

Natural Language Interface To a VHDL Modeling Tool

by

Meenakshi Manek

Thesis Submitted to the Faculty of Virginia Tech in partial fulfillment of the requirements

for the degree of

Master of Science

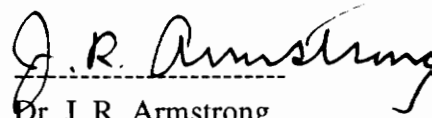
in

Electrical Engineering

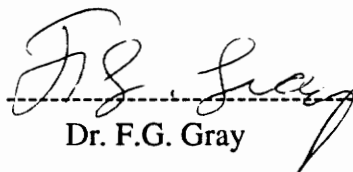
APPROVED:



Dr. W.R. Cyre, Chairman



Dr. J. R. Armstrong



Dr. F.G. Gray

C.2

LD
5655
V855
1993
m263
C.2

A Natural Language Interface To a VHDL Modeling Tool

by

Meenakshi Manek

Dr. Walling R. Cyre, Chairman

Electrical Engineering

(ABSTRACT)

This thesis describes a Natural Language (NL) interface to a VHDL modeling tool called the Modeler's Assistant. The primary motivation for the interface developed in this research work is to permit VLSI modelers who are not proficient in VHDL to rapidly produce correct VHDL models from manufacturer's descriptions. This tool should also be useful in teaching the VHDL language. The Modeler's Assistant has supported graphical capture of behavioral models in the form of Process Model Graphs consisting of processes (nodes) interconnected by signals (arcs). The NL interface that has been constructed allows modelers to specify the behavior for the process nodes using a restricted form of English called ModelSpeak. A Spell-checking routine (of the UNIX operating system) is invoked to reduce input errors. Also, the grammar employed, accepts multi-sentence descriptions rather than just a single sentence. Correct VHDL for each process is synthesized automatically, but user interaction is solicited where needed to resolve ambiguities such as the scope of loops and the type of signals and variables. The Modeler's Assistant can then assemble the VHDL code for these processes, along with the information about the interface description from the PMG, into a complete entity model.

Acknowledgments

I am indebted to my mother for supporting me and encouraging me through all the years of my education, without whom I could not have reached up to this academic level. I take this opportunity to thank Dr. Cyre whose valuable advice and guidance helped me complete this project as anticipated. I had a very good rapport with him which I think is very important when involved in a research project with a faculty member. I also like to thank Dr. Armstrong for giving some good suggestions to implement the project and Dr. Gray for consenting to serve on my thesis committee. It has been a good experience working with all of them. I would finally like to thank God for bringing into my life a special person, Alexander Honcharik. He is my would-be husband and also my research colleague. His warm love and care make all things so wonderful and possible.

Table of Contents

Chapter 1. Introduction.	1
1.1 Motivation and Goal	1
1.2 Approach	3
 Chapter 2. Related Research	 4
2.1 The Phran-Span system	4
2.2 The ASPIN Parser and Semantic Analyzer	6
2.3 The User Interface using BIF for VHDL Behavioral Modeling	8
2.4 The SOPHIE system	9
2.5 The INLAND system built using the LIFER framswork	11
 Chapter 3. Overview of the Modeling Tool	 14
3.1 Process Model Graphs	14
3.2 PMG construction and Model Generation using Modeler's Assistant	16

Chapter 4. The NL Interface	19
4.1 ModelSpeak: The Input language	19
4.2 Parser	24
4.2.1 Language Theory	24
4.2.2 Traditional syntactic Parser	26
4.2.3 VHDL Semantics	28
4.2.4 Parsing and Generation with Semantic Parser	33
(a) Semantic rules for isolated sentences	33
(b) Semantic rules for multi-sentence descriptions	38
4.3 User Interaction	41
 Chapter 5. Program Implementation	 46
5.1 Programming technique and other added features	46
5.2 Interface to the Modeler's Assistant	48
 Chapter 6. Results	 50
6.1 The DMA Interface Model	50
6.6 The Three module system	57
6.5 The MUX primitive	63
6.2 The Parallel I/O port model	65
6.4 The UART model	68
6.3 The Traffic Controller Example	73
 Chapter 7. Limitations and Future Work	 77

References 79

Appendix A. Users guide for the Semantic Parser 81

Appendix B. Semantic categories, Key words and symbols ModelSpeak Language 84

Appendix C. Examples of ModelSpeak sentences 86

Appendix D. VHDL Models of the DMA Interface module, Three Module System, the
Parallel I/O ports, the UART model and the State Machine of a Traffic
light Controller 88

Appendix E. Prolog program implementing the semantic parser 100

Vita 114

List of Illustrations

Figure 1: A Process Symbol in a PMG	15
Figure 2: PMG of a Counter Circuit	15
Figure 3: Overview of the Modeler's Assistant with the NL Interface	16
Figure 4: VHDL Behavioral model of the Counter circuit generated by the Modeler's Assistant	18
Figure 5a: Grammar rules in BNF notation to parse a simple sentence	27
Figure 5b: A Grammatical structure tree for a simple English sentence	27
Figure 6a: Analysis and Generation rules for a signal_assignment_statement of type (1)	33
Figure 6b: A semantic structure tree for a signal_assignment_statement of type(1)	34
Figure 7 : Analysis and Generation rules for a signal_assignment_statement of type (2)	36
Figure 8a: Top-level Analysis and Generation rules for an if_statement	37
Figure 8b: Lower-level Analysis and Generation rule for the if_statement	38
Figure 9: Semantic Rules for Multi-sentence description	40
Figure10: Process symbols representing the partitions of the DMA Interface Model	51

Figure11a: The Three Module System	57
Figure11b: Functional Partitions of the Add & Store Unit	57
Figure11c: PMG for the Three Module System	59
Figure 12: Process Symbol of a MUX	64
Figure 13a: Block Diagram of Intel 8212 Chip	65
Figure 13b: PMG of Parallel I/O Port Model	66
Figure 14 : PMG of the UART Model	67
Figure 15a: State Machine of a Traffic Light Controller	71
Figure 15b: PMG representing the State Machine of the Traffic Light Controller ..	74

Chapter 1

Introduction

1.1 Motivation and Goal

The VHSIC Hardware Description Language (VHDL) [1] is rapidly becoming a industry standard for simulation and design verification of Hardware systems. Also, VHDL is a very useful form of design documentation of digital systems. The versatility of the language supports accurate modeling of the behavior of concurrent hardware. At the same time, learning a formal language like VHDL poses a challenge to the inexperienced modelers. While developing behavioral models, apart from focusing on the accurate algorithmic description of the system to be modeled, the modeler has to adhere to the VHDL syntax and type matching, and also include the necessary declarations. If any of these are violated, the model has to be corrected until it is totally error free and ready for simulation. As it is seen, the task of developing models is rather laborious and time consuming. A modeler who starts with the behavioral specification of a system to be designed, gets entangled with the formatting details.

To overcome these problems a number of CAD tools are available that aid in VHDL modeling. The Modeler's Assistant [2,3] is one such CAD tool that captures a graphical description of a behavioral model, the *Process Model Graph* (PMG). The modeler

constructs a PMG by identifying processes and signals as well as naming and typing these elements. The processes are either created by the modeler or are selected from a process primitive library within the Modeler's Assistant. The PMG provides information on the interface description of the processes and the signals between them. However, the PMG itself does not specify the behavior of the system to be modeled. Instead, each process is associated with a VHDL code describing the behavior of that process. The Modeler's Assistant then assembles the information from the PMG and the individual VHDL process codes to generate the VHDL behavioral model of the system. Thus, the Modeler's Assistant aids in rapid development of behavioral models by allowing a partial graphical entry of the models in the form of a PMG. However, when modelers require the creation of the processes not included in the process primitive library, it is also required that they associate the process behavior in VHDL. In this case, the modelers are required to have a good working knowledge of VHDL. For modelers having almost no experience in writing VHDL code, the process of creating models will be tedious and time consuming, particularly when the models happen to be large and complicated. Furthermore, the modelers are not free from committing syntactic errors, delaying the design process. Instead, if the modeler is permitted to use a restricted natural language, less formal than VHDL, to develop behavioral models, a lot of effort and time can be saved. Thus, a Natural Language Interface has been developed that automatically synthesizes correct VHDL code for the processes from natural language descriptions.

1.2 Approach

In this research work, the NL interface is developed for a behavioral domain. Natural language descriptions in this domain can be expressed using a limited vocabulary and grammatical constructs [4]. Thus the interface is built to accept descriptions in a restricted form of English called the ModelSpeak. Either a single sentence or a paragraph of sentences can form the description of a process. In the case of a multi-sentence description, the corresponding sequence of VHDL code is generated. As a preprocessing step, the description is spell-checked before it is parsed. This process displays a list of misspelled and unknown words thus informing the user to correct the spelling errors and avoid further failure of parses.

The interface structure consists of a single module called the semantic parser [5]. The grammar employed by the semantic parser uses non-terminal symbols corresponding to VHDL concepts rather than the conventional grammatical categories. Then, the concepts found in parsing are concurrently mapped onto VHDL code by the same module.

When constructs implying for and while_loops are encountered, an interaction is initiated with the user to resolve the ambiguity of the scope of the loop. Based on the feedback from the user, a loop construct, encompassing the other desired VHDL statements, is generated.

The semantic parser consists of grammar rules written in Quintus Prolog and currently runs on a SUN SPARC2 workstation.

Chapter 2

Related Research

In this chapter, other researches works that are relevant to this efforts are reviewed. The PHRAN-SPAN system initiated by Granacki at USC and the ASPIN system developed at Virginia Tech accept digital system specifications expressed in natural language. Burton's work with the SOPHIE natural language processor was built to accept student queries and statements and was based on semantic grammar. Of the other two works reviewed, the user interface using BIF is an example of an interface for VHDL behavioral modeling and the INLAND system is built using a natural language framework for a restricted domain called LIFER. The INLAND system is developed to access the navy data.

2.1 The PHRAN-SPAN system

The PHRAN-SPAN [8] system is a natural language interface to a CAD system called ADAM, the USC Advanced Design Automation System. It allows users to enter digital system specifications in restricted English. The behavior of digital systems, so entered, is translated into an internal representation called the Design Data Structure (DDS) [8], a

formal representation that further helps in the synthesis of the hardware systems described. The interface structure includes two major modules the PHRAN (PHrasal ANalyzer) and the SPAN (SPecification ANalyzer). PHRAN , while parsing through a sentence, detects the sentence pattern by referring to a database of pattern-concept pairs. When a match for the pattern is found the corresponding concept is extracted and the variables in the concept are instantiated. The concept holds the meaning of the pattern associated to it and forms an input to SPAN. SPAN does an analysis of the concept and then generates a representation of the design data, corresponding to the input sentence. Also, for user verification SPAN generates the output which is an interpretation of the input sentence. The DDS is the desired target output from PHRAN-SPAN which is used by the ADAM system for synthesis purposes.

For example, consider the sentence

The CPU transfers the block of data bytes from the disk to the control store. (1)

This sentence is matched with the internally stored pattern:

[(or (a_component) (df_opn)) (root%transfer) (df_val)]

[from (or(a_component) (df_opn))]

[to (or(a_component) (df_opn))]

Here, a_component(device), df_opn(data flow operation), df_val(data flow value) are the semantic categories

The concept associated to the pattern is extracted by PHRAN and is a frame containing variable names that are instantiated using data from the input sentence.

The natural language interface developed for the research work described in the present thesis also allows users to enter behavioral descriptions of digital systems in restricted English. Both the systems accept constructs with limited vocabulary and grammar, pertaining only to behavioral modeling and hence employ a semantic-based parser that searches for semantic categories. But it differs from Granacki's work described above, in that the interface structure consists of only a single module namely the semantic parser and is paragraph based. The target output of VHDL statements is directly constructed while interpreting the sentences. There is no intermediate representation to hold the meaning of the input sentence. This limits the parser to be application specific. Whereas, the PHRAN parser can still be used in other applications with a modified database. Also, the PHRAN parser is only sentence based and constructs fragments of the DDS from each sentence. The PHRAN-SPAN has heuristics incorporated so that the ambiguous sentences are interpreted correctly, while in this system the problem of ambiguity is solved by user interaction. In the case when the construct introduces ambiguity in the way in which other related sentences are interpreted an interaction with the user is initiated to ascertain the nature of the output construction desired.

2.2 The ASPIN Parser and Semantic Analyzer

The Automated SPecifications INterpreter (ASPIN) [6] is a tool being developed to take as input digital system specifications written in natural language and maps them onto conceptual graph knowledge representation. From this intermediate representation the target VHDL models are generated. The specifications input to this system can be both

behavioral as well as structural descriptions of digital hardware which by nature encompass a large domain. Because the domain is broader and a richer variety of expression is allowed, the natural language analysis is more complex. The English sentences are run through a syntactic parser that can parse these general type of descriptions.

The example sentences are:

Execution of a starcnt instruction connects the t1 input pin to the counter input and enables the counter. (2)

The program counter is a 16-bit register that contains the address of the next instruction to be executed. (3)

The parser applies grammar rules to the sentence and generates one or more parse trees that give the grammatical categories of the sentence constituents. This is fed to the **semantic analyzer** that generates a conceptual graph [13] which holds the meaning of the sentence. The concepts in an input sentence are replaced with the canonical conceptual graphs from the database and these are attached together based on the valid linkages of concepts, as specified in the library of canonical conceptual graphs. This is then input to the program called **linker** to generate the corresponding VHDL model. Therefore, the capability to handle general descriptions makes the process of generating models lengthy and complex.

In contrast to the ASPIN parser, the research work described in this thesis allows only for behavioral descriptions. Only a limited variety of constructs that map onto VHDL statements are allowed. Hence, the target VHDL model can be generated in a single step.

Also, the parsing technique of the semantic parser employed requires more modest searches for its limited constructs than the analysis process involved in the ASPIN system. The general type of sentences accepted by ASPIN tend to be ambiguous. But, since the semantic grammar used in this thesis is based on semantic concepts and their attachment, it restricts the way in which a sentence is interpreted thus avoiding most of the ambiguity.

2.3 The User interface using BIF for VHDL Behavioral Modeling

BIF (Behavioral Intermediate Format) [11] was mainly developed to be a design environment that provides users with a natural means of specifying designs for synthesis purposes. The motivation for this work had been the fact that although VHDL is a good language for modeling and simulation of digital designs, it does not support synthesis tasks very well. Not all VHDL constructs correspond to realizable hardware, hence the user is restricted to only certain particular modeling styles. BIF eliminates this difficulty by providing state tables that take textual inputs and these are mainly meant for expressing complex designs such as behavioral hierarchy and hierarchical time-outs and events. These BIF descriptions can directly interact with the synthesis tools to carry out the required synthesis steps. In order to enable simulation of BIF descriptions a translator has been included that automatically generates VHDL from the BIF descriptions.

Thus, for expressing models involving hierarchy, concurrency and timing relationships the BIF user interface can be used instead of having to develop models using the tedious process with VHDL. The translator makes use of special templates and auxiliary signals

that allow features in BIF to be translated to those in VHDL. The algorithm first converts the hierarchical BIF description to a data structure called a Table Tree, then, templates are applied to this hierarchical structure to generate the corresponding VHDL. The VHDL thus generated has a typical structure in which the architectural body of the code consists of behavioral blocks which correspond to the states in the BIF description. Each block has a guard expression that checks for the state signal and the event that triggered that state.

Thus, the BIF user interface is used to capture hierarchical system behavior in a natural fashion and the VHDL for such a design is automatically generated by the translator. But, the natural language interface for this thesis is developed for a CAD tool that allows the user to model the behavior of digital design by partitioning the system into processes and interconnecting them with appropriate signals. This is entered graphically, using a representation called the Process Model Graph (PMG). Now, the natural language interface allows modelers to use English descriptions to express behavior of each of the processes. Thus this thesis describes the research work focused on developing behavioral models in VHDL, by using simple natural language descriptions .

2.4 The SOPHIE system

The SOPHIE interface [5] system was built so that students could interact with the machine used as an educational kit in a science laboratory. The SOPHIE natural language

processor allowed students to query and get feedback, aiding them in solving problems easily. The example sentences handled by this system are:

What is the voltage across C3. (4)

What is the current through R9. (5)

Is the positive terminal of R9 shorted to the anode of D6. (6)

Since this level of interaction pertained only to a limited domain, the parsing scheme adopted was based on semantic grammar. Here, semantic concepts like voltage, current and measurement formed the non-terminals of the grammar rules and could in turn consist of other constituent concepts. Other characteristics that were incorporated in the processor were the ability to recognize deletions and ellipsis. Examples for this type of sentences are (8) and (9) shown below.

What is the current through R1. (7)

What is it through R2. (8)

Through R3. (9)

As the sentence is parsed the semantic form or the "meaning" is constructed. The meaning of a phrase that refers to a primary concept is the data object. In the case of a query, the meaning would be a call to the procedure that answers the query. For a command, the meaning calls the procedural specialist that performs the action. Pronouns and deletions are allowed in the statements. The context mechanism used for pronouns and ellipsis searches through the record of previously generated "meanings" and a suitable referent that matches the instance is found.

In the interface described in this thesis, the parsing technique is based on a semantic grammar similar to the that used in the SOPHIE system. Here, the semantic concepts include if_statements, signal_assignments, actor, condition, action and so on. There is no

intermediate form to hold the meaning of the input. This is because the interface does not depend on any other component such as the procedural specialist, "DOFAULT", in the SOPHIE system to generate the target output. Here, the output is neither an answer to a query or an action to a command but a translation from an informal language to a formal language.

Also, the interface is built to accept multi-sentence descriptions that correspond to sequences of VHDL statements. When the description is not a single sentence but paragraph-based, the order of sequence of the generated code depends on the sentence connectivity. This is checked for, while parsing, and the desired sequence of VHDL code is generated. No pronouns or deletions are allowed because no previous histories of parses are retained.

2.5 The INLAND system built using the LIFER framework

LIFER (Language Interface Facility with Ellipsis and Recursion) [12] is a framework that helps develop application oriented languages that accept a subset of the natural language, typically used in the specific domain that it is developed for. The system consists of input functions that aid in specifying the language and a parser that takes as input these specifications and produces the required output. The language specification refers to the syntax and the semantics of the language. The LIFER has built-in mechanisms for correcting spellings, recognizing ellipsis and extending the language by definitions. Thus,

the development of new interfaces for different application domains requires only the new language specification.

INLAND (Informal Natural Language Access to Navy Data) is an interface that was developed using the LIFER framework. The INLAND interface is also based on semantic grammar, similar to the one initiated by Burton for the SOPHIE system. Here, the semantic categories include <ATTRIBUTE>, <SHIP_NAME> and so on. Although, traditionally the words *length*, *displacement*, *Nautilus* and *Kennedy* are considered to belong to the <NOUN> category, here, *length*, *displacement* belong to the category <ATTRIBUTE> and *Nautilus* and *Kennedy* are of type <SHIP_NAME>.

Sentences such as

What is the length of Nautilus? (10)

The pattern matches with the grammar rule shown below.

**<LTG> => <PRESENT> THE <ATTRIBUTE> OF <SHIP_NAME> |
(IDA(APPEND <SHIP_NAME> <ATTRIBUTE>))**

Here, *what is* matches <PRESENT>, *length* matches <ATTRIBUTE> and *Nautilus* matches <SHIP_NAME>. The latter portion of the rule is the part that forms the formal query to the database system. <SHIP_NAME> and <ATTRIBUTE> are replaced by their semantic meanings computed with the information from the input sentence. Thus, the query formed for the given sentence is

(IDA (APPEND '(NAM EQ JOHN. #F. KENNEDY)) '(?LENGTH)))
=> (IDA '(NAM EQ JOHN. #F. KENNEDY)) '(?LENGTH)))

The result of this query would be (LENGTH 1072 feet)

The interface described in this thesis is based on a similar principle of the semantic grammar. But, the application is to generate VHDL code rather than database queries. Also, it accepts a multi-sentence along with the user interaction feature to clarify ambiguities. These features are not present in the INLAND system. But, since it accepts simple queries, features such as recognizing ellipsis and language extendibility are incorporated as a part of the interface using the LIFER framework.

