Generating VHDL Models from Natural Language Descriptions

W. R. Cyre, J. R. Armstrong, M. Manek-Honcharik, and A. Honcharik

The Bradley Department of Electrical Engineering
Virginia Tech, Blacksburg, Virginia

Abstract

This paper describes two approaches to the automatic generation of behavioral VHDL models from descriptions written in natural language. Both approaches are based on a modeling style in which behavior is represented by a system of interconnected processes. The first approach employs a semantic grammar to directly generate a single VHDL process from a paragraph written in a restricted English called ModelSpeak. The second approach accepts more general English and generates models consisting of multiple processes.

Introduction

The VHSIC Hardware Description Language (VHDL) [9] is rapidly becoming a industry standard for simulation and design verification of Hardware systems. At the same time, many integrated product specifications and descriptions are written in English. Preparing models from such descriptions is currently a manual process subject to individual interpretation. Because VHDL is a powerful and complex language, modeling styles may vary among modelers. A series of research projects have been undertaken at Virginia Tech to improve this situation. First, to limit the modeling style and eliminate some of the tedious syntax of VHDL, a graphical model capture tool called the Modeler's Assistant was developed [1,13]. Using this tool, a system is represented by a collection of processes (circles) which are interconnected by signals (lines). Process ports are denoted by small circles on the peripheries of the process circles. Ports may be shaded for inclusion in the sensitivity list of the process. This graphical representation of behavior is called a Process Model Graph (PMG). The actual VHDL code for the behaviors of the processes must be written by the modeler if it cannot be retrieved from a library of primitive processes.

The first natural language interface (Approach I) to the Modeler's Assistant described in this paper accepts a series of sentences describing the behavior of a single process and generated correct VHDL code for it. This interface requires the behavior be expressed in a highly restricted English called ModelSpeak [10]. Conditional expressions (if statements) may be nested to two levels, and interaction is invoked by the interface to resolve scope ambiguity of loops. This interface is implemented with a semantic grammar [3] in Prolog.

The second interface (Approach II) to the Modeler's Assistant for natural language is substantially more complex, but accepts more general English expressions and generates multiple-process models. In this approach, each sentence is parsed by a bottom-up, parallel chart parser to generate a set of valid parse trees. The trees are examined by a Semantic Analyzer [7] to determine one which is semantically correct, and a conceptual graph [14] (semantic network) of its meaning is generated. Once the conceptual graph of individual sentences are joined together [12], a program called the Linker [8] analyzes the graph to produce a Process Model Graph and VHDL process code for the Modeler's Assistant. Nodes in the conceptual graph representing verbs typically generate processes or the conditions which govern them. Nodes representing objects generate signals. The modeling style underlying the Linker produces a sizable collection of processes, each consisting of a single if_then statement. To avoid unnecessary bus resolutions, multiple processes assigning to the same signal are merged. Since signal types are generally not expressed in the English description, they may be added interactively later.

Before describing the approaches in more detail, related research on natural language interfaces is reviewed briefly.

Related Research

The PHRAN-SPAN system [6] by Granacki and Parker accepts digital system specifications in restricted English but translates it into an internal representation called the Design Data Structure (DDS). The PHRAN component detects the phrasal patterns and extracts corresponding concepts from its database to construct a conceptual dependency representation, and the SPAN component that analyzes the conceptual dependencies to produce a DDS description. PHRAN-SPAN is more general than Approach I here, and is comparable to Approach II, which

This research was supported in part by Semiconductor Research Corporation (Contract 92-DJ-230), the National Science Foundation (Grant MIP-9120620) , and Virginia's Center For Innovative Technology (Grant INF-92-005).

uses a different intermediate knowledge representation, and generates VHDL rather than DDS.

Another natural language processor called SOPHIE [3] was built by Burton. This system allowed students to query the machine using a restricted natural language and get feedback, aiding them in solving problems easily. Since this level of interaction pertained only to a limited domain, Burton developed the concept of 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. As the query sentence is parsed the semantic form or the "meaning" is constructed. The "meaning" either would be a call to the procedure that answers the query or a call to the procedural specialist that performs the action. In the Approach I interface described in this paper, the parsing technique is also based on a semantic grammar but 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 to generate the target output.

