A Transformation for Integrating VHDL Behavioral Specification with Synthesis and Software Generation

Frank Vahid[†], Sanjiv Narayan[†] and Daniel D. Gajski
Department of Information and Computer Science
University of California, Irvine, CA, 92717

Abstract

VHDL signals and wait statements provide great expressive power for behavioral specification. However, due to their simulation semantics, most high-level synthesis tools severely restrict their use, eliminating much of their power. Thus, there exists a need for a tool to bridge the gap between an arbitrary VHDL behavioral specification and existing synthesis tools. In this paper, we introduce a set of transformations to convert a VHDL description with signals and wait statements to equivalent constructs that are easily handled by high-level synthesis. The proposed transformations greatly enlarge the synthesizable VHDL subset, thus increasing the usefulness and practicality of the language as an input to high-level synthesis tools. These same transformations can also serve as a basis for converting a VHDL process to a form suitable for generation of software.

1 Introduction

VHDL is rapidly gaining acceptance as a behavioral specification language serving as input to high-level synthesis tools [1, 2, 3, 4, 5, 6, 7] as well as to hardware/software codesign tools [8, 9]. However, several of its constructs, while possessing great expressive power, are not easily handled by existing synthesis tools. Two such constructs are the wait statement and the signal, both having time-based semantics.

The wait statement provides the capability to suspend the execution of a process until some condition is met. This condition can be a fairly complex combination of events, boolean expressions, and time. Hence a single wait statement can be used in place of tens of lines of traditional statements.

The signal not only provides the capability of maintaining data values defined over time, but also provides a very elegant means for representing fine-grained parallelism in a VHDL description (where processes provide for coarse-grained parallelism).

Even though wait statements and signals are so powerful, high-level synthesis tools and software translators have yet to allow their general use. The time-based semantics of these constructs are quite different from those for traditional sequential programming constructs, such as loops, variable assignments, and branches for which there are simple techniques to generate a control/dataflow graph [2, 10, 11, 12, 13]. As a result, tools accept only extremely limited forms of waits and signals. For example, some tools restrict each process to a single read or write of a signal. Other tools treat signals as variables, which changes the functionality. Most do not differentiate properly between simple, bus, and register signals. Wait statements usually can not appear in their general form. Some tools limit their use to detecting clock edges and resets. Others ignore one or more of the statement's various clauses (on, until, and for). Even then, the behavior of the wait statement is often interpreted incorrectly. For example, if signal S has the value "1", and the statement "`wait until S=1`" is encountered, S must first become "0" and then return to "1" before execution can proceed. Many tools instead implement the wait as sensitive to the level, not the edge, of the signal.

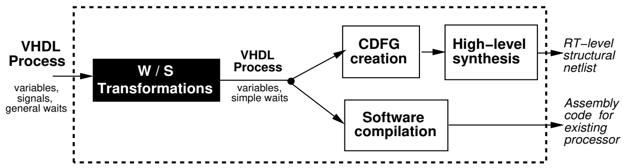

Figure 1: W/S transformation integration.

To bridge this gap between the desired use of the constructs with the current state of high-level synthesis tools, we have developed a set of wait statement and signal (W/S) transformations. Very general uses of waits and signals are automatically transformed into *functionally-equivalent* traditional sequential programming language constructs such as variables, loops, and branches. The latter are easily handled by existing tools. Thus, W/S transformations provide a major step forward in enlarging the synthesizable VHDL subset. In addition, traditional program constructs are also easily handled by software compilers, so these transformations contribute towards automating approaches in which it is desirable to generate either hardware or software from the same VHDL behavior. Figure 1 shows where the W/S transformations fit in with synthesis and compilation. Such transformation tools may become a necessary prerequisite when a single functional specification must serve as input to a variety of high-level synthesis and software generation tools. Similar transfor-

[†]F. Vahid is presently with the Department of Computer Science, University of California, Riverside, CA 92521. S. Narayan is presently with Viewlogic Systems Inc., 293 Boston Post Road West, Marlboro, MA 01752.

mation tools for performing other types of transformations before high-level synthesis are also beginning to emerge [6].