Chapter 3

Overview of the Modeling Tool

3.1 Process Model Graphs

The Modeler's Assistant was designed for developing VHDL behavioral models. The VHDL behavioral model development of a hardware system involves partitioning of the system into modules corresponding to processes and identifying signals between them. This information is then input graphically to the Modeler's Assistant in the form of a Process Model Graph (PMG). The PMG represents a concurrent system in that it is a directed graph with the nodes of the graph representing the concurrent processes in the VHDL model and the edges of the graph representing the signals between them.

In a PMG, a process is represented by a single circle, also called the process symbol. The smaller circles on its boundary represents the process ports. A process port is a signal that the process either writes to or reads from. A port that is filled indicates that the port is in the sensitivity list of the process and if a port is empty it is not. Also, a constant is represented by a filled small square within the process and a variable is represented by an empty small square within the process. Figure 2 shows an example process symbols in a

PMG. In this figure, the process named Proc has three ports I1, I2, I3. Port I1 is in the sensitivity list of the process. Also, the process has a variable V and a constant C.

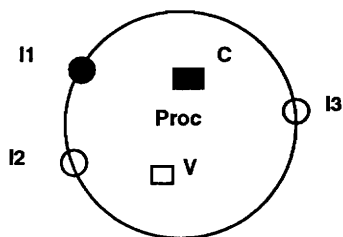


Figure 1: A Process Symbol in a PMG

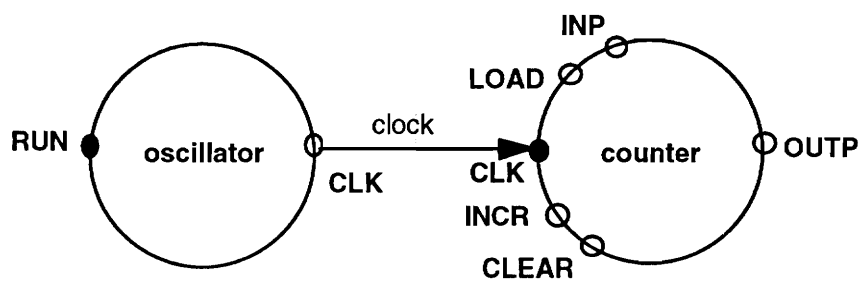


Figure 2: PMG of a Counter Circuit

One or more such processes and interconnecting signals constitute a PMG. The PMG of a counter circuit is shown in Figure 2. The process oscillator represents the oscillator behavior and the process counter represents the behavior of the counter. The signal 'clock' generated by the oscillator forms a clock input to the counter.

3.2 PMG construction and Model generation using Modeler's Assistant

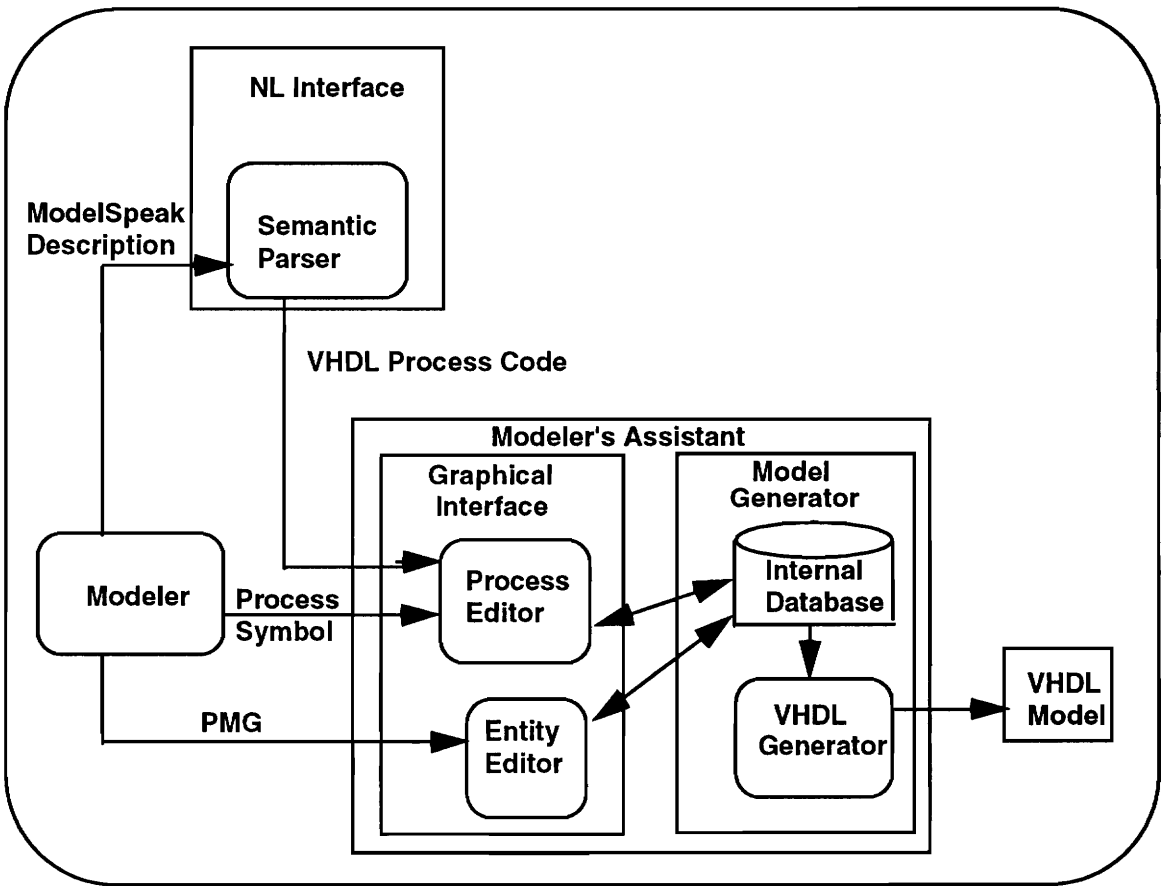


Figure 3 : Overview of the Modeler's Assistant with the NL Interface

Figure 3 shows the overview of the Modeler's Assistant and the NL interface to it. The modeler begins the PMG construction by either selecting processes from a process primitive library or by graphically creating process symbols using the Process Editor. When process symbols are created, information such as the mode and type of entity ports, entity generics, types of variables and constants are entered interactively by the modeler at

this time. Each process symbol created should be accompanied by the VHDL code describing the process. That is, the VHDL code to be included between the begin and end process statements for each individual process that is created has to be input textually. An earlier version of the tool allowed the modeler to construct this code using menus of low level tokens. But this method was done away with as it was inefficient and time consuming. Even the method of directly entering the VHDL code requires the modeler have a good working knowledge of VHDL. Furthermore, the code may involve errors. These factors impede the design process substantially. The Natural Language Interface described here enables the modelers to enter the behavior of processes in a restricted form of English called ModelSpeak and the corresponding VHDL is generated. The names used in ModelSpeak description must match the ones used in defining the process symbol.

Next, the modeler constructs the PMG or the unit using the Entity Editor. This is done by selecting the previously created process symbols stored in the internal database. The processes are then interconnected by signals so as to appropriately model the system. The final PMG is then stored in the database. To generate the VHDL model for the PMG, the VHDL generator is invoked. The VHDL generator extracts information from the PMG and assembles it with the VHDL process codes to generate the VHDL code for the entire model. The behavioral model generated is a single entity containing one architecture body. The architecture body contains the code for the processes generated from natural language descriptions. Figure 4 gives the VHDL behavioral model generated by the Modeler's Assistant for the Counter Circuit of Figure 2. In this model, the entity name COUNTER_SYS is the same as the PMG name specified by the modeler while building the PMG. The entity ports are those to which no signals were connected. Since the port CLK in oscillator and counter are connected by the 'clock' signal, they are not included in

the entity port list but is declared as a signal in the architecture. Also the port names in the processes are replaced by the signal names to which they are connected to. The architecture name is 'BEHAVIOR' by default.

```
entity COUNTER_SYS is
    port(RUN: in BIT; LOAD: in BIT; INCR: in BIT; CLEAR: in BIT;
         OUTP: out BIT_VECTOR(3 downto 0);
         INP: in BIT_VECTOR(3 downto 0);
    end COUNTER_SYS;

architecture BEHAVIOR of COUNTER_SYS is
    signal clock: BIT;
begin

    oscillator : process(RUN)
        begin

            *** VHDL code for the Oscillator ****

        end process oscillator;

    counter : process(clock)
        begin

            *** VHDL code for the counter ***

        end process counter;

end BEHAVIOR;
```

Figure 4: VHDL Behavioral model of the Counter circuit generated by the Modeler's Assistant

Chapter 4

The NL Interface

4.1 ModelSpeak: The Input Language

The NL interface accepts ModelSpeak, a restricted form of English, as the input language. ModelSpeak is designed as a restricted language since the behavioral descriptions of process nodes can be expressed using a limited set of English constructs. The most important criterion on which the ModelSpeak Language is based is that each construct is composed of certain concepts that are readily interpreted and directly mapped to VHDL code. These constraints permit ModelSpeak, to be represented by two components, a vocabulary and a semantic grammar. The vocabulary is a set of words generally used in behavioral description of processes and includes many keywords from VHDL. The current vocabulary is derived from published descriptions of digital components found in manufacturer's data sheets, books on modeling and computer systems. The semantic grammar is based on the concepts used in the descriptions. These are characterized from the VHDL syntax given in the IEEE VHDL Language Reference Manual [1]. Another important consideration is that the description should refer to specific port names and generics which match those used while naming the ports and generics of a process in a PMG. This is because the interface does not rely on any other component of the Modeler's Assistant for the information such as port names and generics, rather it solely

extracts the information from the English text input. Example constructs are given below and the usage of vocabulary and concepts to build limited constructs are explained also.

ModelSpeak sentences specifying `signal_assignment_statement` or `variable_assignment_statement`:

The buffer transfers the data from the input port DI to the output port DO. (11)

The controller asserts the INT signal. (12)

The variable NUM is incremented. (13)

S1 is [not MD or STRB]. (14)

Sentence (11) contains the word '*transfers*' which is the action to be performed. The 'actor' causing the action is the *buffer*, but this information does not have to be used while generating the signal assignment statement. Sentence (12) contains the word '*asserts*' implying the assignment of a value '1' to the target *INT*. Here too, there exists an actor, *controller*, for the action to be performed and this identifier is discarded too for the reason mentioned above. Sentence (13) is an example of a sentence in passive voice. Also, the keyword '*variable*' suggests that '*NUM*' is a variable and not a signal. In ModelSpeak, all identifier names (or unrecognized words) are assumed to be of type signal unless the user explicitly specifies it to be a variable. Also, the keyword '*variable*' must precede the variable name and must not follow it. If this is ignored the program misinterprets the sequence of concepts and results in a failure to generate the VHDL. Sentence (14) is an example of an assignment of a boolean expression to signal *S1*. In ModelSpeak, a boolean expression must be enclosed within square brackets. Also, a space must separate the brackets from the terms of the expression.

ModelSpeak sentences specifying if_statements:

When DMAREQ is set, the controller activates the BR line. (15)

Assertion of DMAREQ activates the BR line. (16)

In example(15), the word '*when*' implies an if_statement. The phrase '*DMAREQ is set*' is the condition-expression and the phrase '*the controller activates the BR line*' is the signal assignment to be performed. Therefore, this sentence structure suggests a conditional signal assignment process in VHDL. Example (16) also suggests the same if_statement to be generated as in example (15). Here, the phrase '*Assertion of DMAREQ*' forms the condition expression without the explicit usage of the word '*When*' and it checks for the rising edge of DMAREQ. Also, it is the cause of the following signal assignment action, suggested by the phrase '*activates the BR line*'. Thus the sentence structure of example (16) does not have an identifier specified as the actor.

ModelSpeak sentences specifying if_statements with 'else' and 'elsif' conditions.

When ENBLD equals '1', the register latches the value of REG to the output port DO otherwise it latches "11111111" on the output port DO. (17)

When RD signal is high, the CPU transfers the contents of MEM(ADDR) to the DATA bus, alternatively if WR signal is high the CPU transfers the data from the DATA bus to MEM(ADDR). (18)

The example (17) implies an if_statement, but is followed by the word '*otherwise*' implying an 'else' clause to be appended to the if_then_statement. Example (18) is also an if_statement, but the phrase '*alternatively if*' suggests an 'elsif' condition to accompany

the `if_statement` that is generated. Also, in ModelSpeak, when bits and bit vectors are implied they have to be enclosed within single quotes and double quotes, respectively as shown by '1' and "11111111" in example (17).

ModelSpeak sentence specifying `nested_if`:

While CS remains low, if RD is high the data from the DATA bus is transferred to the MBR register, otherwise WR is high the contents of MBR register is transferred to the DATA bus. (19)

In any ModelSpeak description the keyword '*while*' implies a `nested_if` and not a `while_loop`. Thus, a `nested_if` is interpreted for the above description. The condition-expression following the word '*while*' is considered to be the condition for the outer `if_statement` and the rest of the sentence after the condition forms the inner `if_statement`. Here, the word '*otherwise*' implies an '*else*' condition, pertaining to the inner `if_statement` by default.

ModelSpeak sentence specifying while_loop:

This is repeated continuously until INT is reset. (20)

The above construct implies that a certain set of tasks in the description preceding the construct have to be repeated. But, in this case ambiguity arises as to which of the tasks have to be repeated. An user-interaction feature is incorporated in the parser to resolve the ambiguity. This is explained in detail in the next section.

To indicate a for_loop the above example is modified as:

This is repeated 4 times. (21)

A similar ambiguity arises in this case too and is dealt in the same way as in the case of the while_loop.

The type of sentence and paragraph constructions described in these examples form a basis for the ModelSpeak semantic grammar rules. A later section explains the concept of the semantic grammar and its rules in BNF notation.

4.2 The Parser

4.2.1 Language Theory

In this section, some terminology of language theory [14,15] is introduced, along with a brief explanation of how it relates to the research work described in this thesis.

An *alphabet* of a language is defined as the set of all possible indivisible symbols of the language. A *language* is defined over an *alphabet* as a subset of a set consisting of all strings, where the strings are obtained by concatenating one or more symbols from the alphabet. The general English language is defined over the alphabet consisting of { a -z 0 - 9 . , ' - / () " ? ! } symbols and strings of the language is formed by the English vocabulary.

Here, in this thesis we deal with two different languages, the ModelSpeak language and the formal VHDL language. ModelSpeak is a restricted form of English, in that the vocabulary (alphabet) of the ModelSpeak language (Appendix B) is a subset of the English vocabulary. Also, the set of strings or the constructs that are acceptable to ModelSpeak (Appendix C) is a small subset of that used in general English.

A grammar G is defined as an quadruple $G = \langle V, T, S, P \rangle$ where

V is a finite set of variables(*non-terminals*)

T is a finite set of *terminal* symbols

S is a special non-terminal that denotes a *sentence* or a *top-level construct*

P is a finite set of *productions*

A *grammar rule* or a *production* consists of a symbol on the left-hand side, and a sequence of constituents that make up the symbol on the right hand side. The symbols that can be replaced by their constituents are the *non-terminal symbols* in the grammar. The *terminal symbols* occur only on the right hand side of the rules and are the actual words of the language. Also, these words are stored under the category of one or more non-terminal symbols and matched against the words of the input sentence.

A grammar $G = \langle V, T, S, P \rangle$ is *context-free* if all productions of G have the form :

$$A \rightarrow X$$

where A is a non-terminal and X is a string of zero or more terminals and non-terminals.

Both ModelSpeak and VHDL have grammars that are context-free since their productions have the form as shown above.

Traditionally, grammar rules are used for *recognizing* and *parsing* sentences. These rules are referred to as the *recognition grammar*. The semantic parser developed in this thesis has incorporated within a single module, the *recognition grammar* for recognizing ModelSpeak sentences and a *generation grammar* to produce corresponding VHDL statements. The special non-terminal, S , in a grammar typically represents a sentence, but in ModelSpeak the special non-terminal symbol is `proc_stat`, which represents a paragraph describing a process.

4.2.2 Traditional Syntactic Parser

A *parser* consists of a collection of grammar rules and a engine for applying rules to the input string. As described above, the non-terminal symbols are made up of other constituents and the terminal symbols that comprise the *vocabulary*. A *parse tree* is generated by attaching suitable grammatical categories to each of the sentence elements.

Figure 5a shows the syntactic grammar rules, in the BNF notation, to parse a simple sentence:

The device asserts the STRB signal. (22)

In BNF notation, the rules or the productions consist of both terminals and non-terminals. The non-terminals or the categories are within angle brackets in bold. Optional elements are enclosed within square brackets and alternate concepts are separated by the symbol '|'. In the syntactic grammar, the non-terminals are the grammatical categories which include the noun_phrase and the verb_phrase. These further break down into other constituent categories, like the determiner, noun, verb and so on. The terminals are the actual words used in the sentences. Figure 5b shows the grammatical structure tree for a sentence parsed by the rules shown in 5a. Generally such parse trees need to further undergo a semantic analysis step in order that the meaning of the input sentence is derived.

```

<sentence> ::= <noun_phrase> <verb_phrase>

<noun_phrase> ::= <determiner> <noun>

<verb_phrase> ::= <verb> <noun_phrase>

<determiner> ::= the

<noun> ::= device <id> (In this example, STRB is the id)

<verb> ::= asserts

```

Figure 5a: Grammar rules in BNF notation to parse a simple sentence

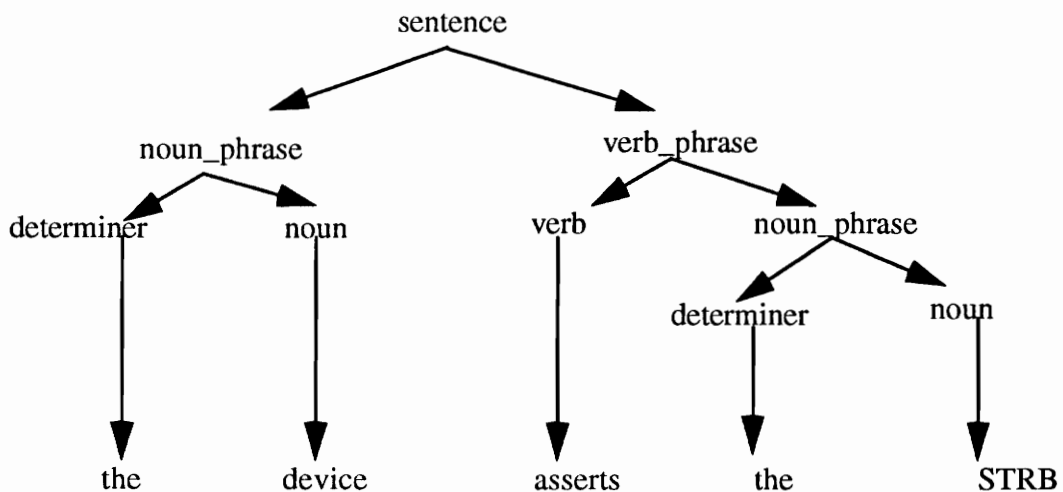


Figure 5b: A Grammatical structure tree for a simple English sentence

Two methods of analysis are possible: *Top-down* and *bottom-up* parsing. Each of these methods can either look for the different possibilities of parse trees in two ways:

- (a) Keeping *Parallel* track of all possibilities.
- (b) *Sequential* or *depth - first method*, in which case backtracking has to occur when a possibility fails.