Approach I

ModelSpeak : The Input Language

The ModelSpeak Language [10] is based on the semantics of VHDL so that expressions are easily mapped into VHDL code. Its vocabulary is a set of words generally used in behavioral description of processes and includes many keywords from VHDL. Additional vocabulary words have been derived from published descriptions of digital components found in manufacturer's data sheets, books on modeling and computer systems. ModelSpeak descriptions of processes may consist of a series of sentences. The sentences may describe simple signal assignments and if_then statements with conditions consisting of Boolean expressions of relations. If_statements may be nested to two levels. The language also supports statements indicating for_loops, such as "This is repeated 4 times." Interaction initiated by the parser is used to resolve scope ambiguities of loops.

Since this approach presumes a Process Model Graph has been constructed and port names have been specified in it, these port names should be used in ModelSpeak expressions to permit linking the process code segments. Sample ModelSpeak expressions appear later in an example.

The Semantic Grammar

Some of the primary semantic constructs (non-terminals) of the language are listed in the table below.

Table 1: Selected Non-Terminals of ModelSpeak

Non-terminal	Description
proc_stat	process description
sig_assign	signal assignment construct
if_stat	if statement construct
action	action
cond	condition
rel	relation
target	destination signal

The ModelSpeak system may be viewed as a set of rule pairs. Each pair consists of a recognition rule for semantic analysis (parsing) of a description, and associated with it is a generation rule for VHDL. A generation rule may sometimes be empty or nil when the non-terminal symbol in the recognition grammar has no corresponding non-terminal symbol in the generation grammar. The semantic grammar includes rules for both isolated sentences and multi-sentence process descriptions.

The recognition and generation rules for simple signal assignment sentences are given below. Non-terminal symbols appear in italics. Non-terminals in recognition rules have *_a* suffices, whereas corresponding generation non-terminals have *_g* suffices. Note that only recognition rules 1, 5, 6 and 7 have corresponding generation rules which produce VHDL code.

sig_assgn_a ::= [*determiner_a*] *actor_a*
    action_a [*determiner_a*] *target_a* (1)
determiner_a ::= {the | a} (2)
actor_a ::= *id* (3)
action_a ::= {*action0_a* | *action1_a*} (4)
action0_a ::= {resets | deasserts | clears} (5)
action1_a ::= {sets | activates | asserts} (6)
target_a ::= [*type_a*] *id* [*class_a*] (7)
type_a ::= {input | output} (8)
class_a ::= {signal | port | line | bus} (9)

sig_assgn_g ::= *id_g* = *value_g* ; (1)
value_g ::= '0' (5)
value_g ::= '1' (6)
id_g ::= *id* (7)

Figure 1: Signal_Assignment Statement Rules

The top-level analysis rule pair for if_statements is shown in Figure 2.

if_stat_a::={when | if } cond_a sig_assgns_a
    [elsif_a][else_a] (10)

if_stat_g ::= if cond_g then sig_assgns_g
    [elsif_g][else_g] end if; (10)

Figure 2. If_Statement Rules

A Modeling Example

This textbook example [11] is a Direct Memory Access (DMA) interface module. The DMA interface module can be functionally partitioned into three processes [2] as shown in Figure 3.

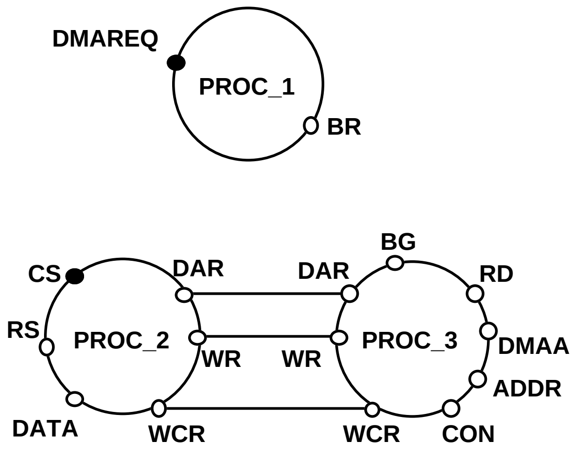

Figure 3: PMG for the DMA Interface Model.