This paper is organized as follows. In Section 2 we summarize the relevant semantics of the signal and wait constructs, and describe the constructs to which signals and waits will be mapped. In Section 3, we introduce the W/S transformations. Since it would appear that the transformed VHDL would require excessive hardware due to the increase in the number of variables and statements, we include Section 4 to describe why this is not so; that in fact standard CDFG optimizations already found in most synthesis tools will yield efficient and practical hardware. Section 5 highlights results of applying the transformations to several examples.

2 VHDL signals and waits

2.1 Signals

In [14] templates are introduced for signal hardware at a process' interface and between processes. Although our focus in this paper is internal to the process, we briefly discuss those templates since we assume the high-level synthesis tool uses them. Figure 2 shows the template for the register signal. A bus signal differs in that it uses no level-sensitive latch, and a simple signal does not require tri-state drivers. Note that a process that drives a signal may require its own latch to store the driving value. For example, an assignment $S <= 50$ causes a process to drive S with the value 50 until S is assigned a different value in the future. We assume that the high-level synthesis tool uses the shown template as the interface for each process.

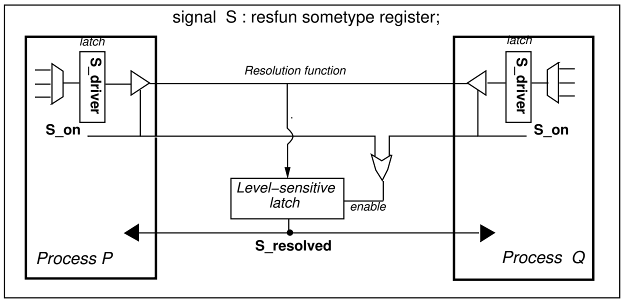

Figure 2: Hardware templates for three signal kinds

A signal assignment, $S <= value$, is quite different from the variable assignment, $V := value$. While the latter updates the value of V, the former merely *schedules* a new value for S. S is not actually updated until the next wait statement is encountered, so reads of S before that wait statement read the old rather than the new value. For example, the following code swaps A and B:

```
A <= B;
B <= A;
wait for 10 ns;
```

2.2 Wait Statements

The syntax of the VHDL wait statement is:

wait_statement	::=	**wait**	[sensitivity_clause] [condition_clause] [timeout_clause] ;
sensitivity_clause	::=	**on**	*signal*_name, {*signal*_name}
condition_clause	::=	**until**	condition
timeout_clause	::=	**for**	*time*_expression

The sensitivity list specifies the signals to which the wait statement is sensitive. When an event occurs on a signal in the sensitivity list, the condition clause specifies a condition that must be met for the process to resume execution. The timeout clause specifies the maximum time that the process will be suspended at the current wait statement.

The semantics of the wait statement are explained with the help of the flowgraph of Figure 3(a). The function *current_time* provides the current simulation time, while *advance_time* advances simulation time to the point when the next event occurs. These two functions are used to determine when the timeout interval has expired.

A process suspended at a wait statement can resume in two ways: either an event occurs on a signal in the sensitivity list *and* the condition in the condition clause evaluates to true, or the timeout interval in the timeout clause expires.

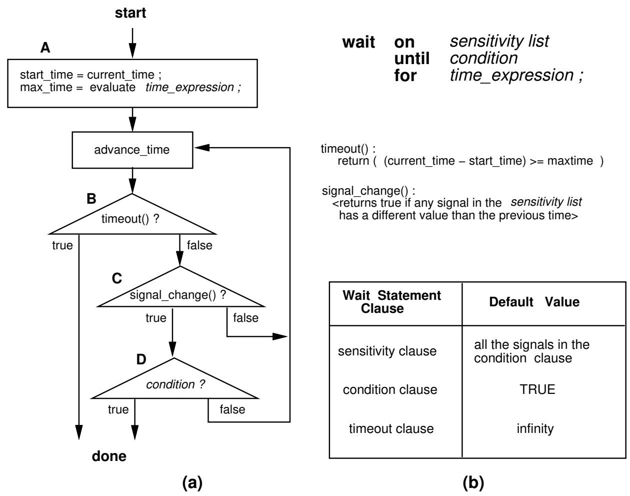

Figure 3: Wait statement flowgraph.