In *top-down procedure*, the analysis begins with top-most structure in the language, generally a sentence, and it proceeds to examine its constituents. It further tracks down

by expanding the constituents until it reaches the lowest level of the terminal-symbols. Then, it matches the terminals with the words in the input sentence.

In a *bottom-up procedure*, instead of beginning with the top-level rules the analysis process begins with the words of the sentence. Then it compares these with the sequence sequence of constituents on the right-hand side. If a match is found the words are combined into a higher-level constituent whose left-hand side is matched. Then, these combine with each other until the words of the sentence can be identified to belong to one valid sentence(S) symbol.

When the words in the input sentence match the categories forming the sentence, in the appropriate sequence, the parse tree for that sentence can be generated. The parser could generate one or more parse trees, depending on the number of interpretations of the input sentence. Thus, parsers consist of recognizing procedures which are augmented to store the recognized structures as well. Keeping track of these structures help in the *generation* of parse trees.

4.2.3 VHDL Semantics

In VHDL, *processes* are concurrent statements that describe the behavior of interconnected blocks in a design. These *processes* either execute in parallel or in the order controlled by the occurrence of signals in the *sensitivity list* or clauses in the *wait_statement*. When an event occurs on the signals in the sensitivity list or when the conditions in the *wait_statement* are met, the process is triggered and is executed. Within each process is a *process_statement_part* that describes some part of the overall design

and it consists of sequential statements. *Sequential_statements* are primarily used to describe an algorithm and they execute in the order in which they appear.

Signal_assignment_statement assigns a value to a signal, the *signal* being an object that holds all previously assigned values as well as for the future. Signals can be driven by more than one process in which case the values assigned to it have to be resolved or multiplexed. Unlike signals, *variables* are objects that hold only one value, currently assigned to it. To differentiate these assignments, we have the symbols, $\leq$ and $:=$ for *signal_assignments* and *variable_assignments*, respectively.

An *if_statement* executes a sequence of statements whose corresponding condition is satisfied. The *condition_expression* that it checks for must be of type BOOLEAN. A *loop_statement* encloses a sequence of statements that are to be repeatedly executed. Depending on the *iteration_scheme*, it is either a *while_loop* or a *for_loop*. A *while_loop* executes the statements within the loop while the condition is satisfied. A *for_loop* executes the enclosed statements for a fixed number of times. A *wait_statement* temporarily suspends the process execution. The time period for which it stays suspended is dependent on the clauses specified in the statement. The *sensitivity_clause* specifies the signals on which the process is sensitive and an event on any one of them should trigger a process, provided the *condition_clause*, if present, is satisfied. The *time_out_clause* specifies a maximum time the process remains suspended.

The subset of VHDL [1], that is used by the semantic parser is shown in BNF form below. This has formed the basis for developing the ModelSpeak language and also the semantic

grammar rules that parse these sentences. The keywords in the VHDL syntax are shown in bold.

```
process_statement_part ::=
    sequence_of_statements
```

```
sequence_of_statements ::=
    {sequential_statement}
```

```
sequential_statement ::=
    wait_statement |
    signal_assignment_statement |
    variable_assignment_statement |
    if_statement |
    loop_statement .....
```

```
signal_assignment_statement ::=
    target <= waveform;
```

```
variable_assignment_statement ::=
    target := expression;
```

```
if_statement ::=
    if condition then
        sequence_of_statements
    { elsif condition then
        sequence_of_statements }
    [else
        sequence_of_statements]
    end if;
```

```
loop_statement ::=
    [iteration_scheme] loop
        sequence_of_statements
    end loop;
```

```
iteration_scheme ::=
    while condition
    | for loop_parameter_specification
```

```
wait_statement ::=
```

```

wait [sensitivity_clause] [condition_clause] [time_out_clause];

sensitivity_clause ::=
    on sensitivity_list

condition_clause ::=
    until condition

time_out_clause ::=
    for time_expression

target ::=
    name

wave_form ::=
    {wave_form_element}

wave_form_element ::=
    value_expression [after time_expression]

```

4.2.4 Parsing and Generation with Semantic grammar

Since the ModelSpeak sentences have limited structure and vocabulary, they can be viewed in terms of non-terminals based on VHDL concepts (described in section 4.2.3) rather than the conventional grammatical categories. The non-terminals are the semantic concepts and the parser is based on the rules that searches for these concepts. The semantic concepts included are sig_assgn, if_stat, actor, action, cond and so on. These non-terminals, along with the terminals, form the *semantic grammar* rules of the parser. Employing this type of grammar integrates the parsing and the semantic analysis into a single module. The VHDL semantics [1] briefly reviewed in the previous section highlight the correspondence of VHDL to the semantic categories used in the parser.

Since the interpretation of the sentence takes place at the initial step itself, the other desired action, that is generation of VHDL statements, can also be made to proceed

concurrently. In this context, the parser may be viewed as a set of rule pairs. Each pair consists of a recognition rule for semantic analysis of the sentence, as described above, and associated with it is a generation rule for VHDL. The generation grammar translates the ModelSpeak sentences into corresponding VHDL statements. A generation rule may sometimes be empty or **nil** when the non-terminal symbol in the recognition grammar has no direct corresponding non-terminal symbol in the generation grammar.

The semantic parser which is implemented in Prolog operates in a top-down, depth-first manner. Since this parser has a limited set of rules, extensive backtracking does not occur as in other cases of depth-first searches. Thus, computation time is apparently instantaneous. Also, using depth-first search strategy offers advantage in saving space since only a single generative structure is stored at one time. This is opposed to a parallel strategy, where the space required to store all alternatives is enormous.

The rule pairs are represented in this section using the BNF notation. Non-terminals for ModelSpeak analysis, such as `<sig_assgn_a>`, are denoted with an `_a` suffix. The corresponding non_terminal `<sig_assgn_g>` of the VHDL generation rule is given a `_g` suffix.

The semantic grammar is built to accept multi-sentence descriptions and the parsing procedure takes place as follows: First, it begins looking for a paragraph that describes a single process. While parsing using the analysis rules, the words in the input sentence are matched with the rules for individual VHDL statements such as rules for a `signal_assignment_statement` or an `if_statement`. Then, the corresponding VHDL statement is generated using the generation rules. Next, using the rules for multi-sentence

descriptions the other consecutive sentences are parsed and the VHDL generated. The current section is divided into two sub-sections. The first sub-section describes the rules for *isolated sentences*. The second sub-section section deals with the rules for *multi-sentence descriptions*.

(a) Semantic Rules for Isolated sentences

For sentences such as *The device asserts the STRB signal* (22) the ModelSpeak analysis rules that implement semantic interpretation and their corresponding generation rules are shown side by side in Figure 6a, with the generation rules in bold. Figure 6b gives a diagrammatic tree representation of the sentence interpretation and VHDL generation using the semantic rule pairs (that is, the analysis rules and the generation rules in bold) shown in Figure 6a.

<code><sig_assgn_a> ::= [<determiner_a>] <actor_a></code>	<code><sig_assgn_g>::=</code>
<code> <action_a> [<determiner_a>] <target_a></code>	<code><id_g><= <value_g> ;</code>
<code><determiner_a> ::= {the a}</code>	<code>nil</code>
<code><actor_a> ::= Id</code>	<code>nil¹</code>
<code><action_a> ::= {<action0_a> <action1_a>}</code>	<code>nil</code>
<code><action0_a> ::= {resets deasserts clears}</code>	<code><value_g> ::= '0'</code>
<code><action1_a> ::= {sets enables asserts}</code>	<code><value_g> ::= '1'</code>
<code><target_a> ::= [<type_a>] Id [<class_a>]</code>	<code><id_g> ::= Id</code>
<code><type_a> ::= {input output}</code>	<code>nil</code>
<code><class_a> ::= {signal port line bus}</code>	<code>nil</code>

Figure 6a: Analysis and Generation rules for a signal assignment statement of type (1)

¹nil refers to an empty generation rule.

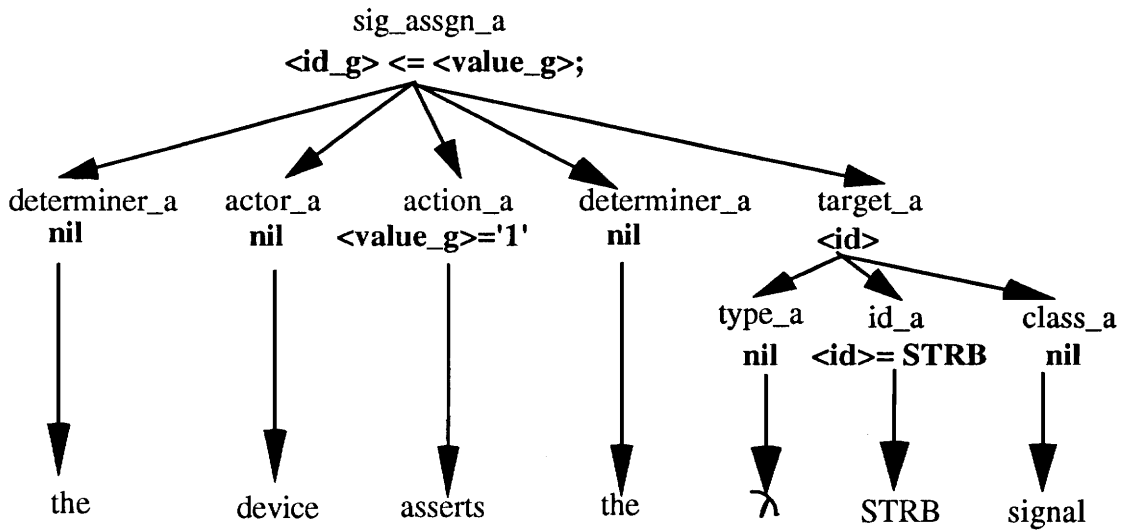


Figure 6b: The semantic structure tree for a signal assignment statement expressed in ModelSpeak

The analysis rule `<sig_assgn_a>`, as shown above, parses for its constituent concepts: `<determiner_a>`, `<actor_a>`, `<action_a>` and `<target_a>`. These constituent concepts, or the non-terminals, may further break down into sub-constituents. In the above example, the concept `<target_a>`, can be replaced by the concept `<type_a>`, followed by the Id and the concept `<class_a>`. But the other constituent concepts `<actor_a>`, `<determiner_a>` and `<action_a>` are to be matched directly with the terminals or actual words from the input sentence. Terminals also include identifiers which are certain specific names used in the description but not listed in the ModelSpeak vocabulary. When the terminals listed under any of the concepts match the words being parsed in the input sentence the concept is said to be satisfied and the parsing of the next concept follows. Let us analyze the parsing process using the above set of rules. The word 'The' of the input sentence satisfies the concept `<determiner_a>` which expects terminals such as 'the' or 'a'. The concept `<actor_a>` accepts any identifier for the terminal Id. An Id is either captured or discarded depending on the context. In this example, the word 'device' is the Id but is discarded as it

has no use in the VHDL generation. Next, the concept <action_a> is examined and it is found that it can be replaced by either the concept <action0_a> or the concept <action1_a>. According to the grammar rules, <action0_a> is first checked. Since the input word 'asserts' does not match any of the terminals listed under this concept, the parsing of this concept is said to have failed. The program retries satisfying <action_a> by examining <action1_a> and the terminals listed under it. This time <action1_a> succeeds as 'asserts' is listed as one of the acceptable terminals. The concepts <action0_a> and <action1_a> have associating generation rules. The <value_g> rule associated to <action0_a> returns a value '0', in consistent with the terminals such as 'clears', 'resets' and 'deasserts' that imply a value '0'. Similarly, the concept <action1_a>, expecting terminals such as 'sets' and 'pulses', is associated with the generation rule <value_g> that returns the value to be assigned as '1'. The next concept <determiner_a> is satisfied by the word 'the'. Finally, the last of the constituent concepts, <target_a> is examined. As mentioned before, this concept is further relaced by other concepts. The <type_a> is optional, indicated by being enclosed within brackets. Words such as 'input' or 'output' to describe the target signal belong to this concept type, but this example sentence does not have such a word. The word 'STRB' satisfies the Id, which is expected. Then the generation rule <id_g> returns the actual signal name, Id, to which the value is to be assigned. Note that in this case, the Id is retained as it is the actual signal name to be used in the signal assignment statement. The <class_a> concept is then examined that expects any one of the terminals listed under it. The word 'signal' matches one of them and hence satisfies <class_a>. Having parsed the sentence by applying the <sig_assgn_a> analysis rule, the corresponding generation rule <sig_assgn_g> is invoked. Therefore, in the generation step, the values returned by <target_a> and <action1_a> from the analysis rule of <sig_assgn_a> are instantiated in the variables <id_g> and <value_g> of <sig_assgn_g>.

Therefore, for the sentence *"The device asserts the STRB signal"*, the generation rule of $\langle \text{sig_assgn_g} \rangle$, that is $\langle \text{id_g} \rangle \leq \langle \text{value_g} \rangle ;$, is instantiated as $\text{STRB} \leq '1'$; and hence the required VHDL for the sentence is generated. The concept $\langle \text{sig_assgns_a} \rangle$ is also a one of the constituent concepts of the $\langle \text{if_stat_a} \rangle$ rule.

As seen in the above example structure, we do not have an explicit value specified in the description that can be assigned to a target. Instead we have the concept 'action' which implies a certain value to be generated and this is then assigned to a target on the left hand side.

$\langle \text{sig_assgn} \rangle ::= [\langle \text{determiner_a} \rangle] \langle \text{actor_a} \rangle \langle \text{actn_a} \rangle$	$\langle \text{sig_assgn_g} \rangle ::=$
$[\langle \text{determiner_a} \rangle] \{ \langle \text{source_a} \rangle \mid \langle \text{value_a} \rangle \}$	$\langle \text{id2_g} \rangle \leq \langle \text{id1_g} \rangle ;$
$\langle \text{prep_a} \rangle [\langle \text{determiner_a} \rangle] \langle \text{target_a} \rangle$	
$\langle \text{actor_a} \rangle ::= \text{Id}$	nil
$\langle \text{actn_a} \rangle ::= \{ \text{transfers} \mid \text{applies} \mid \text{latches} \mid \text{transmits} \mid \text{puts} \}$	nil
$\langle \text{source_a} \rangle ::= \{ \text{data} \mid \text{contents} \mid \text{value} \} \langle \text{prep_a} \rangle$	
$[\langle \text{determiner_a} \rangle] \langle \text{type_a} \rangle \text{Id} \langle \text{class_a} \rangle$	$\langle \text{id1_g} \rangle ::= \text{Id}$
$\langle \text{value_a} \rangle ::= \text{Value}$	$\langle \text{id1_g} \rangle ::= \text{Value}$
$\langle \text{target_a} \rangle ::= [\langle \text{type_a} \rangle] \{ [\langle \text{class_a} \rangle] \text{Id} \mid \text{Id} \langle \text{class_a} \rangle \}$	$\langle \text{id2_g} \rangle ::= \text{Id}$
$\langle \text{type_a} \rangle ::= \{ \text{input} \mid \text{output} \}$	nil
$\langle \text{class_a} \rangle ::= \{ \text{signal} \mid \text{port} \mid \text{line} \mid \text{bus} \}$	nil

Figure 7: Analysis and Generation rules for a signal assignment statement of type (2)

For signal assignment statements expressed differently, such as, *"The buffer transfers the data from the input port DI to the output port DO"*, (11) we have a different set of analysis and generation rules as shown in Figure 7. In this structure, the value to appear in the right hand side of the signal assignment to be generated, is explicitly specified in the input ModelSpeak sentence. In the current example, the identifier, DI, is to be captured and used as a value on the right hand side of the signal assignment.

The top-level analysis rule for `<if_stat_a>` and the generation rule `<if_stat_g>`, associated to it are shown in Figure 8a. the lower-level analysis and generation rules for the `if_statement` are shown as in Figure 8b.

<pre> <if_stat_a>::={when if }<cond_a> <sig_assgns_a> [<elsif_a>] [<else_a>] </pre>	<pre> <if_stat_g> ::= if <cond_g> then <sig_assgns_g> [<elsif_g>] [<else_g>] end if; </pre>
--	--

Figure 8a : Top-level Analysis rules and Generation rules for an if_statement.

When the analysis rules for `<if_stat_a>` are applied to a sentence, the parser searching is done for the concepts `<cond_a>` , `<sig_assgns_a>` and the other optional concepts, if any. When the analysis rule for each of the constituent concepts of `<if_stat_a>` is satisfied, the corresponding generation rule is applied to construct the appropriate VHDL code. This completes the analysis step for `<if_stat_a>` where upon the generation rule `<if_stat_g>` is

invoked, where the code generated by the constituent concepts are instantiated in the appropriate slots.

$\langle \text{cond_a} \rangle ::= \langle \text{var_a} \rangle \langle \text{rel_a} \rangle \langle \text{val_a} \rangle$	$\langle \text{cond_g} \rangle ::= \langle \text{id_g} \rangle$ $\langle \text{rel_g} \rangle$ $\langle \text{val_g} \rangle$
$\langle \text{var_a} \rangle ::= \text{Id}$	$\langle \text{id_g} \rangle ::= \text{Id}$
$\langle \text{rel_a} \rangle ::= \{ \text{equals} \mid \text{is} \}$	$\langle \text{rel_g} \rangle ::= =$
$\langle \text{rel_a} \rangle ::= \{ \text{does not equal} \mid \text{is not} \}$	$\langle \text{rel_g} \rangle ::= /=$
$\langle \text{val_a} \rangle ::= \{ \text{high} \mid '1' \mid \text{set} \}$	$\langle \text{val_g} \rangle ::= '1'$
$\langle \text{val_a} \rangle ::= \{ \text{low} \mid '0' \mid \text{reset} \}$	$\langle \text{val_g} \rangle ::= '0'$
$\langle \text{elsif_a} \rangle ::= \{ \text{alternatively} \mid \text{otherwise} \} \{ \text{if} \mid \text{when} \}$ $\langle \text{cond_a} \rangle \langle \text{sig_assgns_a} \rangle$	$\langle \text{elsif_g} \rangle ::= \text{elsif}$ $\langle \text{cond_g} \rangle \text{ then}$
$\langle \text{else_a} \rangle ::= \{ \text{otherwise} \mid \text{or else} \} \langle \text{sig_assgns_a} \rangle$	$\langle \text{else_g} \rangle ::=$ $\langle \text{sig_assgns_g} \rangle$

Figure 8b: Lower-level Analysis and Generation rules for the if_statement

(b) Semantic rules for Multi-Sentence descriptions

According to the VHDL syntax, a typical process in VHDL consists of one or more sequential statements. The most common of these sequential statements are signal assignment statements, if statements, loop statements and wait statements. Since the input to the NL interface is a description of a process, the semantic grammar is built to incorporate the feature of multi-sentence descriptions. That is, the description can either be a single sentence or cross sentence boundaries. In a multi-sentence description, the correct sequence of VHDL code is generated after checking for appropriate sentence connectivity.

