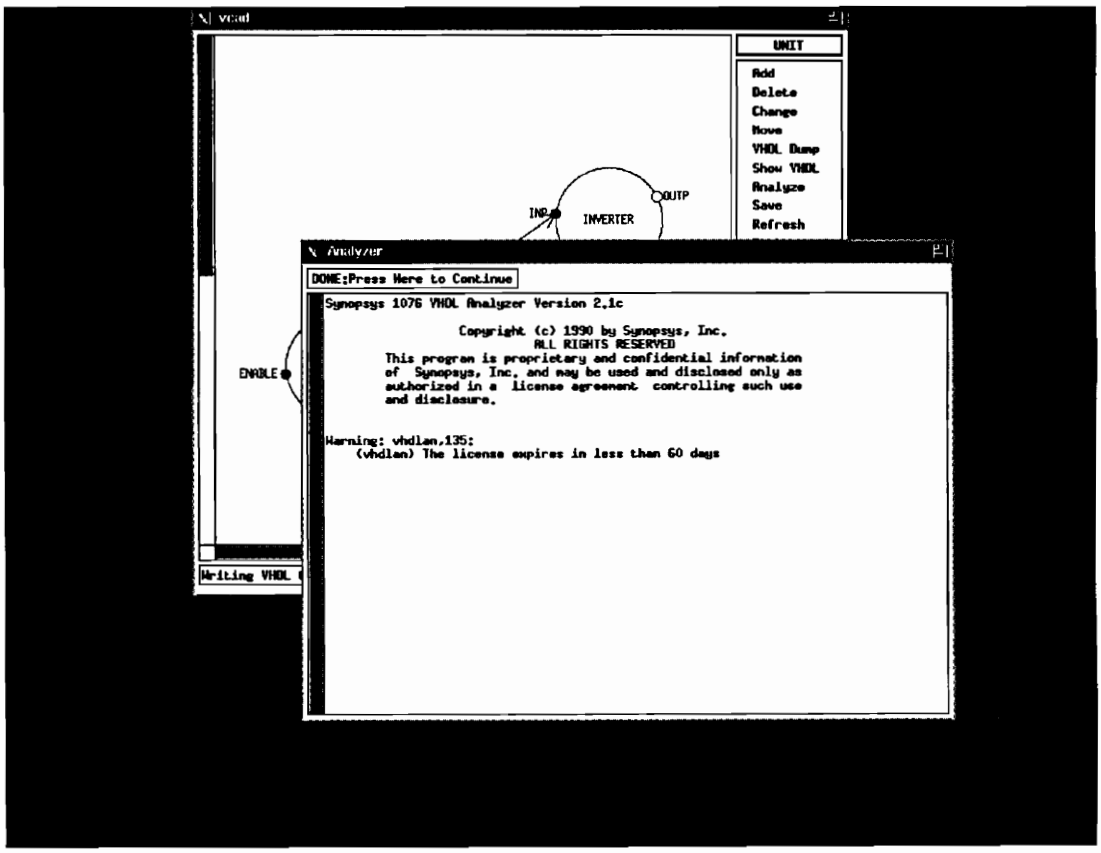


Figure 42. Analyzing generated VHDL code

VITA

Balraj Singh was born in Calcutta, India, on the 26th of February, 1968. He received a Bachelor of Technology degree in Electrical Engineering from the Indian Institute of Technology at Bombay, India, in 1989. He is currently working towards the completion of requirements for the Master of Science degree in Electrical Engineering at the Virginia Polytechnic Institute and State University, Blacksburg, Virginia.

7/17/91