VHDL defines default values for all the clauses in the wait statement, shown in Figure 3(b). Thus, if some of the clauses are omitted in the wait statement, the resulting flowgraph can be simplified greatly. For example, in the absence of a condition clause, the default value is true and we can thus eliminate the branch-node D from the flowgraph of Figure 3. Similarly, in the absence of a timeout clause, the branch node labeled B can be eliminated along with the statement block A.

2.3 Problem Statement

Given a VHDL process possessing the signal and wait constructs described above, we wish to obtain a VHDL process which: 1. *Has no signals*; instead the process now accesses variables representing the original signal, and 2. *Has only trivial wait statements* that detect a clock edge.

Such statements are easily handled by existing synthesis tools. The output VHDL should be such that efficient hardware can be synthesized; in other words, the hardware created from the output VHDL should be what would be expected from the input VHDL, and not more complex.

We assume that the subsequent high-level synthesis tool uses the template shown in Figure 2. *S_driver* represents the value of the signal S driven by the process. Since several processes could be driving the signal, *S_resolved* represents its resolved value. This is also the value that is used in all reads of S. A tristate buffer may be required in case the driver is turned off in the process by a null assignment. *S_on* represents the control signal for this tristate buffer. *S_driver* and *S_resolved* are of the same type as the signal S while *S_on* is of type boolean. We assume that the subsequent synthesis tool recognizes these three variables. We do not currently permit the use of *after* clauses in signal assignments.

3 W/S transformations

Figure 4 shows the W/S transformations applied to the VHDL signal assignment and wait statements. We now discuss these transformations in detail.

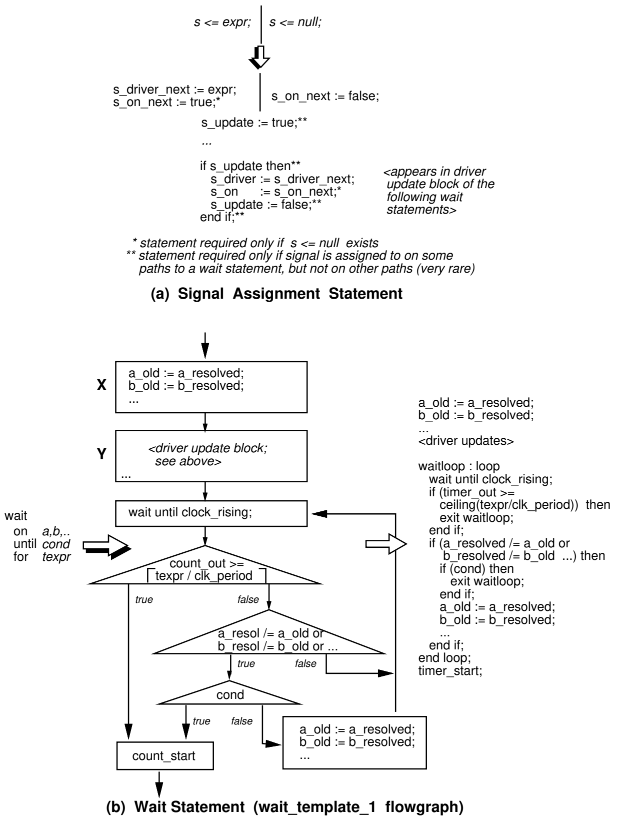

Figure 4: The Wait/Signal transformations.

3.1 Signal Assignment Statements

Figure 4(a) shows how signal assignment statements are represented using variables only. In a signal assignment "`S <= expression ;`", the driver for S is only updated with the value of the *expression* at the next wait statement. We need to store this next value of S in a new variable called *S_driver_next*. Thus, all assignments to S are replaced by "`S_driver_next := expression ;`".

In case the driver for signal S is turned off in the process, then each non-null assignment to S must turn on the driver. This can be achieved by adding "`S_on_next := true`" after the assignment statement, where the variable *S_on_next* represents the next value of *S_on*; *S_on* will be updated when the next wait statement is reached. A null assignment to S in the process should turn off the driver. This is achieved by replacing "`S <= null ;`" by "`S_on_next := false ;`".