In Figure 9 are shown the semantic grammar - analysis rules that permit multi-sentence descriptions. Here, the rule `<proc_stat_a>`, denoting a process, can be replaced by a sequence of statements, `<seq_stats_a>`. `<seq_stats_a>` in turn, begins with a individual VHDL statement, `<seq_stat_a>`. A `<seq_stat_a>` can be a signal assignment statement `<sig_assgns_a>`, if_statement `<if_stat_1a>`, nested_if `<if_stat_2a>` or a wait statement `wait_stat`. A `<sig_assgns_a>` parses for all consecutive signal assignments without executing `<seq_stat_a>` each time for every consecutive signal assignment. The `<sig_assgns_a>` corresponds to signal_assignment statements and the rule `<next_a>` checks for period and a comma with `<delimiter_a>` and conjunctions using `<conntn_a>`. Then `<next_a>` recursively calls `<sig_assgn_a>` to parse all consecutive signal_assignment statements represented by succeeding sentences in the paragraph. The corresponding generation rule `<sig_assgns_g>` concatenates all the consecutive signal_assignment statements.

If the analysis rules for a particular sentence structure are not satisfied, that is the words in the input sentence do not correlate with the terminals expected by the analysis rules used to interpret the sentence, then, the program backtracks to reanalyze the sentence using a different set of analysis rules. When a particular set of rules are satisfied, the input sentence is identified with a valid VHDL statement and the code for it is also simultaneously generated. Then, the rule `<next_a>` looks for a `<delimiter_a>` such as period or a comma if any, then looks for keywords indicating sentence connectivity(then, also, after that, after this) or conjunctions (and , and then, or). Finally after again checking for a delimiter, the `<seq_stat_a>` rule is executed for the next consecutive statement in the description. Since `<sig_assgns_a>` takes care of all consecutive signal_assignments, the next statement parsed by `<seq_stat_a>` would only be an

if_statement, or a nested_if or a wait_statement. <seq_stat_a> rule is executed repeatedly until the entire description has been parsed and no more statements need to be generated. The generation rules associated with the analysis rules in Figure 9 only concatenate all the statements generated by the rules for the isolated sentences shown in the previous section. The <sig_assgns_a> rule is also embedded within the <if_stat_a> rule.

<proc_stat_a> ::= <seq_stats_a>	<proc_stat_g> ::=
	<seq_stats_g>
<seq_stats_a> ::= <seq_stat_a> <next_a>	<seq_stats_g> ::=
	<seq_stat_g> <more_g>
<seq_stat_a> ::= {<sig_assgns_a> <if_stat1_a>	<seq_stat_g> ::=
<if_stat2_a> <wait_a> <loop_a>}	{<sig_assgs_g> <if_stat1_g>
	<if_stat2_g> <wait_g>
	<loop_g>}
<next_a> ::= [<delimiter_a>] {<conntn_a>	<more_g> ::= <seq_stat_g>
<conjn_a> } [<delimiter_a>]	<more_g>
<seq_stat_a> <next_a>	
<next_a> ::= []	<more_g> ::= []
<delimiter_a> ::= {' ' ',' }	nil
<conntn_a> ::= { then also after that after this }	nil
<conjn_a> ::= { and and then or }	nil
<sig_assgns_a> ::= <sig_assgn_a> <next_a>	<sig_assgns_g> ::=
	<sig_assgn_g> <more_g>

Figure 9: Semantic rules for multi-sentence description

For example, a description crossing sentence boundaries can be written as:

The controller asserts the DMAA signal and puts the value of ADDRESS register on ADDR bus. Then, when RD is high the controller puts the contents of the DI register on the DATA bus. (22)

Here, the <seq_stats_a> rule first executes the <sig_assgns_a> rule. The whole sentence *The controller asserts the DMAA signal and puts the value of ADDRESS register on ADDR bus.* is interpreted as a sequence of two signal assignments with the conjunction 'and' making it a compound sentence. Next, the word *Then* followed by a comma, are interpreted as sentence connections. Finally, the <if_stat1_a> rule is satisfied by matching the concepts obtained from the remaining part of the sentence *when RD is high the controller puts the contents of the DI register on the DATA bus.* Since the entire description has been parsed the <next_a> rule encounters the end of file symbol, [], and the generation rules <seq_stat_g>, <seq_stats_g> and <proc_stat_g> associated to <seq_stat_a>, <seq_stats_a> and <proc_stat_a> respectively, put together the VHDL code for the process.

4.4 The User Interaction Feature

This feature is provided to resolve ambiguities in the scope of either the for_loop or the while_loop and the type of a either the signal or variable declaration. When constructs implying For or While Loops are encountered by the parser, it initiates an interaction with the user before generating the VHDL code. The technique to resolve the scope ambiguity is as follows. The parser looks back through all the sequential statements starting from the most recent one. It displays one statement at a time and prompts the user to answer if that statement was intended to be within the scope of the loop. It continues displaying statements until either the user answers a 'n' or until after the first statement of the

description is prompted. If the user answers a 'n' for a particular statement midway in the backtracking the parser leaves the interaction mode , encloses all indicated statements within the scope of the loop, and then generates the entire code for the process. If the user indicates a 'y' for all statements displayed, then the parser generates the loop statement encompassing all the statements within its scope.

Also, when words such as 'increment' or 'decrement' are used in the description, the ambiguity that arises is whether the user implies an integer increment or a bit vector increment. To resolve this ambiguity it is necessary to know if the variable or signal is declared as an integer or a bit vector. This is done by prompting the user, after the Prolog prompt `l:` , as follows.

`l:` Type 'y' if (signal or variable name) is integer, if not type 'n'.

If the user replies with a 'y' then an expression is generated with integer 1 added to the original signal or variable. Otherwise the function "`incr( )`" is used as the increment function to carry out bit vector increment. The signal or variable is instantiated within the function.

For example, consider the description *if $A(I)$ equals '1', the variable NUM is incremented. This is repeated 3 times.* (24)

The interaction proceeds as follows:

Type 'y' if NUM is integer, if not type 'n'.

`l:`y.

Which of the following statements are to be repeated.

Answer with a 'y' or 'n'.

NUM:= NUM + 1;

l: y.

if(A(I) = '1') then

NUM:= NUM + 1;

end if;

l: y.

Consider the case when the user intends to enclose the entire if_statement within the for_loop. In this example, the immediately preceding statement to the sentence implying the for_loop is NUM:= NUM + 1;. since this statement is a part of the if_statement that was intended to be within the loop the user replies with a 'y'. Then, the entire if_statement was displayed and to this also the user replies with a 'y'. As a result, the VHDL generated for this description would be a for_loop statement enclosing the if_statement as follows :

```
for I in 0 to 2 loop
    if(A(I) = '1' then
        NUM := NUM + 1;
    end if;
end loop;
```

Consider another example shown below, assuming DAR and WCR are declared as bit_vectors. Also, the user intends to include only the two signal_assignments within the while_loop:

When BG is high the contents of DAR is incremented and the value in WCR is decremented. This is repeated continuously until WCR reaches "0000". (25)

The interaction in the above case proceeds as follows:

Type 'y' if DAR is integer, if not type 'n'.

l: n.

Type 'y' if WCR is integer, if not type 'n'.

l: n.

Which of the following statements are to be repeated.

Answer with a 'y' or 'n'.

WCR <= decr(WCR);

l: y.

DAR <= incr(DAR);

l: y.

if(BG = '1') then

DAR <= incr(DAR);

WCR <= decr(WCR);

end if;

l: n.

In this example, again the user is queried about the type of declaration for WCR followed by the type of declaration for DAR. Since the user had declared them as bit_vectors, the user replies with a 'n' for both DAR and WCR. Next, a while_loop is interpreted by the second sentence of example (24). Interaction is initiated again where the statement **WCR <= decr(WCR);** is displayed, to which the user replies a 'y'.

Next, **DAR <= incr(DAR);** is displayed and the user replies a 'y'. But when the if_statement is displayed the user replies a 'n'.

The VHDL generated for this description would be:

```
if(BG = '1') then
    while (not WCR = "0000") loop
        DAR<= incr(DAR);
        WCR<= decr(WCR);
    end loop;
end if;
```

Chapter 5

Program Implementation

5.1 Programming Technique and other added Features

The Semantic Parser module of the NL Interface is written in Quintus Prolog employing the top-down programming technique described in section 4.2.2 and 4.2.4. The Semantic Parser is written using a simple and easy technique illustrated in the textbook, Programming in Prolog by Clocksin and Mellish [14]. It is a special grammar rule notation, using the symbol ' --> '. In this way, much information that is used by the program but which clutters the code is hidden from the view of the programmer. Therefore, rules used in the semantic parser are not in the regular Prolog format of rules and facts. Instead, a more natural way of writing the parser was adopted by using the notation, as shown in Example (a). To compare these two ways, Example (b) shows a few rules in the regular format. Prolog translates the special notation format into a regular format for compilation purposes.

In Example (a), the rules seen explicitly are the analysis rules and the generation associated to them are shown within braces. Some analysis rules have no generation rules associated with them in which case the generation rule is empty.

```

sig_assgn([P1,<=,P2,;]) --> determiner,
                                actor,
                                action(P2),
                                determiner,
                                target(P1).

determiner      --> [the].
determiner      --> [a].

```

Example a

```

sig_assgn(S0,SN,[P1,<=,P2,;]) :-    determiner(S0,S1),
                                actor(S1,S2),
                                action(S2,S3,P2),
                                determiner(S3,S4),
                                target(S4,SN,P1).

determiner([the|S],S).
determiner([the|S],S).

```

Example b

One of the added features of the program includes Spell-check. 'Spell' is a routine provided by the UNIX operating system for spell-checking. The Prolog program makes a system call to this routine so that this feature is automatically invoked by parser, when the ModelSpeak input is typed. Spell checking is done on the ModelSpeak input and a list of misspelled words along with identifiers' names not included in the vocabulary are displayed. This is done because, if the description containing misspelled words is directly parsed then the program either does not produce the VHDL code or even goes into an infinite loop searching for a match. Thus, when the misspelled words are listed, then it becomes convenient to the user to correct the spelling mistake and re-enter his description. Also, the semantic parser writes the output to a file. After the entire description is parsed, the VHDL code is displayed to the user for verification.

Simultaneously, the VHDL code is written to a file and stored in the database of the Modeler's Assistant. The name of this file is provided by the Modeler's Assistant as an argument while invoking the Prolog program. The code generated for each process of the PMG is then used to generate the integrated model represented in the PMG.

5.2 Interface to the Modeler's Assistant

The Modeler's Assistant program developed at Virginia Tech is written in C and is running on the SUN SPARC2 workstation. The Prolog program is developed as a natural language interface to this VHDL modeling tool. An executable file of the Prolog program is developed which behaves as an external routine invoked by the Modeler's Assistant.

The interface program is stored in a file called Process.pl. It is programmed to receive a command line argument which is the name of the file to which the VHDL code has to be written to. The fully developed program was then compiled using the Quintus command:

qpc -c Process.pl

The -c option compiles the program into a QOF (Quintus Object Format) file and stops. The Object file obtained was Process.qof.

Next, Process.qof was linked using the qld(QOF Link eDitor) command:

qld -D -o Process Process.qof

The -D option enables linking the program with the Development Kernel rather than the Run-time Kernel. The -o option gives the name of the executable file. Here, 'Process' is the name of the executable file.

The final executable file "Process" is then invoked by the Modeler's Assistant when the user clicks on the menu 'English' while creating a process. This is done separately for every process in the PMG. The Modeler's Assistant also passes a file name the program, to which the output has to be written to. A window pops up and runs the "Process" executable. This loads Prolog and brings up the Prolog prompt |: At this prompt the program is initiated by typing 'description'. After the program parses the entire description, the output is sent to the file stored in the internal database of the Modeler's Assistant.

Chapter 6

Results

In this section, the example models are taken from the textbooks Computer System Architecture [9], Structured Logic Design with VHDL [15] and VHDL:Hardware Description and Design [16]. The main purpose of the examples is to illustrate the style of the ModelSpeak input language in describing the process of a model and the effectiveness of the interface in aiding the Model development process. The examples are a DMA Interface Model, a UART model, a Three Module System, a MUX primitive, a Parallel I/O port model and a State Machine of a Traffic Light Controller.

6.1 The DMA Interface Model (A detailed example)

We have taken a Direct Memory Access(DMA) interface module as an example. This model has been described in the textbook Computer System Architecture [9]. It forms a part of the DMA transfer in a microprocessor system. One of the reasons why this particular model was chosen for detailed illustration is because the ModelSpeak language used to describe the processes in this example is similar to the description of the model given in the book, with certain slight modifications such as inclusion of specific port names. This also as a hypothesis that since designers are familiar with this form of

description in such textbooks, it may require very little time and effort on their part to get accustomed to this type of an interface as opposed to using the formal VHDL language.

The DMA interface module can be functionally partitioned into three processes [10]. These are

- 1. The response of the DMA controller to the bus request sent from the peripheral device.
- 2. The initialization done by the microprocessor for the transfer.
- 3. The transfer carried out by the controller when the Bus Grant signal is made high.

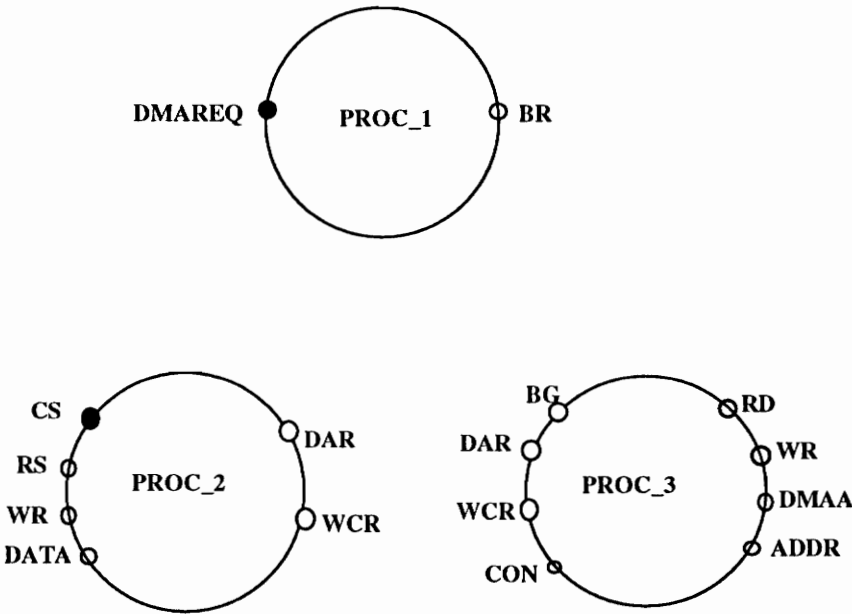


Figure 10: Process symbols representing the partitions of the DMA interface model

The processes created by the modeler using the Modeler's Assistant's graphical interface are as shown in Figure 10.

The NL description of each of these processes are the following:

Process_1: The PMG shows a process named Proc_1 having an input port² DMAREQ and an output port BR. The process has DMAREQ in the sensitivity list, so indicated by a darkened circle. To request a DMA transfer, the peripheral device asserts the DMAREQ, and the DMA responds by communicating this request to the microprocessor through the BR (Bus Request) signal. Here, it is assumed that the DMAREQ is driven by a process corresponding to the peripheral device module not shown in the diagram. The English description for the behavior of this process can hence be written in ModelSpeak as:

"When DMAREQ is set, the controller activates the BR line."

The VHDL generated by the NL interface is then:

```
if( DMAREQ = '1') then
    BR <= '1';
end if;
```

Process_2: The Process named Proc_2 has input ports CS, RS, WR and DATA and the output ports DAR and WCR. CS is in the sensitivity list of this process. Here, CS and RS are chip Select and register Select bits in the controller, initialized by the microprocessor. The microprocessor also makes the WR signal high so that the starting address and word count are communicated to the DMA controller through the DATA bus. Here too, it is assumed that CS, RS and WR are signals driven by a process corresponding to the microprocessor module.

The NL description for this process can be written in ModelSpeak as:

²Here, port refers to a process input or output

"While CS is low and WR is high, if RS is set the controller transfers the data from the DATA bus to the DAR register otherwise the controller transfers the data from the DATA bus to the WCR register. "

The VHDL generated by the NL interface for this description is

```
if(CS = '0' and WR = '1') then
    if (RS = '1') then
        DAR <= DATA;
    else
        WCR <= DATA;
    end if;
end if;
```

Process_3: The process named Proc_3 is not as simple as the other two processes. The input ports are BG, DAR ,WCR and CON and the output ports are RD, WR, DMAA, and ADDR. The DMA transfer begins after the microprocessor relinquishes the bus and makes the Bus Grant signal, BG, go high. Therefore, this function is modeled as a process, which waits for signal BG to go high. The CON signal is either set or reset by the microprocessor to indicate either a read or a write from the memory and hence can also be assumed to be driven by a process, again not shown in the diagram. Based on its value the controller initiates a RD or a WR . Then, it sends a DMA acknowledge signal, DMAA, to the peripheral device and simultaneously puts the appropriate memory address on the address bus, from or to which the transfer has to occur. After each transfer the DMA controller increments the DAR register and decrements the WCR register. And the transfer is completed when WCR reaches zero. For this example let us consider that the DAR and WCR ports of processes PROC_1 and PROC_2 are of type BIT_VECTOR of length 4 bits. The above description of the behavior of the process can be rephrased to be input to the NL interface as follows:

" The process waits on BG until BG equals 1. While BG remains high, if CON equals 1 the controller initiates the RD signal otherwise the controller initiates the WR signal. Also, when BG is high the controller asserts the DMAA signal, puts the value of the DAR register on the ADDR bus and waits for 5 ns. Then, the controller increments the DAR register and decrements the WCR register. This is repeated continuously until WCR reaches "0000". After this the controller activates the INT line."

Since the WCR is of width 4-bits the end condition for the while_loop has to be specified as "WCR2 reaches "0000" and not as "WCR2 reaches zero". Note that the description specifies the condition "BG is high" twice, once to specify the nested if_statement and the second time to specify that the rest of the sequential statements are to be executed while the same condition, "BG is high" , holds true. It is obvious that the second check is redundant but is necessary for the following reason. When there is a situation of a nested-if (i.e. an inner if_statement nested within another if_statement) followed by some sequential statements, then ambiguity arises as to whether the sequential statements are within the if_statement or not a part of it. The parser adopts the strategy of associating the sequential statements with the immediately preceding if_statement. But, if the user intends a different association he or she must explicitly specify the outer if condition again, which when met causes the subsequent sequential statements to be executed.

As a result of parsing the description for PROC_3, the user enters an interaction mode. Here, we assume that the user intends the following situation for each transfer. The DMA communicates the address to the bus and waits for the transfer to take place. Then, it increments the DAR register and decrements the WCR register. Meanwhile, the RD or WR signal is held high along with the DMAA signal until the end of the transfer. Based

on this, the interaction is assumed to proceed as shown below. Here, the sign, |:, is the Prolog prompt at which the user replies with a 'y' or 'n'.

Type 'y' if DAR is integer, if not type 'n'.

|: n.

Type 'y' if WCR is integer, if not type 'n'.

|: n.

Which of the following statements are to be repeated.

Answer with a 'y' or a 'n'.

WCR <= decr(WCR);

|: y.

DAR <= incr(DAR);

|: y.

wait for 5 ns;

|: y.

ADDR <= DAR;

|: y.

DMAA <= '1';

|: n.