Process_1: The response of the DMA controller to the bus request sent from the peripheral

The PMG shows a process named PROC_1 having an input port DMAREQ and an output port BR. The process has DMAREQ in the sensitivity list, indicated by a solid circle. The ModelSpeak description of the process is Sentence 1. The rules of Figures 1 and 2 are applied to this sentence to produce the VHDL code in Figure 4 below.

When DMAREQ is set, the controller activates the BR line. (1)

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

Figure 4. Sentence and VHDL for Process 1.

Process_2: Initialization by the microprocessor for the transfer.

The ModelSpeak description for this process consists of the single if_then_else expression shown in Figure 5.

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. (2)

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

Figure 5. Sentence and VHDL for Process 2.

Process_3: The transfer carried out by the controller when the bus grant signal is made high.

This process involves a loop and is described by a paragraph of six sentences of Figure 6..

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. (3-8)

Figure 6. Sentences Describing Process 3.

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 because 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. This is done so that the other actions performed when BG is high can be executed.

The for_loop indicated by the last sentence of the description causes the parser to enter an interactive mode. Here, the increment of DAR and decrement of WCR are to be repeated for each transfer, along with the wait for the transfer, and the transferring of DAR value to the ADDR bus. Based on this, the interaction should proceed as shown below. The user responses are the bold **y** and **n** entries.

```
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.
```

Figure 7. Interaction to Resolve Loop Scope Ambiguity.

The interaction terminates with the first **n** entry by the user and the VHDL code below 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;
```

Figure 8. VHDL for Process 3.

Approach II

Conceptual Graphs

The second approach employs four procedures to generate VHDL code from natural language sentences. This approach employs conceptual graphs as an intermediate representation of the meanings of sentences and to support integration of sentence meanings. Conceptual graphs [14] are bipartite, directed, labeled graphs consisting of concept nodes and relation nodes. In the following, a conceptual graph is represented by listing its concept nodes. Each concept node is immediately followed by all its incident-out relations with pointers to their target concepts. Concept nodes are denoted by two labels enclosed in square brackets. The first labels is a unique identifier and the second is a concept type label. The labels are separated by colons, as in [1: load] and [7:"STRB"]. Concept types of interest here include actions (load, reset), events (rise) devices (memory, register, bus), and values (data, '0'). The concept types form a lattice with respect to a generalization partial ordering. For example *register* is a *memory* is a *device* is a *object*. The type *load* is a *action. Action* and *event* types are subtypes of *behavior*. *Device* and *value* types are subtypes of *object*. Literals (enclosed in quotes) are identifiers, a subtype of object. Relation nodes may have two labels enclosed in rounded parentheses, a type label and a marker label (if present) which is typically the preposition or function word which indicates the relation. Relations may have any -arty, but only unary and binary relations are used here.

For example, Figure 9 shows the conceptual graph for Sentence 9 below.

The 8-bit data is loaded into the ACC register when STRB rises. (9)