3.2 Wait Statements

We will first define some of the terminology used in this section. A *statement block* is defined as the set of statements between any two successive wait statements. The term *preceding_paths* refers to all paths leading from any preceding wait statement to the current one, while *following_paths* represents all paths from the current wait statement to the next wait statement.

Before we present the transformations associated with wait statements, we briefly discuss how they are interpreted with a view to synthesize hardware. A wait statement indicates that all targets of signal assignment statements in the statement block preceding it have been updated with their new values. Thus, a wait statement implies an explicit clock boundary for synthesis. However, the synthesis tool is free to add more clock boundaries to implement the computations in the statement block, as we shall see later in Section 4.

In VHDL, all signals assigned a value in a statement block are updated at the next wait, and these updated values are available to statements following the wait statement. Thus, all computations that are performed in the statement block must have completed before the process can resume execution due to the expiration of the timeout interval. This implies that the process, when synthesized, is suspended at the wait statement for an amount of time equal to the difference between the timeout interval and the time required to perform the computations. For example, if the statement block requires 200 ns to compute the new values for all the signals, the next wait statement "`wait for 300 ns`" will effectively wait for 100 ns after the computations have been completed. On the other hand, a sensitivity list or a condition clause will be evaluated only after all the computations in the preceding statement block have been completed.

The flowchart representing the wait statement implemented using only "`wait until clock_rising`" is shown in Figure 4(b). The equivalent VHDL code generated for this template is also shown in the figure. This code is used by the W/S transformations as a template for replacing wait statements. The clock boundary can be specified in any manner acceptable to the synthesis tool.

To be able to monitor a change on the signals in the sensitivity list we need to store the current value ($S\_resolved$) of each sensitivity list signal S in the variable S_old. This is shown for signals in box X in the flowchart of Figure 4(b). The value S_old can then be compared with $S\_resolved$ after each rising edge of the clock to detect a change on S.

If the signal S was assigned a value in the preceding statement block, the driver value S_driver needs to be updated, as shown in box Y of in the flowchart of Figure 4(b). Since there could be several preceding paths leading up to the wait statement under consideration, it might be the case that the signal S is updated on only some of those paths. Thus, an additional boolean variable is set to true whenever the signal is assigned to in any of the paths. At the wait statement, if S_update is true, we can update S_driver with the variable S_driver_next (computed at the previous signal assignment).

VARIABLE	TYPE	DESCRIPTION	WHEN CREATED
S_driver	same as S	Value of S driven by process.	Always created if process writes S.
S_resolved	same as S	Signal value resolved from multiple process drivers. This is the value used in all expressions involving S.	If S is a resolved signal.
S_driver_next	same as S	Value with which S_driver will be updated at the next wait statement. Appears as the target for all assignments to S.	If S is assigned a value AND occurs in an expression in same statement block between two waits.
S_old	same as S	Old value of the signal, which is used to detect a change in the signal.	If S occurs in the sensitivity list of a wait statement.
S_update	boolean	Indicates whether S is to be updated at next wait statement. Set to TRUE when S is written. Set to FALSE at wait.	If S is assigned a value on some paths (but not all) between two wait statements.
S_on	boolean	Control Input of driver's tristate buffer.	If signal is assigned a NULL value
S_on_next	boolean	Value with which the S_on will be updated at the next wait statement.	If signal is assigned a NULL value

Figure 5: Variables created by the Wait/Signal Transformations for a signal S written to in a process

To implement timeout clauses, we can use a counter which is incremented on every clock. In pure VHDL behavior, the statements between two wait statements take zero time to execute. In synthesized hardware, these statements may require one or more clock cycles to execute. To maintain the same timing with respect to any external interface, the counter is started when we leave a wait statement so that it can be used by the next wait statement to determine the time elapsed since the previous wait. A timeout is detected whenever the counter value, $count\_out$, is greater than or equal to the timeout expression expressed in terms of clock cycles (i.e. $\lceil timeout\_expression/clock\_period \rceil$). As explained earlier, this also ensures that the time spent at the wait statement includes the time required to perform the computations in the preceding statement block. The function $count\_start$ initializes the counter. We assume that the functions $count\_start$ and $count\_out$ are recognized by the subsequent synthesis tool.