The interaction terminates here and the VHDL code below for the process description is generated.

```

wait on BG until BG = '1';
if (BG = '1') then
    if (CON = '1') then
        RD <= '1';
    else WR <= '1';
    end if;
end if;

if( BG = '1') then
    DMAA <= '1';
    while not WCR = "0000" loop
        ADDR <= dar;
        wait for 5 ns;
        DAR <= incr(DAR)3;
        WCR <= decr(WCR);
    end loop;
    INT <= '1';
end if;

```

To draw a Process Model Graph (PMG) of the above model, one would connect the process port DAR of Proc_2 to the port DAR of Proc_3 with a signal. Similarly, one would want to connect the ports WCR of the two processes. This is not allowed by the Modeler's Assistant since signals DAR and WCR are unresolved ports. That is, they are assigned values by two processes(Proc_2 assigns the value DATA to DAR and WCR and Proc_3 increments and decrements the values of DAR and WCR, respectively.) To avoid conflict between the values assigned by the two drivers, the two processes that are developed individually can be merged. The final entity model (shown in Appendix D) is shown having the processes containing the signals DAR and WCR, merged. Due to this merging, a slight modification has to be done to the VHDL model. The change is

³It is assumed that 'incr' and 'decr' are in-built procedures in VHDL which the parser automatically associates with words such as increment and decrement in the description.

that the wait statement should wait on an event on both BG and CS signals, since the two processes that contained them are merged. This change is manually done and the modified model is included in Appendix D, but note that this not not automatically handled by the semantic parser.

6.2 The Example of a Three Module System

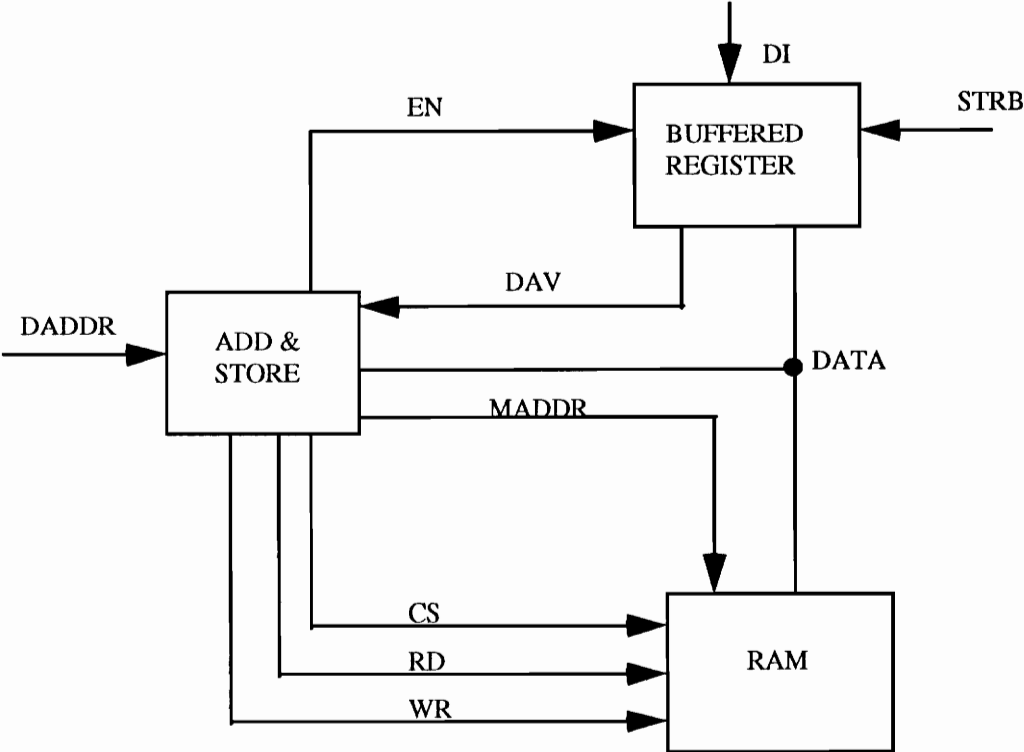


Figure 11a: Three Module system

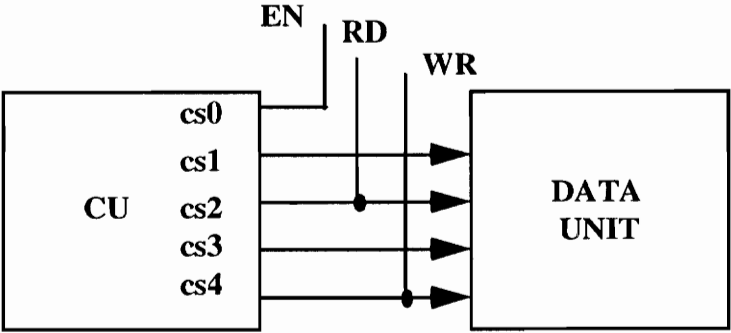


Figure11b: Functional Partitions of Add and Store Unit

Figure 11a is the block diagram of a three module system taken from the textbook "Structured Logic Design with VHDL" [17]. The Add & Store Unit reads in the data from the buffered register and stores it in an internal register. Next, it reads data from a RAM location and performs an addition of the data from the RAM and the data in the internal register. Finally, it stores the result in a RAM location.

As seen in the diagram, the three modules or the three physical partitions of the system are the Buffered Register, the RAM and the Add & Store Unit. The Add & Store unit can be further functionally partitioned into a Control Unit and a Data unit as shown in Figure 11b. Also, the Register can be partitioned into a process which reads the input (REG_IN) and a process that outputs the data stored within it onto the DATA bus (REG_OUT). For convenience of model development, the Add & Store unit is further partitioned into three sub-units and the function of each unit is described below.

Each of the units mentioned above is modeled as a process. The Process Model Graph is then developed for the Three Module System by interconnecting these processes with signals. The PMG shown in Figure 11c is the PMG for the Three Module System .

CU : The Control Unit process generates the control signals for the Register, the RAM and Data units of the ADD & STORE.

ADU_1, ADU_2 and ADU_3 : The partitions of the DATA unit are: ADU_1, ADU_2 and ADU_3. Each of them is triggered by a different signal and performs a different operation. ADU_1 reads the register data into an internal register and performs addition of the data from RAM and data stored in the internal register. ADU_2 copies the

DADDR address to the address bus MADDR. ADU_3 outputs the result of the addition onto the bus, to be written into the RAM location.

REG_IN, REG_OUT: Processes to handle the input and output operations of the Buffered register.

RAM : This process models the RAM.

Since DATA signal is driven by more than one process, it has to be declared as a type MVL_vect (Multi-Valued Logic Vector, that is vector consisting of values '0', '1' and 'Z') with a bus resolution function.

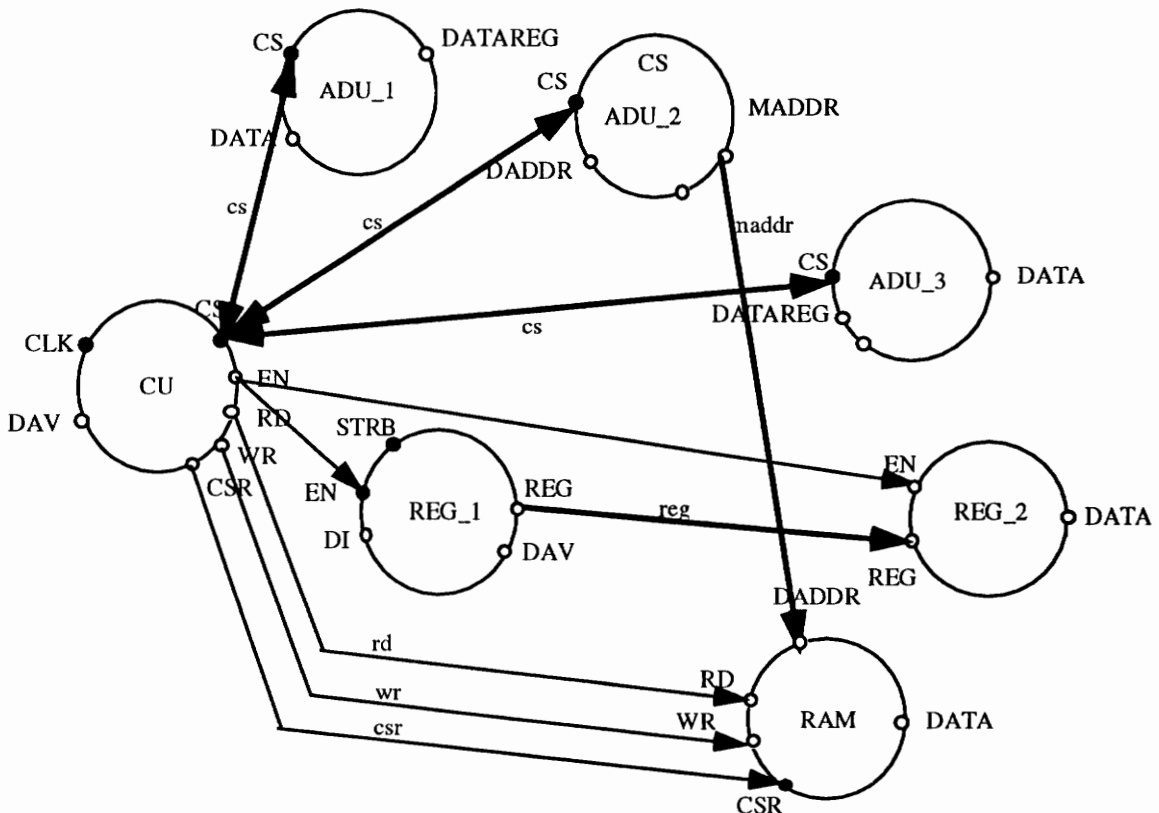


Figure 11c: Process Model Graph for the Three Module System

The CU: This is the Control Unit. It generates the control signals CS(0), CS(1), CS(2), CS(3) and CS(4), declared as an array CS(0 to 4) , that is CS is a bit_vector. CU is modeled as a circular shift right register beginning with initial value "10000", that is, CS(0) bit being '1' and the other bits being '0'. At every rising edge of clock pulse, if DAV (Data Available) is set it initiates a sequence of operations by assigning "10000" to CS. Otherwise the contents of CS undergo a circular shift right operation, thus advancing the high '1' to the next higher significant bit, that is CS takes values "01000", "00100" and so on. These signals trigger the ADD & STORE units ADU_1, ADU_2 and ADU_3 in a particular sequence. Also, the CS(0) signal is tied to EN signal of the Register, the CS(2) bit is tied to RD of RAM and CS(4) is tied to WR of RAM. Also, the chip select of the RAM goes high when either CS(2) or CS(4) is high. Thus, this Process has input ports CLK, DAV(Data Available) and CS.

A ModelSpeak description of CU can be written as:

"While CLK rises, if DAV is high then the value "10000" is applied to CS otherwise CS is [CS(4) & CS(1 to 3)]. Then, when CS changes CS(0) is applied to EN, CS(2) is applied to RD and CS(4) is applied to WR . Also, CSR is [CS(2) or CS(4)]. "

The VHDL generated is:

```

if (CLK'event and CLK= '1') then
    if (DAV = '1') then
        CS <= "10000";
    else CS <= CS(4) & CS(1 to 3);
    end if;
end if;
if (CS'event) then
    EN <= CS(0);
    RD <= CS(2);
    WR <= CS(4);
    CSR <= CS(2) or CS(4);
end if;

```

ADU_1, ADU_2 and ADU_3 : An (ADU) is triggered by CS(1), CS(2), CS(3) and CS(4). CS(1) loads data into an internal data register and CS(3) does the addition. CS(2) copies the memory address to MADDR and enables the CSR (Chip Select) of RAM and CS(4) outputs a result to the DATA bus.

A ModelSpeak description of ADU_1 can be written as:

"When CS(1) equals '1' the ADU reads the data from the DATA bus into the DATA_REG register otherwise if CS(3) equals '1' the ADU adds the value from the DATA bus to the contents of DATA_REG and stores it in the DATA_REG register."

The VHDL generated is:

```
if(CS(1) = '1') then
    DATA_REG <= DATA;
elsif (CS(3) = '1') then
    DATA_REG <= ADD(DATA, DATA_REG);
end if;
```

A ModelSpeak description of ADU_2:

"When CS(2) equals '1' or CS(5) equals '1' the ADU transmits the value of DADDR to MADDR."

```
if( CS(2) = '1' or CS(5) = '1') then
    MADDR <= DADDR;
end if;
```

A ModelSpeak description of ADU_3 can be written as:

"When CS(4) equals '1' the ADU outputs the data from DATA_REG onto the DATA bus otherwise it applies "ZZZZZZZZ" to the DATA bus."

The VHDL generated is:

```
if ( CS(4) = '1' ) then
    DATA <= DATAREG;
else DATA <= "ZZZZZZZZ";
end if;
```

REG_IN : This Process has input ports STRB, EN and DI, with EN and STRB in the sensitivity list. The output ports are REG and DAV. The EN signal is asserted by the Add and Store Unit, upon receipt of the DAV signal.

A ModelSpeak description of REG_IN can be written as:

"When STRB rises, the data DI is strobed into the REG register and the DAV signal made high and when EN goes high, DAV is deasserted."

The VHDL generated is:

```
if( not STRB'stable and STRB = '1') then
    REG <= DI;
    DAV <= '1'
end if;
if( EN = '1') then
    DAV <= '0';
end if;
```

REG_OUT: This is the output process which gates the register data onto the data bus.

A ModelSpeak description of REG_OUT can be written as:

"Assertion of EN gates the contents of REG register onto the DATA bus. Alternatively, "ZZZZZZZZ" is applied to the DATA bus."

The VHDL generated is :

```
if(EN='1' and EN'event) then
    DATA <= REG;
else DATA <= "ZZZZZZZZ";
end if;
```

RAM: When the chip select signal CSR is high, it transfers data into or out of the memory depending on the RD or WR signal.

A ModelSpeak description of RAM:

"While CSR remains high, if RD is high the data from variable MEM(INTVAL(MADDR)) is loaded onto the DATA bus otherwise if WR is high the data from the DATA bus is loaded into the variable MEM(INTVAL(MADDR)). If CSR equals '0', "ZZZZZZZZ" is applied to the DATA bus."

The VHDL generated is :

```
if( CSR = '1') then
    if (RD = '1') then
        DATA <= MEM(INTVAL(MADDR));
    elsif (WR = '1') then
        MEM(INTVAL(MADDR)) := DATA;
    end if;
end if;
if (CSR = '0') then
    DATA <= "ZZZZZZZZ";
end if;
```

6.3 The Multiplexer Example

Individual processes can be created and stored in the process primitive library, for use in the future model developments. For example the Multiplexer(MUX) can be created as shown in Figure 12. Here, IN0, IN1, IN2 and IN3 are MUX inputs SEL is the select input to the MUX and O is the single output line.

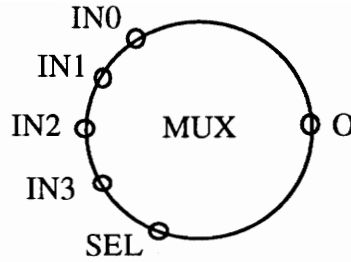


Figure 12: Process Symbol of a MUX

The ModelSpeak Description of MUX is:

"When SEL equals "00" , the MUX transfers input IN0 to the output O, otherwise when SEL equals "01" it transfers IN1 to the output O. When SEL equals "10" the MUX transfers IN2 to the output O otherwise when SEL equals "11" it transfers IN3 to the output O."

After creating the process symbol and giving the ModelSpeak description of the process which generates the VHDL code for the process, the primitive is created out of this process and stored in the library. The VHDL code generated for the process is:

```

if ( SEL= "00") then
    O <= IN0;
elsif (SEL = "01") then
    O <= IN1;
end if;

if ( SEL = "10" ) then
    O <= IN2;
elsif (SEL = "11") then
    O <= IN3;
end if;

```

6.4 Parallel I/O ports example (8212)

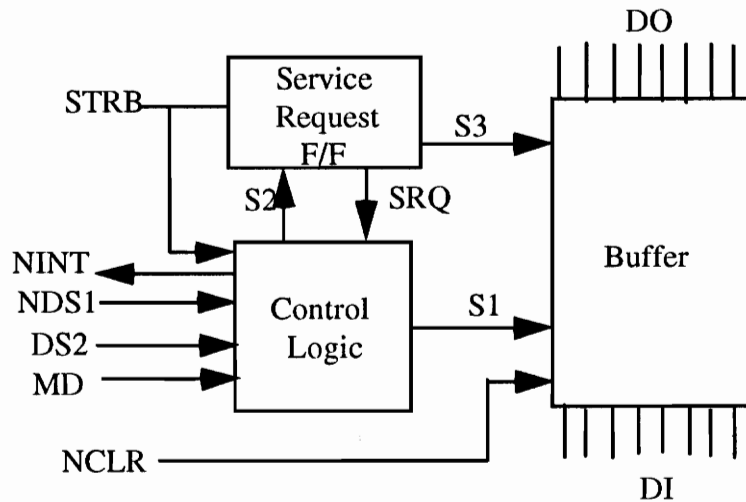


Figure 13a: Block diagram of Intel 8212 chip

The Block diagram of a buffered latch (Intel 8212 chip) is shown in Figure 13a. A VHDL model for this chip has been given in the textbook "[Chip Level Modeling with VHDL](#)". The Control logic has inputs MD(MODE), STRB and NDS1 and DS2 (Device Selection Logic). The combinational control logic generates outputs S1, S2 and S3 which control the operations of the Buffer and the Service Request flip flop.

Shown below in Figure 6 is the PMG of the above chip. In this PMG, the Control logic and the Service Request Flip Flop are modeled as processes. The Buffer has been functionally partitioned into two processes, the D_LATCH and the OUTPUT(output of the buffer).

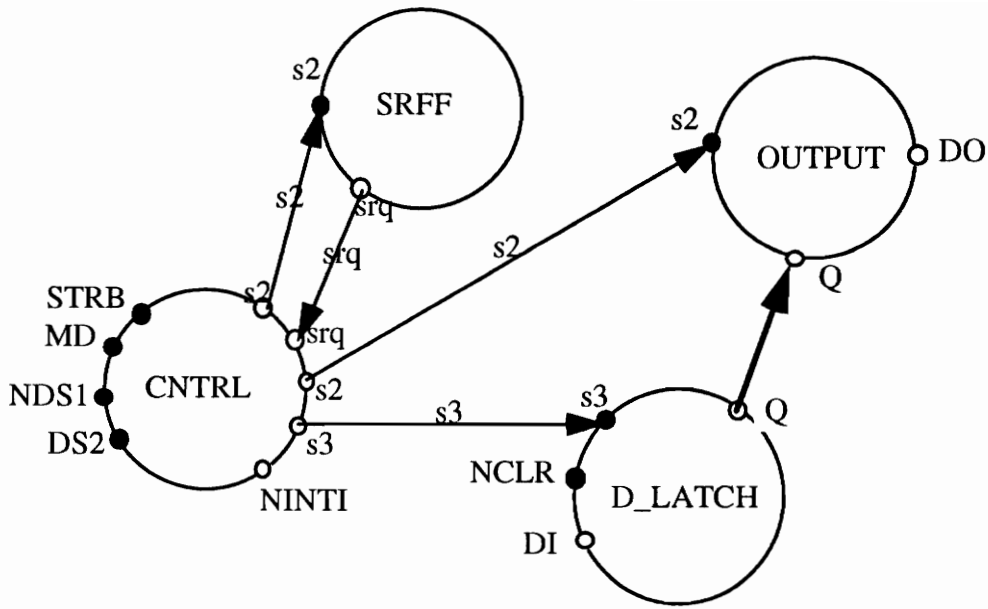


Figure 13b: PMG of parallel I/O port model

CNTRL : It is a combinational logic block. The output signals can be expressed in the form of boolean expressions in terms of the inputs.

The expressions must be enclosed within square brackets and a space must separate the bracket and the expression(ex. [---]).