```
[ 1 : load ]
      ->( object) -> [ 2 ]
      ->( destination : into ) -> [ 3 ]
      ->( condition : when ) -> [ 4 ]
[ 2 : data ]
      ->( size : adj ) -> [ 5 ]
      ->( det : the )
[ 5 : 8-bit ]
[ 3 : register ]
      ->( name ) -> [ 6 ]
      ->( det : the )
[ 6 : "ACC" ]
[ 4 : rise ]
      ->( agnt ) -> [ 7 ]
[ 7 : "STRB" ]
```

Figure 9. Conceptual Graph of Sentence 9.

Sentence Analysis

In the first procedure applied to an input sentence, it is parsed by a bottom-up, parallel chart parser [17] to determine all its valid parse trees. The syntactic grammar for this parser consists of about 120 rules. Because English is ambiguous, some sentences have several trees.

The second procedure, the Semantic Analyzer [7,15], identifies a "meaningful" parse tree and generates a conceptual graph (semantic network) representing its meaning. (In a separate procedure, a graphical representation of the interpretation of the sentence is fed back to the user for validating and editing the interpretation [16].) The Semantic Analyzer largely operates by retrieving the meaning of each significant sentence word. These meanings are represented by "basis" conceptual graphs. Because English is ambiguous, a given word may have multiple basis graphs. These basis graphs are assembled by rules associated with the parsing rules such that a concept of one basis graph is joined (merged) with a concept of another basis graph only if they are identical or if one is a specialization (subtype) of another.

The third procedure integrates the conceptual graphs of the sentences of a description into a single conceptual graph [12] by detecting concepts which refer to the same action or thing (are coreferences). Research is currently in progress to also integrate conceptual graphs which are automatically generated from other source language notations such as block diagrams, flow charts and timing diagrams [4,5]

Generating VHDL

The final procedure, which is implemented by the Linker, generates VHDL in a form acceptable to the Modeler's Assistant from a conceptual graph. The Linker employs a database which associates appropriate VHDL code segments with basis graphs for *behavior* concept types.

Interpreting a conceptual graph as VHDL Process Model Graphs is accomplished in two steps. First, the conceptual graph is analyzed to identify all behavior and object concept types. The *object* concepts become VHDL signals variables or values, and the *behavior* types generate the behavioral code. In addition, all subgraphs consisting of an action and either an agent or condition relation with a behavior are identified. These subgraphs are called condition_action links. In the graph of Figure 9, the behaviors are [1: load] and [4: rise]. The subgraph [1:load]->(condition:when)->[4:rise] is the only condition_action link. The graph also contains the three objects

[2:data]->(size)->[5:8-bit],
[3:register]–>(name)->[6:"ACC"] and
[7:"STRB"]

which are mapped to the VHDL signals:

data {type=BIT, size =8},
ACC {type=unknown, size=unknown}, and
STRB {type=unknown, size=unknown}, respectively.

The unknown types and sizes of these signals will have to be specified interactively later by the modeler.

In the second step of the Linker procedure, code fragments retrieved from the database for the object and behavior concept types are assembled into processes. The appropriate basis graphs and corresponding VHDL fragments are shown below in Figure 10. The VHDL in parentheses contributes to port declarations. A symbol marked by an asterisk is added to the sensitivity list of the process being generated.

Basis Graphs	VHDL

[1: data]
-> (size) -> [2]
[2: attribute]
	type=signal,
	name="data", value="data"
	stype=BIT; size=*size*

[1 : load] ->(object) -> [2] ->(destination) -> [3] [2: value] [3: memory]	(*object*:in, *destination*:out) *destination* <= *object*
[1 : rise] ->(agent) -> [2] [2: value]	(**agent*:in:BIT) (*agent*='1')and not(*agent*'stable)

Figure 10. Basis Graphs and VHDL Fragments

The second step uses the condition_action links to construct if_then statements for process bodies. The final result is a list of processes with associated VHDL code and a list of the signals connecting them, in essence a Process Model Graph. The lists generated for the Register example Sentence 9 are shown in Figure 11. Note that the names of signals STRB and ACC were derived from the conceptual graph. "data" is an anonymous name given to a signal that the conceptual graph implies but does not specifically name. The types of all signals were also automatically determined.

```
process LOAD_1  (STRB)
begin
        if ((STRB=1)and not(STRB'stable)) then
           ACC <=  data;
        end if;
end process;