3.3 Common Simplifications

Figure 5 summarizes all the seven variables that may be required to implement a signal S in the most general case. However, we will rarely need all of these variables. In this section we present some simplifications that are part of the W/S transformations that reduce the number of variables used for any given signal.

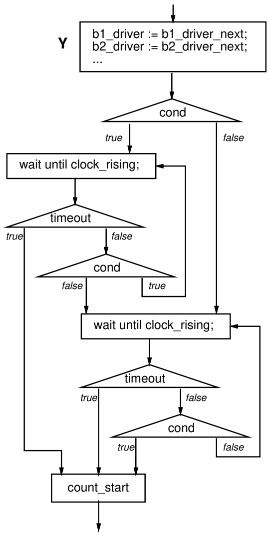

Figure 6: Template for wait statement where signals in sensitivity clause and condition clause are identical (wait_template_2)

First, for a signal S, we can eliminate the variable S_update if it is updated on all paths between every two successive wait statements, which is usually the case. Second, if S is an *unresolved* signal written only by the process under consideration, $S\_resolved$ always equals S_driver. Thus all occurrences of $S\_resolved$ in the transformed VHDL can be replaced by S_driver. Third, if the driver for S is never turned off using a null assignment, the boolean variables S_on and S_on_next are not needed.

Another simplification can be invoked with wait statements whenever all the signals in the condition clause and sensitivity list are identical. This is very common in VHDL descriptions, especially since this is the default when no sensitivity list is explicitly specified. In such cases, checking for a change on the signals and then evaluating the condition is redundant, because a change in condition value implies a signal in the sensitivity list has changed. If the condition is false before the wait statement, we only need to wait until it becomes true, which also implies that some sensitivity list signal must have changed. If the condition was true before the wait statement, we must first wait until it becomes becomes false, then wait until it becomes true. The template shown in Figure 6 can be applied to avoid using the S_old variables.

Even after the above-mentioned simplifications are performed, the transformations performed by the high-level synthesis tool would further optimize these variables and very few of them will actually be implemented as storage. The next section discusses these optimizations. The Wait/Signal transformations are summarized in Algorithm 3.1.

Algorithm 3.1 : Wait/Signal transformation

```
for each signal S loop
    Replace reads of S by reads of S_resolved
    if S is assigned null in the process then
        Replace each assignment S <= expr; by:
            S_driver_next := expr;
            S_on_next := true;
        Replace each assignment S <= null; by
            S_on_next := false;
    else
        Replace each assignment S <= expr; by
            S_driver_next := expr;
    end if
end for

for each wait statement w loop
    if w's sensitivity-list and condition-clause
            signals are the same then
        Replace w by wait-template-2,
            leaving section Y empty (Figure 6)
    else
        Replace w by wait-template-1,
            leaving section Y empty (Figure 4)
    end if
    for each signal S in preceding paths of w loop
        if all preceding paths assign to S then
            Add to section Y of template:
                S_driver := S_driver_next ;
            if S is assigned null in the process then
                Add to section Y of template:
                    S_on := S_on_next;
            end if
        else
            Add after writes to S in a preceding path:
                S_update := true;
            Add to section Y of template:
                if  S_update  then
                    S_driver := S_driver_next;
                    S_update := false;
                end if;
            if S is assigned null in the process then
                Add to section Y of template:
                    S_on := S_on_next;
            end if
        end if
    end for
end for

for each signal S loop
    if S is assigned to in the process
            AND S is unresolved then
        Replace S_resolved reads by S_driver reads
    end if
end for
```

4 Efficient-hardware synthesis

The many variables and complex templates introduced by W/S transformations might appear to lead to a complex hardware implementation. In general, this is not the case since the dataflow representation in a CDFG eliminates many intermediate variables, and CDFG transformations eliminate many branches and statements. It is not our purpose here to discuss CDFG representations and transformations in detail; instead we refer the reader to [2, 10, 15]. Here we shall illustrate that efficient hardware is obtainable by applying some of the common transformations to reduce registers, eliminate some paths, and eliminating the need for an external timer.