In ModelSpeak these expressions can be written as:

*"S0 is [not NDS1 and DS2 after GDEL], S1 is [(S0 and MD) or (STRB and not MD) after 2*GDEL], S2 is [S0 nor not NCLR after GDEL] and S3 is [S0 or MD after GDEL]. Also, NINTI is [not SRQ nor S0 after GDEL].*

The signal assignments statements in VHDL are generated as shown:

```

S0 <= not NDS1 and DS2 after GDEL ;
S1 <= (S0 and MD) or (STRB and not MD) after 2*GDEL;
S2 <= S0 nor not NCLR after GDEL;
S3 <= S0 or MD after GDEL;
NINTI <= not SRQ nor S0 after GDEL;

```

D_LATCH: The CLK input to the data latch is given by the S1 signal. NCLR is the clear signal, overriding the CLK. DI is the input and Q is the output.

ModelSpeak Description of D_LATCH can be written as :

"When NCLR goes low, "00000000" is applied to the output Q, otherwise when S1 rises DI is transmitted to the output Q."

```
if (NCLR = '0') then
    Q <= "00000000";
elsif ( S1'event and S1 = '1') then
    Q <= DI;
end if;
```

OUTPUT: This models the output section of the buffer. When either STRB or MD is high the Buffer is enabled and the latched data is gated to the output port.

The ModelSpeak description is written as:

"When S3 equals '1' the data Q is gated to the port DO otherwise "ZZZZZZZZ" is applied to DO."

The VHDL is :

```
if(S3 = '1') then
    DO <= Q;
else DO <= "ZZZZZZZZ";
end if;
```

SRFF(Service Request Flip Flop) : SRQ is set when NCLR goes low or the device is selected, this logic is implemented as the S2 signal. SRQ is reset by the negative edge of STRB.

ModelSpeak Description of SRFF can be written as :

When S2 goes low SRQ is set, otherwise when STRB falls and S2 equals '1' SRQ is reset.

The VHDL generated is:

```
if( S2 = '0' ) then
    SRQ <= '1';
elsif( S2 = '1' and STRB'event and STRB = '0') then
    SRQ <= '0';
end if;
```

6.5 The UART Example

The UART model is taken from the textbook, "Chip-Level Modeling with VHDL" [10]. The PMG development for the UART begins with functional partition of the UART model. The resultant functional partitions that can be modeled as processes are : The Parallel_Input, Serial_Output, Serial_Input and Parallel_Output. Also, included are Control blocks, CNT_1 and CNT_2 that generate internal clocks ICLK and OCLK, respectively. The PMG developed for this model is shown below in Figure 14.

NINTO and O are signals driven by both PAR_IN and SER_OUT processes. To avoid conflict in values assigned to these signals, they are given different names in the two processes. These signals then have to be multiplexed. The final entity model for this example (shown in Appendix D) includes a process that multiplexes these signals. This process is not generated by the semantic parser program, but it is assumed that the behavioral model has been modified by the user to include this process, in order to have a complete model. Signal NINTI is also multiplexed for the same reason mentioned above.

Also, to correct type mismatches, certain conversion functions like TRIVEC_TO_BITVEC and BITVEC_TO_TRIVEC need to be included in the signal assignments. Since the semantic parser has no detailed information on types, it does not

handle type conversions. Therefore, even though the semantic parser does not generate these functions it has been included in the model shown in Appendix D, assuming that the user has taken care of them.

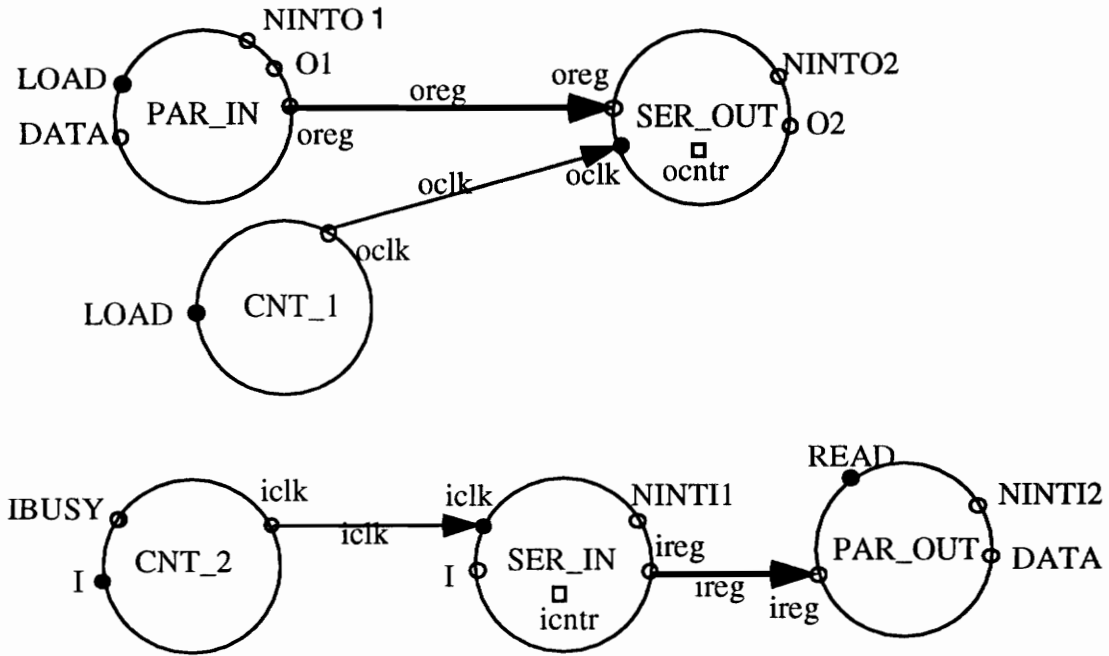


Figure 14 : PMG of the UART model

Par_In: The LOAD input triggers a parallel load of the DATA input and O is the serial output port which is applied a low when LOAD rises.

The ModelSpeak description can be written as:

"The rising edge of LOAD loads the input DATA into the output register oreg, applies a high to the interrupt signal NINTO1 and transmits a low to the output signal O1."

```

if(not LOAD'stable and LOAD='1') then
    oreg <= DATA;
    NINTO1 <= '1';
    O1 <= '0';
end if;

```

Cnt_1 : This process generates OCLK which controls the serial shift of the contents of OREG register to the serial output O.

The ModelSpeak description of this process could be written as :

"The rising edge of LOAD complements OCLK after 50ns. Then, when OCLK changes OCLK is complemented after 50 ns."

```
if(not LOAD'stable and LOAD='1') then
    oclk <= not(oclk) after 50ns;
end if;

if(not oclk'stable) then
    oclk <= not(oclk) after 50ns;
end if;
```

Ser_Out: This process serializes the parallel data. It does it for count, from 7 down to 0 and at the completion of this, puts a high on the serial port O and deasserts the interrupt signal NINTO. OCNTR is declared as a variable initialized to a value 7 when this process symbol is created.

The ModelSpeak Description would be:

"While OCLK changes , if OCNTR does not equal -1 then the value OREG(OCNTR) is transferred to the output O2 and variable OCNTR is decremented, otherwise the output O2 is made high and NINTO2 signal made low. Also, value 7 is loaded into variable ocntr."

Here, the VHDL generation is preceded by a query from the parser. A reply to this query then is followed by the VHDL code.

Type 'y' if ocntr is integer and 'n' if not;
l: y

```

        if(not oclk'stable) then
            if(ocntr /= -1) then
                O2 <= oreg(ocntr);
                ocntr := ocntr -1;
            else
                O2<= '1';
                NINTO2 <= '0';
                ocntr := 7;
            end if;
        end if;

```

Cnt_2: At the first negative transition of serial Input I, provided the IBUSY signal is low, reading of serial input data begins. This is controlled by the ICLK generated by this logic. Therefore, the ModelSpeak description can be written as :

"The negative edge of input I with IBUSY being low, applies a high to the IBUSY signal. The positive edge of IBUSY complements ICLK after 25ns. Then, when ICLK changes, ICLK is complemented after 50ns."

The VHDL generated for this process would be :

```

        if(not I'stable and I = '0' and IBUSY ='0') then
            IBUSY <= '1';
        end if;

        if(not IBUSY'stable and IBUSY ='1') then
            iclk <= not(iclck) after 25ns;
        end if;
        if(not iclk'stable) then
            iclk <= not(iclck) after 50ns;
        end if;

```

Ser_In: This process reads in the serial input data at the rise of each pulse of ICLK. At the completion of this read, the NINTI is reset and IBUSY made low.

The NL description in ModelSpeak can be written as :

"While ICLK changes, if ICNTR does not equal -1 then the input I is transmitted to IREG(ICNTR) and variable ICNTR is decremented, otherwise NINTI1 is reset and IBUSY is made low. Also, value 7 is loaded into variable icntr."

The resulting VHDL generated is :

Type 'y' if icntr is integer and 'n' if not;

l: y

```

        if(not iclk'stable) then
            if(icntr /= -1) then
                ireg(icntr) <= I;
                icntr := icntr - 1;
            else
                NINTI2 <= '0';
                IBUSY <= '0';
                icntr := 7;
            end if;
        end if;
    end if;

```

Par_out: When the processor sends the READ signal, the serial data stored in IREG register is transferred to the DATA output.

The ModelSpeak description can be written as :

"When READ is high the contents of IREG register is transferred to DATA output and NINTI2 is made high otherwise "ZZZZZZZZ" is applied to the DATA output."

The VHDL generated for this process is :

```

        if(READ = '1') then
            DATA <= ireg;
            NINTI2 <= '1';
        else DATA<= "ZZZZZZZZ";
        end if;
    end if;

```

6.6 The Traffic Light Controller Example

This example, taken from the textbook "VHDL: Hardware Description and Design" [16], illustrates the modeling of a finite state machine for a traffic controller, the traffic light being at the junction of a highway and a farmroad. The four states of the machine are : Highway_light_green, Highway_light_Yellow, Farmroad_light_Green and Farmroad_light_Yellow. Car_on_Farmroad is an input to the state machine, which is true when a car on the farmroad is sensed. There exists a timer that keeps track of time durations LTO(Long_Timeout) and STO(Short_Timeout). These are the on-time durations of Green light and Yellow light respectively. These also form inputs to the State Machine. The outputs of the state machine are Highway light and Farm Road light. The state machine for this controller is shown below in Figure 15a.

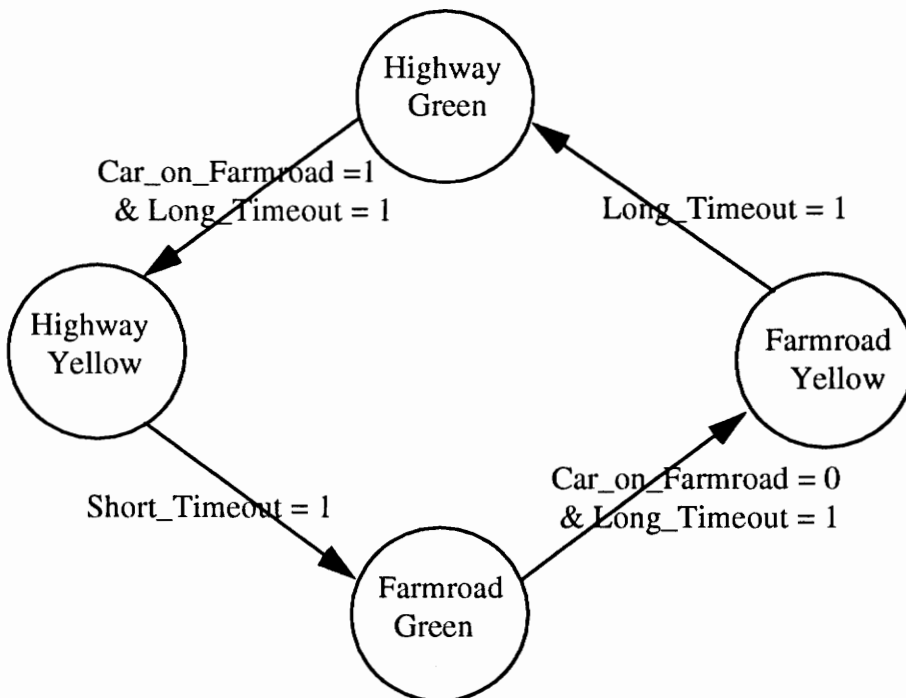


Figure 15a: State machine for the Traffic light Controller

The Process Model Graph representing the Finite State Machine is shown in Figure 15b. The Processes in the PMG are the Timer, the Finite State Machine(FSM) and the output processes for the highway light and farm road light respectively.

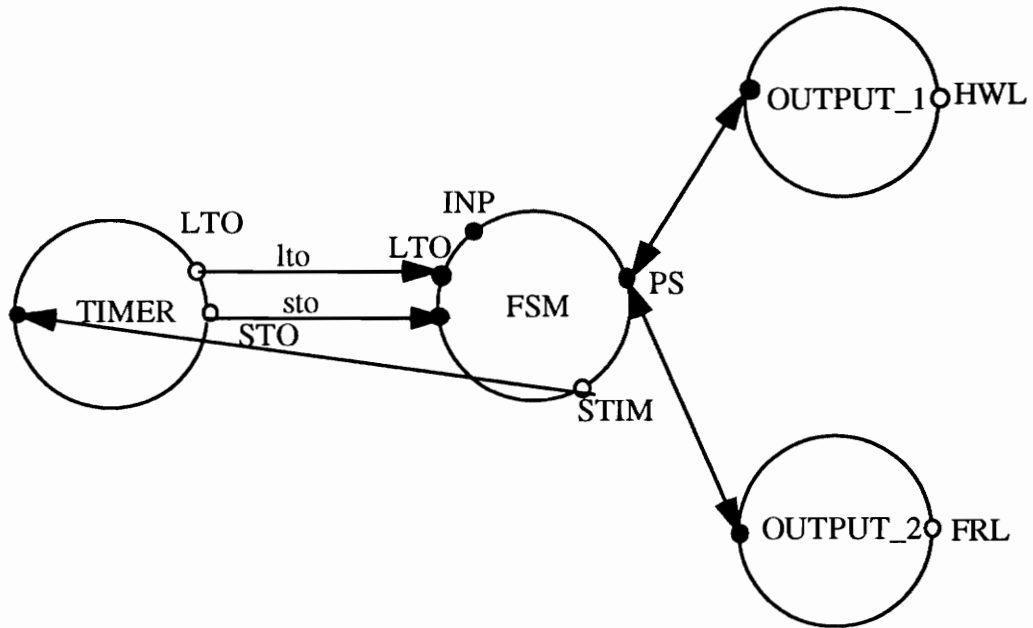


Figure 15b : PMG representing the state machine of a Traffic Light Controller

TIMER: STIM is the input, reset by the FSM to start the Timers LTO and STO. The LTO and STO are set after a period of 'long_time' and 'short_time', respectively.

The ModelSpeak description can be written as :

"LTO is ['0', '1' after long_time]. Also, STO is ['0', '1' after short_time]. "

The VHDL generated is :

```

LTO <= '0', '1' after long_time;
STO <= '0','1' after short_time;

```

FSM: INP is the Car_on_Farmroad Input. PS represents the Present_state of the Finite State Machine, it is declared as an integer. PS = 0 is the Highway_Light_Green state, PS = 1 is the Highway_Light_Yellow state , PS = 2 is the FarmRoad_Light_Green state and PS = 3 is the FarmRoad_Light_Yellow state.

The ModelSpeak description is :

" When PS is 0 with INP being '1' and LTO being '1' , PS changes to 2. When PS is 1 and STO is '1' PS changes to 2. When PS is 2 with INP being '0' and LTO being '1' PS changes to 3. and when PS is 3 and STO '1' PS changes to 0. Also,When PS changes it complements STIM.

The VHDL generated is :

```
if ( PS = 0 and INP = '1' and LTO = '1') then
    PS <= 1;
end if;
if( PS = 1 and STO = '1') then
    PS<= 2;
end if;
if( PS = 2 and INP = '0' and LTO = '1') then
    PS <= 3;
end if;
if( PS = 3 and STO='1') then
    PS <= 4;
end if;
if (PS'event) then
    STIM <= not STIM;
end if;
```

OUTPUT_1: This process gives the highway light output (HWL), the value depending on the Present State(PS). HWL is declared as an integer. Each number indicates a different color of the light. A 0 indicates a green, 1 indicates a yellow and 2 indicates red.

The ModelSpeak description would be :

" When PS is 0 , HWL is made 0 . Alternatively when PS is 1 HWL is made 1. When PS is 2 or PS is 3 then HWL is made 2."

The VHDL generated would be:

```
if(PS = 0) then
    HWL <= 0;
elsif(PS = 1) then
    HWL <= 1;
end if;
if(PS=2 or PS = 3) then
    HWL <= 2 ;
end if;
```

OUTPUT_2: This process gives the FarmRoad light output(FRL). Similar to HWL, FRL is declared as an integer and the numbers indicate the colors.

The ModelSpeak description can be written as :

" When PS is 2 FRL is made 0. Alternatively if PS is 3 FRL is made 1 and when PS is 0 or PS is 1 , FRL is made 2."

The VHDL generated is :

```
if(PS = 2 ) then
    FRL <= 0;
elsif(PS = 3) then
    FRL <= 1;
end if;
if(PS = 0 or PS = 1) then
    FRL <= 2;
end if;
```

Chapter 7

Limitations and Future Work

The work done in developing this interface was concentrated mainly in defining the ModelSpeak language, its constructs and vocabulary such that semantic rules could be developed to easily analyze the sentences and map them into VHDL statements. Not all types of VHDL statements can be generated from ModelSpeak. For example, the `case_statements`, `nested_loops`, `if_statements` with more than one level of 'elsif' condition are not generated since the English descriptions that imply such statements are awkward and unnatural.

Although ModelSpeak is a restricted form of English, it is developed to be close to natural language, so that users will find it easier to write descriptions in ModelSpeak than in the formal VHDL language.

The NL interface described in this thesis generates VHDL code for processes, which are then used by the Modeler's Assistant to generate behavioral models. But, the interface does not interact with the Modeler's Assistant's internal database where information on names and types of ports, signals and variables is stored. Modification of Modeler's Assistant is beyond the scope of this project. This is because the Modeler's assistant database is in the form of C data-structures and the interface is written in Prolog. Due to

this limitation, the user has to explicitly enter the names of the ports, signals and variables in the description. Information on the type of signals or variables must also be obtained by interacting with the user. Another limitation is that the interface program does not maintain a history of information typed in by the user. So, even when the same information about the name and types of elements is required while building another process in the PMG, they have to be re-typed by the user.

The information on ports, signals and variables are entered while constructing the PMG using the graphical interface. This is stored in the internal database. To eliminate the limitations described above the interface program can be built to interact with this internal database of the Modeler's Assistant. The information is stored in C data-structures and the Prolog program can be modified to read this data-structure and store the information within its database using the Prolog data-structures. This information can then be looked up easily each time it is required. This way the user is not required to type in redundant details. This will also help reduce the time spent on developing the 'processes' in the PMG.

References

- [1] IEEE Standard VHDL Language Reference Manual, 1988.
- [2] B. Singh, "Parametrized CAD tool for VHDL Model Development with X windows," Master's Thesis, Virginia Polytechnic Institute and State University, 1990.
- [3] J. R. Armstrong and D. Burnette, "Automated Assists to the Behavioral Modeling Process," Proceedings of the First International Workshop on Rapid System Prototyping, Research Triangle Park, NC, June 1990.
- [4] W.R. Cyre, "The Design of Restricted Sublanguage," Proc. Theoretical Approaches to Natural Language Understanding Workshop, Halifax, Nova Scotia, May 28 -30, 1985.
- [5] Richard R. Burton, "Semantic Grammar: An Engineering Technique for constructing Natural Language Systems", Dec 1976.
- [6] R. Greenwood and W.R. Cyre, " Conceptual Modeling of Digital Systems from informal descriptions," MASCOTS 1993 International Workshop on Modeling, Analysis and simulation of computer and Telecommunication Systems, January 17-20, 1993, San Diego, CA.
- [7] J. F. Sowa, Conceptual Structures: Information Processing in Mind and Machine, Addison_Wesley, Reading, MA, 1984.