--signal list
--      sig 0  STRB (BIT[0])
--      sig 1 data (BIT[8])
--      sig 2 ACC(BIT[8])
```

Figure 11: VHDL Process Code and Signal List

Post Processing

The linker also has a post processing mode to address the following problems. 1) If the conceptual graph translation process yields more than one process which drive the same signal, and the user desires merging, these processes are merged automatically into a single process. This eliminates the need to consider resolution functions in the linking process. 2) Sometimes the description of a signal in the resultant model will be incomplete in that the conceptual graph will not indicate its type or size.

(This was not the case in example given in this paper) Some assumptions are made in these cases, but these may or may not be correct, so in post processing one has the ability to change a signal's type and/or size.

Experimental Results

The Linker has been applied to conceptual graphs of many individual sentences and produced correct VHDL code. In addition it was applied to seven test cases which were picked from verbal descriptions of computer logic. Shown in Table 2 are the model names, number of English sentences, total number of English words, number of concepts in the conceptual graph, number of resultant VHDL processes, and number of lines of resultant VHDL code.

Table 2. Linker Generated Model Results

Model	Sent	Words	Con	Proc	Lines
FSM	5	98	38	7	30
RAM	4	44	22	2	25
Register 1	3	34	26	3	29
Register 2	3	43	20	1	25
Register 3	4	64	31	5	46
Register 4	5	62	30	6	44
Latch	10	106	50	8	60

In all cases studied, correct VHDL code was derived automatically. In some cases, type ambiguity of a VHDL signal had to be resolved by using the post automatic link editor. Details of the Linker operation are presented in [8].

Conclusions

The ModelSpeak language allows modelers to generate correct process models to instantiate a Process Model Graph using a restricted English. This relieves the modeler from VHDL syntax and permits students and modelers who do not know VHDL to produce VHDL models. ModelSpeak supports signal_assignment statements, Boolean expressions of conditions, nested if_then statements, and loops. The complex systems may be modeled provided they are decomposed into an interacting collection of relatively simple processes.

The operation of the linker is based on relationships between concept types and VHDL fragments. The concepts in a conceptual graph can be partitioned into two subsets: behavior concepts and object concepts. The behavior concepts result in VHDL code that drives a VHDL signal or in code that is a condition expression of an if_statement. This if_statement will then control the execution of other code which drives a VHDL signal. The object concepts map to VHDL signals or the characteristics of signals.

References

[1] J. R. Armstrong and D. Burnette, "Automated Assists to the Behavioral Modeling Process," Proc. First Int'l Workshop on Rapid System Prototyping, Research Triangle Park, NC, June 1990.

[2] .R. Armstrong, Chip Level Modeling with VHDL, Prentice Hall Inc. 1988.

[3] R. R. Burton, Semantic Grammar: An Engineering Technique for constructing Natural Language Systems, BBN Report #3453, Bolt, Beranek, and Newman, Inc., Cambridge, MA, December, 1976.

[4] W. Cyre, "The Conceptual Representation of Waveforms for Temporal Reasoning", IEEE T. Computers, 43(2), 186-200, February 1994.

[5] W. Cyre, "Mapping Design Knowledge from Multiple Representations," Proc. 1991 IEEE International Conference on Computer Design (ICCD 91), Boston, MA, October 1991.

[6] 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.

[7] R. Greenwood and W.R. Cyre, "Conceptual Modeling of Digital Systems from Informal Descriptions," MASCOTS'93, January 17-20, 1993, San Diego, CA.

[8] A. J. Honcharik, Generation of VHDL from Conceptual Graphs of Informal Specifications, Master's Thesis, Bradley Department of Electrical Engineering, Virginia Tech, Blacksburg, VA, July 1993.

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

[10] M. Manek, A Natural Language Interface To a VHDL Modeling Tool, Master's Thesis, Bradley Department of Electrical Engineering, Virginia Tech, Blacksburg, VA, July 1993.

[11] M. Mano, Computer System Architecture, Prentice Hall Inc. 1982.

[12] S. Shankaranarayanan and W. R. Cyre, "Identification of Coreferences with Conceptual Graphs," Proc. Int'l Conf. on Concpetual Structures, August, 1994.

[13] B. Singh, J. Wicks, P. Wright, and J.R. Armstrong, "The Modeler's Assistant: A CAD Tool For Behavioral Model Development," Proceedings of CHDL 93, 347-354, Toronto, Canada, April 1993.

[14] . F. Sowa, Conceptual Structures: Information Processing in Mind and Machine, Addison_Wesley, Reading, MA, 1984.

[15] J. F. Sowa and E. C. Way, "Implementing a Semantic Interpreter Using Conceptual Graphs," IBM J. Research and Development, Vol. 30, 57-69, January, 1986.

[16] A. Thakar and W. R. Cyre, "Visual Feedback for Validation of Informal Specifications," MASCOTS'94, January 1994.

[17] T. Winograd, Language as a Cognitive Process, Vol. 1: Syntax, Addison-Wesley, Reading, MA, 1983.