- [8] J.J Granacki and A. Parker, "PHRAN-SPAN : A Natural Language Interface for System Specifications, "Proc. 24th Design Automation Conference, Miami Beach, FL, June 1987.
- [9] Morris Mano, "Computer System Architecture", Prentice Hall Inc. 1982.
- [10] J.R. Armstrong, "Chip Level Modeling with VHDL," Prentice Hall Inc. 1988.
- [11] N.D. Dutt, J.H.Cho, "A User Interface for VHDL Behavioral Modeling", Proc. of 10th Int. Symposium on CHDL, 1991.
- [12] G. Hendrix, D. Saciedoti, D. Saglowicz and J. Solcum, Developing a Natural Language Interface to Complex Data", ACM Transactions of Database systems, Vol 3, 1978.
- [13] W.F. Clocksin and C.S. Mellish, "Programming in Prolog", 3rd edition, Springer-Verlag, 1987.
- [14] T. Winograd, "Language as a Cognitive Process", Vol. 1 : Syntax, Addison - Wesley, Reading MA, 1983.
- [15] Peter Linz, "An Introduction to formal Languages and Automata" D.C Heath & CO, Lexington, MA, 1990
- [16] R. Lipsett, C. Schaefer, C. Ussery, "VHDL: Hardware Description and Design" Kluwer Academic Publisher, Boston, 1989
- [17] J.R. Armstrong,F.G. Gray, "Structured Logic Design with VHDL"

Appendix A

User's guide for the Semantic Parser

As mentioned earlier the NL interface is written in Quintus Prolog on a Sun Sparcstation platform. There are two versions of the program, a demo version and an another version that is interfaced to the Modeler's Assistant.

Demonstration

In the demo version, the executable file "Demo" reads the description that is already stored in the file "demo_des". The description consists of a paragraph of sentences implying all types of VHDL statements handled by the parser. To run it at the Unix prompt, type in:

%Demo filename

Where *filename* is the output file to which the VHDL output is written. This loads Prolog and hence the Prolog prompt `!:`, appears. At this prompt type in :

!:description.

The description stored in "demo_des" is then parsed. Since the last line of the description implies a `while_loop` the user-interaction feature is invoked to resolve the ambiguity of the scope of the loop. Depending on the user response, appropriate VHDL is generated and displayed for user verification. Also, the output is written to the file specified by the user.

To quit the demo session, type:

!:*halt.*

Using the ModelSpeak Interpreter

In the version that is interfaced to the Modeler's Assistant, the program is invoked from within the Modeler's Assistant. After the process symbol is created, click on the main menu 'English' to enter the ModelSpeak description of the process. The Modeler's Assistant then automatically invokes the executable Prolog file 'Process'. It also passes a filename to the Prolog program as an argument, to which the output can be written. When 'Process' is executed a window pops - up, loading Prolog into it and the Prolog prompt ***!:***, appears. At this type in:

!:*description.*

At this a message appears:

Enter the description of the VHDL process.

Then, type in the ModelSpeak description in a paragraph and terminate with a period and hit return. If a sentence in the description implies a loop or there is ambiguity regarding the type of signal or variable, then the user-interaction is invoked. For each query, respond with a 'y' or 'n' followed by a period and a carriage return. When the description is parsed the VHDL code for the process is generated and also written to the file specified by the Modeler's Assistant. This file is stored in the internal database of Modeler's Assistant along with the process symbol associated with it.

To quit from Prolog and return to Modeler's Assistant type:

!halt.

At this, the Prolog program is terminated and the window disappears thus returning to the Modeler's Assistant environment. Each process of the PMG constructed, is so developed and all the information is integrated by the Modeler's Assistant to generate the VHDL behavioral model.

Appendix B

Semantic categories, Key words and symbols of the ModelSpeak Language

action concepts

asserts
asserted
applies
applied
loads
loaded
activates
activated
enables
enabled
disables
disabled
clears
cleared
resets
reset
sets
set
pulses
strobes
strobed
deasserts
deasserted
transfers
transferred
gates
gated
transmits
transmitted
complements
complemented
changes
increments
incremented

latches
latched
store
stored
adds
puts

event concepts

assertion
deassertion
rises
falls
rising edge
falling edge
positive edge
negative edge

relation concepts

is
is not
equals
does not equal

value concepts

high
low
'1'
'0'
set
reset

type concepts

input
output

class concepts

signal
port
register
line
bus

connection concepts

then
also
after this
after that

conjunction concepts

and
or

determiners

the
a

prepositions

to
from
of
in
on
into
onto

Keywords

if
when
otherwise

Keywords

variable

repeated

continuously

times

until

[] (for boolean expressions)

while

This

waits

process

reaches

nouns

data

contents

delimiters

,

.

Symbols

(

)

*

"

'

Numbers

0 - 9

Appendix C

Examples of ModelSpeak Sentences

Examples for signal_assignment_statements:

- (a) The dma asserts the INT signal.
- (b) The dma activates the BR line.
- (c) The device puts a high on WR line.
- (d) The register applies a high to the IN port.
- (e) The device transfers the data from the ADDR bus to the MEM register.
- (f) The dma puts the value of ADDREG register on the ADDR bus.
- (g) The buffer transmits the REG value to the output port DO.
- (h) The dma increments the ADDR register.
- (i) A high is applied to the IBUSY signal.
- (j) IBUSY is made high.
- (k) The input DATA is strobed into the OREG register.
- (l) The contents of OREG are transferred to the DATA_BUS.
- (m) The value of OCLK is complemented.
- (n) S1 is [not MD and STRB].
- (o) S2 is [CS(1) & CS(2)].

Examples for simple if_statements :

- (a) When STRB equals '1' the device applies the contents of DATA register on PORTC and then pulses STRA.
- (b) If BG is set, the dma transfers the data from the ADDR bus to the MEM register.

- (c) Assertion of STRB latches the data on PORTC into the PORTCL register.
- (d) The positive edge on STRA deasserts the STRB line.
- (e) When STRB rises the device disables the RC line.
- (f) When SET equals '1' and RESET equals '0', the device puts a high on the output port Q.
- (g) When PS is 1 and INP is '1' PS changes to 2.
- (h) When PS changes it complements the ICLK.
- (i) When OCNTR does not equal -1, I is transferred to OREG(OCNTR).

Examples for if_statements with 'elsif' and 'else' clauses:

- (a) If J equals '1' and K equals '0' the output Q is set, alternatively if J equals '0' and K equals '1' output Q is reset.
- (b) If SIG equals '1' the microprocessor enables the RD line otherwise it enables the WR line.

Example for nested_if_statement :

- (a) While CS is low, if RD is high the microprocessor transfers the data from the DATA bus to the MBR register otherwise if WR is high the microprocessor transfers the contents of the MBR register to the DATA bus.

Examples for for_loop & while_loop statements:

- (a) This is repeated continuously 3 times.
- (b) This is repeated continuously until C reaches 0.

Appendix D.

VHDL Model for the DMA Interface Module

```
use WORK.VHDLCAD.all, WORK.USER_TYPES.all;
-- *****

entity DMA_BEH is
  port (WCR : inout BIT_VECTOR(3 downto 0);
        DAR: inout BIT_VECTOR(3 downto 0);
        INT: out BIT;
        ADDR: out BIT_VECTOR(3 downto 0);
        DMAA: out BIT;
        WR: inout BIT;
        RD: out BIT;
        CON: in BIT;
        BG: in BIT;
        DATA: in BIT_VECTOR(3 downto 0);
        RS: in BIT;
        CS: in BIT;
        BR: out BIT;
        DMAREQ: in BIT);
end DMA_BEH;
-- *****

architecture BEHAVIORAL of DMA_BEH is

begin

  -----
  -- Process Name: PROC_2_3
  -----

  PROC_3_4: process
  begin
    wait on BG, CSR;
    if (BG = '1') then
      if (CON = '1') then
        RD <= '1';
      else
        WR <= '1';
      end if;
    end if;
    if (BG = '1') then
      DMAA <= '1';
      while not WCR = "0000" loop
        addr <= DAR;
```

```

        wait for 5 ns;
        DAR <= incr(DAR);
        WCR <= decr(WCR);
    end loop;
    int <= '1';
end if;

if( CS= '1' and WR = '1') then
    if( RS = '1') then
        DAR <= DATA;
    else
        WCR <= DATA;
    end if;
end if;

end process PROC_2_3;

-----
-- Process Name: PROC_1
-----

PROC_1_25: process (DMAREQ)
begin
    if (DMAREQ = '1') then
        BR <= '1';
    end if;

end process PROC_1_25;

end BEHAVIORAL;

```

VHDL Model for the Three Module System

```

use WORK.VHDLCAD.all, WORK.USER_TYPES.all;
-- *****
entity threemod is
    port (data: inout TSL_VECTOR(7 downto 0);
          dav: inout BIT;
          di: in BIT_VECTOR(7 downto 0);
          strb: in BIT;
          data_reg: inout BIT_VECTOR(7 downto 0);
          daddr: in BIT_VECTOR(4 downto 0);
          data: in BIT_VECTOR(7 downto 0);
          clk: in BIT);
end threemod;

```

```
-- *****
```

architecture BEHAVIORAL of threemod is

```
signal maddr: BIT_VECTOR(4 downto 0);
signal csr: BIT;
signal wr: BIT;
signal rd: BIT;
signal reg: BIT_VECTOR(7 downto 0);
signal en: BIT;
signal cs: BIT_VECTOR(0 to 4);
begin

-- -----
-- Process Name: ram
-- -----

ram_4: process (csr)
    variable mem: MEMORY:= (others =>(others =>'0'));;
begin

    if(csr = '1') then
        if(rd = '1') then
            data <= mem(intval(maddr));
        elsif (wr = '1') then
            mem(intval(maddr)) := data;
        end if;
    end if;

    if(csr = '0') then
        data <= "ZZZZZZZZ";
    end if;

end process ram_4;

-- -----
-- Process Name: reg_2
-- -----

reg_2_12: process (en)
begin
    if(en = '1' and en'event) then
        data <= BITVEC_TO_TRIVEC(reg);
    else data <= "ZZZZZZZZ";
    end if;

end process reg_2_12;
```

```
-----  
-- Process Name: reg_1  
-----
```

```
reg_1_17: process (en,strb)  
begin  
  if(not STRB'stable and STRB = '1') then  
    reg<= TRIVEC_TO_BITVEC(di);  
    dav <= '1';  
  end if;  
  if(en = '1') then  
    dav <= '0';  
  end if;
```

```
end process reg_1_17;
```

```
-----  
-- Process Name: adu_3  
-----
```

```
adu_3_24: process (cs)  
begin  
  if(cs(4) = '1') then  
    data <= BITVEC_TO_TRIVEC(data_reg);  
  else data<= "ZZZZZZZZ";  
  end if;
```

```
end process adu_3_24;
```

```
-----  
-- Process Name: adu_2  
-----
```

```
adu_2_29: process (cs)  
begin  
  if(cs(2) = '1' or cs(4) = '1') then  
    maddr <= daddr;  
  end if;
```

```
end process adu_2_29;
```

```

-----
-- Process Name: adu_1
-----

adu_1_34: process (cs)
begin
    if (cs(1) = '1') then
        data_reg <= TRIVEC_TO_BITVEC(data);
    elsif (cs(3) = '1') then
        data_reg <= add(TRIVEC_TO_BITVEC(data),data_reg);
    end if;

end process adu_1_34;

-----
-- Process Name: cu
-----

cu_39: process (cs,clk)
begin
    if(clk'event and clk = '1') then
        if (dav = '1') then
            cs <= "10000";
        else cs <= cs(4) & cs(0 to 3);
        end if;
    end if;

    if(cs'event ) then
        en<= cs(0);
        rd <= cs(2);
        wr<= cs(4);
        csr <= cs(2) or cs(4);
    end if;
end process cu_39;

end BEHAVIORAL;

```

VHDL Model for the 8212 chip (Parallel Input/Output Ports)

```

use WORK.VHDLCAD.all, WORK.USER_TYPES.all;
-- *****
entity parin is
    generic (gdel: TIME
    );

```

```

port (do: out MVL_VECTOR(0 to 7);
      di: in MVL_VECTOR(0 to 7);
      nclr: in BIT;
      strb: inout BIT;
      s0: inout BIT;
      ninti: out BIT;
      ds2: in BIT;
      nds1: in BIT;
      md: in BIT);
end parin;
-- *****

```

architecture BEHAVIORAL of parin is

```

signal q: MVL_VECTOR(0 to 7);
signal s3: BIT;
signal s1: BIT;
signal srq: BIT;
signal s2: BIT;
begin

```

```

-----
-- Process Name: outpt
-----

```

```

outpt_4: process (s3)
begin
  if(s3 = '1') then
    do <= q;
  else do <= "ZZZZZZZZ";
  end if;

```

```

end process outpt_4;

```

```

-----
-- Process Name: d_latch
-----

```

```

d_latch_9: process (s1)
begin
  if(nclr = '0') then
    q <= "00000000";
  elsif(s1'event and s1 = '1') then
    q <= di;
  end if;

  end process d_latch_9;

```

```

-----
-- Process Name: srff
-----

```

```

srff_15: process (s2)
begin

    if(s2 = '0') then
        srq <= '1';
    elsif(s2 = '1'and strb'event and strb = '0') then
        srq <= '0';
    end if;


```

```

end process srff_15;

```

```

-----
-- Process Name: cntrl
-----

```

```

cntrl_20: process (ds2,nds1,md,strb)
begin
    s0 <= not nds1 and ds2 after gdel;
    s1 <= (s0 and md) or (strb and not md) after 2*gdel;
    s2 <= s0 nor not nclr after gdel;
    s3 <= s0 or md after gdel;
    ninti <= not srq nor s0 after gdel;


```

```

end process cntrl_20;

```

```

end BEHAVIORAL;

```

VHDL Model for the UART Module

```

use WORK.VHDLCAD.all, WORK.USER_TYPES.all;
-- *****
entity uart is
port (NINTI: out BIT;
      I: in BIT;
      IBUSY: inout BIT;
      LOAD: in BIT;
      O: out BIT;
      NINTO: out BIT;
      DATA: inout BIT_VECTOR(0 to 7);
      READ: in BIT);
end uart;

```

```
-- *****
```

architecture BEHAVIORAL of uart is

```
signal ireg: BIT_VECTOR(0 to 7);
signal iclk: BIT := '0';
signal oclk: BIT := '0';
signal oreg: BIT_VECTOR(0 to 7);
signal NINTI1,NINTI2,NINTO1,NINTO2,O1,O2: BIT;
```

```
begin
```

```
-----
-- Process Name: Ser_in
-----
```

```
Ser_in_4: process (iclk)
  variable icntr: INTEGER := 7;
begin
  if( not iclk'stable) then
    if(icntr /= -1) then
      ireg(icntr) <= I;
      icntr := icntr -1;
    else
      NINTI1 <= '0';
      IBUSY <= '0';
      icntr := 7;
    end if;
  end if;
end if;
```

```
end process Ser_in_4;
```

```
-----
-- Process Name: cnt_2
-----
```

```
cnt_2_12: process (I)
begin
  if(not I'stable and I = '0' and IBUSY ='0') then
    ibusy <= '1';
  end if;

  if(not IBUSY'stable and IBUSY ='1') then
    iclk <= not(iclk) after 25ns;
  end if;
  if(not iclk'stable) then
    iclk <= not(iclk) after 50ns;
  end if;
end process cnt_2_12;
```

```
-----  
-- Process Name: cnt_1  
-----
```

```
cnt_1_17: process (LOAD)  
begin  
    if(not LOAD'stable and LOAD = '1') then  
        oclk <= not (oclk) after 50ns;  
    end if;  
  
    if(not oclk'stable) then  
        oclk <= not(oclk) after 50ns;  
    end if;  
  
end process cnt_1_17;
```

```
-----  
-- Process Name: ser_out  
-----
```

```
ser_out_21: process (oclk)  
    variable ocnt: INTEGER := 7;  
begin  
    if(not oclk'stable) then  
        if (ocnt /= -1) then  
            O2<=oreg(ocnt);  
            ocnt := ocnt -1;  
        else  
            O2<= '1';  
            NINTO2 <= '0';  
            ocnt :=7;  
        end if;  
    end if;  
end process ser_out_21;
```

```
-----  
-- Process Name: Par_in  
-----
```

```
Par_in_29: process (LOAD)  
begin  
    if(not LOAD'stable and LOAD= '1') then  
        oreg <= MVLVEC_TO_BITVEC(DATA);  
        NINTO1 <= '1';  
        O1<= '0';  
    end if;  
end process Par_in_29;
```

```

-----
-- Process Name: par_out
-----

par_out_36: process (READ)
begin
    if(READ = '1') then
        DATA <= BITVEC_TO_MVLVEC(ireg);
        NINTI2 <= '1';
    else data <= "ZZZZZZZZ";
    end if;

end process par_out_36;

-----
-- Process Name: pro_mux
-----

pro_mux_44: process(NINTO1,NINTO2,NINTI1,NINTI2,O1,O2)
begin
    NINTO<= NINTO1 when not NINTO1'quiet else
        NINTO2 when not NINTO2'quiet;

    NINTI <= NINTI1 when not NINTI1'quiet else
        NINTI2 when not NINTI1'quiet;

    O <= O1 when not O1'quiet else
        O2 when not O2'quiet;

end process pro_mux_44;

end BEHAVIORAL;

```

VHDL Model for the State Machine of a Traffic Light Controller

```

use WORK.VHDLCAD.all, WORK.USER_TYPES.all;
-- *****
entity traffic is
    generic (short_time: TIME;
        long_time: TIME
    );
    port (frl: out INTEGER;
        hwl: out INTEGER;
        inp: in BIT);
end traffic;
-- *****

```

architecture BEHAVIORAL of traffic is

```
signal ps: INTEGER;  
signal stim: BIT;  
signal sto: BIT;  
signal lto: BIT;
```

```
begin
```

```
-----  
-- Process Name: outptu_2  
-----
```

```
outptu_2_4: process (lto,sto,ps,inp)
```

```
begin
```

```
  if(ps = 2) then  
    frl <= 0;  
  elsif (ps = 3) then  
    frl <= 1;  
  end if;
```

```
  if(ps = 0 or ps = 1 ) then  
    frl <= 2;  
  end if;
```

```
end process outptu_2_4;
```

```
-----  
-- Process Name: output_1  
-----
```

```
output_1_8: process (ps)
```

```
begin
```

```
  if(ps = 0) then  
    hwl <= 0;  
  elsif ps = 1 then  
    hwl <= 1;  
  end if;  
  if(ps = 2 or ps = 3) then  
    hwl <= 2;  
  end if;
```

```
end process output_1_8;
```

```
-----  
-- Process Name: fsm
```

```

-----
fsm_12: process (ps)
begin
  if(ps = 0 and inp = '1' and lto = '1') then
    ps <= 1;
  end if;
  if(ps=1 and sto = '1') then
    ps <= 2;
  end if;
  if(ps = 2 and inp = '0' and lto = '1') then
    ps <= 3;
  end if;
  if(ps = 3 and sto = '1') then
    ps <= 4;
  end if;

  if(ps'event) then
    stim <= not stim;
  end if;

end process fsm_12;

```

```

-----
-- Process Name: timer
-----

```

```

timer_19: process (stim)
begin
  lto <= '0','1' after long_time;
  sto <= '0', '1' after short_time;

end process timer_19;

```

```

end BEHAVIORAL;

```

Appendix E

Prolog program implementing the semantic parser

```
/* This program takes care of sentences of the form
1. When BR equals 1, controller enables the INT signal otherwise DMA disables the INT signal.
   if -- else -- endif;

2.While CS remains high if RD equals 1 DMA transfers the ADDREG contents onto the DATA bus
alternatively if WR equals 1 DMA transfers DATA bus contents to the ADDREG register. ....
   if (if --elsif--endif) endif; .....
Here the word "while" suggests a nested if.

3. If A(I) equals 1 the variable NUM is incremented. This is repeated continuously 5 times.
   for I in 0 to 4 loop (.....) end loop;
Also rules include those for a while_loop

N.B:- The signal assignment statements can be part of the
"if_statements". They can precede or follow the "if_statements".
*/

% read_in.pl has the code that reads English sentences describing
% the process and converts it into the list data structure
% (the words being the atoms of the list).

:- consult('read_in.pl').

/* Typing 'description' fires up the parser. Rule 'read_sp' reads the input description, invokes a spell
checker routine(UNIX routine) on the input and invokes the rule 'sem_par' to initiate the parsing of the
input.RES holds the VHDL generated by the parser and PP(RES) pretty prints the output.*/

description :-unix(argv([H|T])),nl,nl,
             printstring("Enter the description of the VHDL process"),
             nl, read_sp(RES),nl,nl,tell(H),
             pp(RES),told,pp(RES).

read_sp(RES):- read_in(S),nl,nl,tell(spl_ch),pp(S),nl,told,
              unix(system('spell spl_ch')),nl,nl,
```

```

        printstring("If there are any spellings listed that are incorrect type 'y' and restart with
        'description', otherwise type 'n'"),
        read(X),!,sem_par(X,S,RES).

sem_par(y,S,[]):- fail.
sem_par(n,S,RES):- proc_stat(RES,X,S,[]).

printstring([]).
printstring([H|T]) :- put(H), printstring(T).

/*****

/* proc_stat is the top-most rule of the code that starts parsing. The following are the non - terminals in
the code The latter section contains the terminals*/

/* This grammar contains both rules for Isolated sentences and multi-sentences descriptions. Some of
the other rules included here, are for the user-interaction feature */

/* Multi-sentence description rules */

proc_stat(RES,P) --> seq_stats(RES,P).

seq_stats(RES,P) --> seq_stat(P,FLAG,FLAG1,A),
                    check(FLAG,FLAG1,P,A,R),
                    next(R,RES).

next(P,X)    --> period,
                conntn,
                seq_stat(Q,FLAG,FLAG1,A),
                add(P,[],Q,INT),
                check(FLAG,FLAG1,INT,A,R),
                next(R,X).

next(R,R)    --> period.

sig_assgns(RES,FLAG,FLAG1,A)--> (sig_assgn(P)|wait_stat(P)),
                                next_1(P,RES,FLAG,FLAG1,A).

next_1(P,X,FLAG,FLAG1,A) --> period,
                            sig_conntn,
                            ((sig_assgn(Q),join(P,Q,R,FLAG))|(wait_stat(Q),join(P,Q,R,FLAG))
                            |(loop_stat1(P,P1,P2,A,C),
                            !,test(P2,P1,A,C,R,FLAG1))),
                            next_1(R,X,FLAG,FLAG1,A).

```

```

next_1(R,R,FLAG,FLAG1,A)  --> period.

seq_stat(P,FLAG,FLAG1,A)  --> sig_assgns(P,FLAG,FLAG1,A).
seq_stat(P,FLAG,FLAG2,A)  --> if_stat_1(P,A,FLAG,FLAG2).
seq_stat(P,FLAG,FLAG1,A)  --> if_stat_2(P).

/*****
/* if_statements rules with 'elsif' and 'else' conditions*/

if_stat_1(IF,A,FLAG,FLAG2)  --> when(P4,P),
                                cond(P1),
                                and_or(P3,P2),
                                add(P1,P2,P3,P4),period,
                                sig_assgns(Y,FLAG,FLAG1,A),period,
                                els_if(C),period,
                                els(R,S),
                                add(P,C,Y,D),
                                add(D,R,S,INT),
                                check1(FLAG1,INT,FLAG2),!,
                                flag_bit(FLAG1,INT,A,IF,P,Y,FLAG2).

/* nested_if statement */

if_stat_2(NIF)  -->   while(P4,X,P),
                        cond(P1),
                        and_or(P3,P2),
                        add(P1,P2,P3,P4),period,
                        if_stat_1(X,A,FLAG,FLAG2),
                        end(R),
                        add(P,[],R,NIF).

cond(P1)  --> event(P2,P1)|
              (event(P2,P1),
              prep,
              signal(P2)).

cond(P1)  -->
              cond_var(P2,P1),
              cond_val(P2).

cond_var(P2,[P,P2])-->[P].
cond_val([P,P1])-->
              rel(P),
              value(P1).

els_if(Q)  --> when(P4,P),
                cond(P1),
                and_or(P3,P2),
                add(P1,P2,P3,P4),
                sig_assgns(RES,FLAG1,FLAG,A),

```

```

        add(P,[],RES,Q).

els_if([]) -->[].

els([R,endif,;],[else]) --> (otherwise | alternatively ),
                             sig_assgns(R,FLAG1,FLAG,A).
els([],endif;]) -->[].

/*****
/* wait_statements */

wait_stat([[ wait,for,P,ns,;]])-->([waits,for,P,ns])|
                                   (det,id(Q),[waits,for,P,ns]).
wait_stat([[ wait,on,P,until,P,=,P1,;]]) -->
                                   ([waits,on,P,until,BG,equals,P1])|
                                   (det,id(Q),[waits,on,P,until,BG,equals,P1]).

/*****
/* loop_statements */

loop_stat1(P,P1,P2,A,C) --> repeat,
                           itrtn_schme(A),reverse(P,INT),message1,
                           pr(INT,X1,P1,X2,P2),end_loop(C).

loop_stat(P,INT) -->      repeat,
                           itrtn_schme(A),
                           end_loop(C), add(A,C,INT,P).

end([endif;]) -->[].
end_loop([end,loop;]) -->[].

/*****
/*      The following rules are to implement the
        User Interaction      */

test([],P1,A,C,P1,1,S,S).
test([H|T],P1,A,C,R,0,S,S):- append([H|T],A,P),

```

```

        append(P,P1,Q),append(Q,C,R).

join(P,Q,R,0,S,S):- append(P,Q,R).

message1(S,S):-nl,
printstring(" Which of the following statements
are to be repeated.
Answer with 'y' or a 'n'."),
nl.

message2(1,P,X,S,S):- nl,printstring("Type 'y'if"),tab(1),
        write(P),tab(1),
printstring("is integer,if not type 'n'"),read(X).

message2(2,P,X,S,S):- nl,printstring("Type 'y'if"),tab(1),
        write(P),tab(1),
printstring("is integer,if not type 'n'"),read(X).

message2(3,P,n,S,S).

check(0,0,INT,A,INT)-->[].
check(0,1,[for|T],A,[for|T])-->[].
check(0,1,[while|T],A,[while|T]) -->[].
check(0,1,INT,A,R) --> end_loop(C),add(A,C,INT,R).

check1(0,INT,0)-->[].
check1(1,INT,FLAG2)--> pp(INT,X),!,match(X,FLAG2).

match(y,1,S,S).

match(n,0,S,S).

flag_bit(0,INT,A,INT,P,Y,0)-->[].

flag_bit(1,INT,A,IF,P,Y,1) -->end_loop(C),add(A,C,INT,IF).
flag_bit(1,INT,A,IF,P,Y,0) --> difference(INT,P,INT1),
        difference(INT1,Y,INT2),
        end_loop(C),add(A,C,Y,Q),add(P,INT2,Q,IF).

/*****

/* For sig_assign that accept boolean expressions */

```

```

sig_assgn([[P1,<=, P2,T]]) -->    lhs(P1),
                                   brace,
                                   rhs(P2),
                                   timeout(T).

lhs(P1)          --> [P1,is].
brace            --> ['['].
rhs([])          --> [']].
rhs([H|T])       --> [H],rhs(T).

/* For sig_assgn_statements expressed in active voice */
/* type: The controller asserts the int signal.
And, the controller increments the DAR */
sig_assgn([[P1,P2,T]]) -->    (action1(P2),
                                det,
                                target(P1),timeout(T)) |

                                (action20(I),det,
                                 ((target1(P1),
                                  difference(P1,[:=],[P])) |
                                  (target(P1),
                                   difference(P1,[<=],[P]))),
                                  message2(I,P,X),
                                  action2(X,I,P,P2),timeout(T) )|
                                (det,
                                 actor,
                                 ((action1(P2),
                                  det,
                                  target(P1),timeout(T)) |
                                  (action20(I),det,
                                   ((target1(P1),
                                    difference(P1,[:=],[P])|
                                    (target(P1),
                                     difference(P1,[<=],[P]))
                                    )),
                                   message2(I,P,X),
                                   action2(X,I,P,P2),timeout(T)) )),

/* type : The buffer transfers the input DI to the output DO */
sig_assgn([[P1,P2,T]]) -->    (action,
                                det,
                                source(P2),
                                prep,
                                det,
                                (target1(P1) | target(P1)),timeout(T)) |
                                (det,
                                 actor,
                                 action,
                                 det,
                                 source(P2),

```

```

        prep,
        det,
        (target1(P1) | target(P1)), timeout(T)).

/* type : the device adds DATA and contents of REG and stores it in REG */
sig_assgn([[P2,add,'(P,Q,)',T]]) --> action3,
        det,
        source(P),
        prep,
        det,
        source(Q),
        sig_conntn,action,it,prep,det,target(P2),
        timeout(T).

% for sig_assgn sentences expressed in passive voice.
/* type :INT signal is asserted. And type : DAR is incremented */

sig_assgn([[P1,P2,T]]) --> det,
        ((target1(P1),difference(P1,[:],[P]))
        |(target(P1),difference(P1,[<=],[P]))),
        ((action_p1(P2),timeout(T)) |
        (action_p2(I),message2(I,P,X),
        action2(X,I,P,P2),timeout(T))).

/* type DI is transferred to DO */
sig_assgn([[P1,P2,T]]) --> det,
        source(P2),
        action_p,
        prep,
        det,
        (target1(P1)|target(P1)), timeout(T).

target([P1,<=]) --> type,
        (signal | port | carrier | store | no ),
        id(P1) ,
        (signal | port | carrier | store | no).

target1([P1,:=]) -->
        variable,id(P1).

source(P1) --> (data | contents | no),
        prep,
        det,
        type,
        (store | carrier | port | no),!,
        value(P1),
        (store | carrier | port | no).

```

```

timeout([after,P,ns,:]) --> [after,P,ns].
timeout([after,P,:]) --> [after,P].
timeout([:])--> [].

/*****
/* This section contains the terminals of the code */

when(P4,[if,'(,P4,)',then]) -->[when].
when(P4,[if,'(,P4,)',then]) -->[if].

when(P4,[elseif,'(,P4,)',then]) -->[alternatively,if].
when(P4,[elseif,'(,P4,)',then]) -->[otherwise,if].
when(P4,[elseif,'(,P4,)',then]) -->[alternatively,when].
when(P4,[elseif,'(,P4,)',then]) -->[otherwise,when].

when(P4,[if,'(,P4,)',then]) --> [].
when(P4,[elseif,'(,P4,)',then]) --> [].

while(P4,Y,[if,'(,P4,)',then,Y]) -->[while].

period      -->['.'].
period      -->['.'].
period      -->[].
connt       -->[and,at,the,same,time].
conntn      -->[also,'].
conntn      -->[also].
conntn      -->[].
conntn      -->[then,'].
conntn      -->[then].
sig_conntn  -->[also,'].
sig_conntn  -->[also].

sig_conntn  -->[and].
sig_conntn  -->[then].
sig_conntn  -->[after,P].
sig_conntn  -->['.',after,P].
sig_conntn  -->[].

value("1") -->["1"].

```

```

value("1") -->[high].
value("1") -->[set].
value("0")-->["0"].
value("0")-->[low].
value("0")-->[reset].
value("P1") -->["P1"].
value(P1)   -->[P1].

```

```

action      -->[transfers].
action      -->[transmits].
action      -->[applies].
action      -->[puts].
action      -->[reads].
action      -->[writes].
action      -->[strokes].
action      -->[loads].
action      -->[stores].
action      -->[latches].
action      -->[gates].

```

```

action_p    -->[is,transferred].
action_p    -->[is,applied].
action_p    -->[is,transmitted].
action_p    -->[is,gated].
action_p    -->[is,loaded].
action_p    -->[is,strobed].
action_p    -->[is,latched].
action_p    -->[is,gated].

```

```

action1("1") -->[enables].
action1("1") --> [activates].
action1("1") --> [asserts].
action1("1") --> [sets].
action1("1") --> [initiates].

```

```

action1("0") --> [disables].
action1("0") --> [deactivates].
action1("0") --> [deasserts].
action1("0") --> [resets].
action1("0") -->[clears].

```

```

action_p1("1") -->[is,enabled].
action_p1("1") -->[is,activated].
action_p1("1") -->[is,asserted].
action_p1("1") -->[is,made,high].

```

```
action_p1("1") -->[is,set].
action_p1("1") -->[is,initiated].
```

```
action_p1("0") --> [is,disabled].
action_p1("0") --> [is,deactivated].
action_p1("0") --> [is,deasserted].
action_p1("0") -->[is,reset].
action_p1("0") -->[is,made,low].
action_p1(P) -->[is,made,P].
```

```
action20(1) --> [increments].
action2(y,1,P,[P,+,1],S,S).
action2(n,1,P,[incr(P)],S,S).
```

```
action20(2) --> [decrements].
action2(y,2,P,[P,-,1],S,S).
action2(n,2,P,[decr(P)],S,S).
```

```
action20(3) --> [inverts].
action2(n,3,P,[not,P],S,S).
```

```
action20(3) --> [complements].
```

```
action_p2(1) -->[is,incremented].
action_p2(2) --> [is,decremented].
action_p2(3) --> [is,inverted].
action_p2(3) --> [is,complemented].
```

```
action3 --> [adds].
```

```
otherwise -->[otherwise].
alternatively -->[alternatively].
```

```
type --> [input].
type --> [output].
type --> [interrupt].
type --> [].
```

```
signal -->[signal].
signal -->[pulse].
port -->[port].
```

```
variable -->[variable].
```

```
id(P1) --> [P1].
```

store -->[register].

carrier --> [bus].

carrier -->[line].

no --> [].

data --> [data].

data --> [value].

data --> [values].

contents --> [content].

contents --> [contents].

rel(=)-->[equals].

rel(=)-->[is].

rel(=) -->[being].

rel(=)-->[remains].

rel(/=) -->[does,not,equal].

rel(/=) -->[does,not,

remain].

rel(/=) -->[is,not].

event(P,[not,P,"stable',and,P,=","1"])-->
det,[P,rises].

event(P,[not,P,"stable',and,P,=","0"])-->
det,[P,falls].

event(P,[not,P,"stable',and,P,=","1"])-->
[the,positive,edge].

event(P,[not,P,"stable',and,P,=","1"])-->
[the,rising,edge].

event(P,[not,P,"stable',and,P,=","0"])-->
[the,negative,edge].

event(P,[not,P,"stable',and,P,=","1"])-->
[the,falling,edge].

event(P,[not,P,"stable',and,P,=","1"])-->
[assertion].

event(P,[not,P,"stable'])-->
[P,changes].

prep -->[of].

prep -->[to].

prep -->[into].

prep -->[onto].

prep -->[on,to].

prep -->[from].

```

prep    -->[on].
prep    -->[in].
prep    -->[].
it      -->[it].
it      --> [].

signal(P1) -->det,id(P1),(signal | no).


actor --> [P].

det -->[the].
det -->[a].
det -->[an].
det -->[].

a -->[a].
a -->[].

repeat -->[this,is,repeated,continuously].
repeat -->[this,is,repeated].

itrtn_schme([while,P,loop]) --> [until],
            (un_cl(P) | cond(P)).
itrtn_schme([for,i,in,0,to,P,loop])--> [P1],sub(P1,P),[times].

sub(P1,P,S,S):- is(P,P1 -1).

un_cl([not,P1,=,P2]) --> [P1,reaches,P2].


/*****
/* These are some predicates that aid in
pretty printing of lists
and joining of three lists together */

and_or([and],P2)-->[with],cond(P2).
and_or([P3],P2)-->[P3],cond(P2).

and_or([],[])-->[].

add(P1,P2,P3,P4,S,S):-append(P1,P3,P),append(P,P2,P4).

pp([H|T]) :-!,pp(H),tab(1),ppx(T).
pp(then):- write(then),nl,tab(4).

```

```
pp(:):-write(;;),nl.
pp(loop):-write(loop),nl,tab(4).
```

```
pp(X) :- write(X).
ppx([]).
ppx([H|T]) :-pp(H),tab(1),ppx(T).
```

```
pp([H|T],X,S,S) :-!,pp(H),tab(1),ppx(T,X).
pp(then):- write(then),nl,tab(4).
pp(loop):- write(loop),nl,tab(4).
pp(:):-write(;;),nl,tab(4).
```

```
pp(X) :- write(X).
ppx([],X):-read(X).
ppx([H|T],X) :-pp(H),tab(1),ppx(T,X).
```

```
pp([H|T],P2,P3,S,S) :-nl,
printstring(" Which of the following
statements are to be repeated.
Answer with 'y' or a 'n'."),
nl,
!,pp(H),c(H,X1),tab(1),ppx(T,X1,P2,Y,P3).
pp(X) :- write(X).
ppx([],P1,P2,Y,P3):-append(P1,[],P2),append(Y,[],P3).
ppx([H|T],X1,P2,Y,P3) :-pp(H),c(H,I),append(X1,I,P),int(H,P,P1,Y,Y1),
tab(1),ppx(T,P1,P2,Y1,P3).
```

```
int([:],P,[],Y,Y1):- read(X),=(X,n),append(Y,P,Y1).
int(H,P,P,Y,Y).
c(H,[H]).
```

```
reverse(List, Rev,S,S) :-
    rev(List, [], Rev).
```

```
rev([H|T], WorkList, Rev) :-
    rev(T, [H|WorkList], Rev).
rev([], Rev, Rev).
```

```
pr([H|T],X1,P1,X2,P2,S,S):-ppr(H),int(H,X1,X3,T,T1,X4),
nl,pr(T1,X3,P1,X4,P2,S,S).
pr([],X3,P1,X4,P2,S,S):-append(X3,[],P1),
append(X4,[],P2).
ppr([]).
ppr([H|T]):- ppr(H),tab(1),ppr(T).
ppr(H):- write(H).
```

```
int(H,X1,X3,T,T,X4) :- read(X),(X,y),
append(H,X1,X3).
int(H,X1,X1,T,[],X4):- append(T,H,X4).
```

```
difference([H|T], L, D,S,S) :-
    member(H, L), difference(T, L, D,S,S).
difference([H|T], L, [H|R],S,S) :-
    difference(T, L, R,S,S).
difference([], _, [],S,S).
```

```
member(H, [H|_]).
member(H, [_|T]) :-
    member(H, T).
```

```
/******  
/* The end */
```

Vita

Meenakshi Manek was born and brought up in Madras, India. She completed her Bachelor's degree in Electrical Engineering at Regional Engineering College, Trichy, India. Then, she continued her studies in Virginia Tech to earn a master's degree. Meenakshi has now accepted the position of a CAD Engineer with Intel Corporation in Chandler, Arizona.

Meenakshi Manek