A common misconception of synthesis from VHDL is that variables correspond to registers. In fact, a variable may be implemented as a register or as a *wire*. A register is required only when the value of variable must be maintained across control step boundaries; in other words, the variable is updated in one control-step and read in a subsequent control-step. Since wait statements in the description denote explicit control steps, and since the *sig_driver_next*, *sig_on_next*, and *sig_update* values need not be maintained across those explicit control steps, then these variables will rarely require a register. For example, recall the swap example of Section 2. The variables *A_driver_next* and *B_driver_next* created by W/S transformations will be mapped to wires.

If a particular path of a branch can never be reached due to the condition for that path always being false, then the condition leading to that path, along with the path's operations, can be deleted. While such code is rarely written by the modeler, it occurs quite often after the W/S transformations. For example, consider the simple code portion shown in Figure 7(a). The code after transformations is shown in flowchart form in Figure 7(b). After CDFG creation, simple dataflow analysis of the branch condition results in determination that the condition is always true, as shown in Figure 7(c). Hence we can eliminate the branch condition and the false path. After doing so, *pc_old* is written but not read, so it too can be eliminated. Although these optimizations are performed on the CDFG, for illustrative purposes we show the equivalent code in Figure 7(d). Note its simplicity. Also note that as discussed above, no register will be needed for *pc_driver_next*.

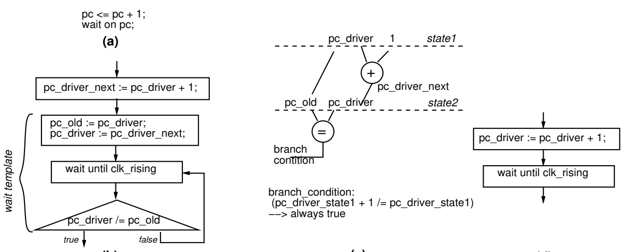

Figure 7: Eliminating false branches.

Finally, it is always possible for the synthesis tool to eliminate the need for an external timer, because after scheduling the CDFG, all the control-steps are known. A counter variable can be created that is incremented on each clock, and incorporated into the scheduled CDFG. Often this variable itself will then be eliminated, especially after loop unrolling transformations. For example, consider the code in Figure 8(a). Assume a clock period of 100 ns. W/S transformations would use the timer portions of the wait-template in Figure 4. Figure 8(b) illustrates a simple CDFG transformation in which a variable called *count* is declared; all occurrences of *count_start* are replaced by $count := 0$, all clocks are followed by $count := count + 1$, and all occurrences of *count_out* are replaced by *count*. Complete independence from an external timer is thus achieved. The design can be further improved by loop unrolling, as shown in Figure 8(c). The variable *count* becomes useless and is therefore eliminated.

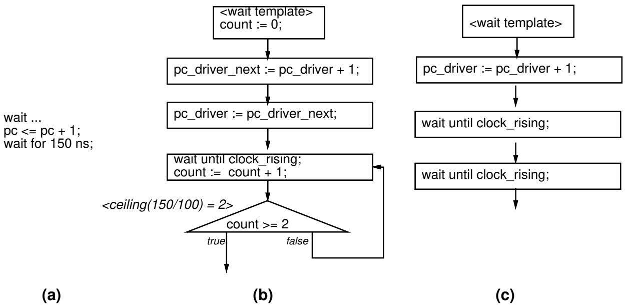

Figure 8: Eliminating the external timer.

5 Results

The W/S transformations have been integrated with our VHDL parser and internal representation, all of which make up roughly 15,000 lines of C code. The input to the W/S transformations is any general VHDL process, and the output is another VHDL process without any signal assignments and containing only wait statements sensitive to the clock.

Example	No. of lines in example	No. of lines after W/S transformation	Time for W/S transformations
Answering Machine	1057	2527	2.2
Ethernet Network Coprocessor	1154	1577	1.8
Microwave Transmitter Controller	657	1203	1.7
DRACO	282	407	1.2

Figure 9: Results of the W/S transformations.

Among the examples tested were a three-stage pipelined processor, the Rockwell DRACO I/O backplane custom integrated circuit, a telephone answering machine, an Ethernet network coprocessor, and a microwave transmitter controller. The table in Figure 9 shows the number of lines of VHDL process code before and after the transformations, as well as the CPU time in seconds for the transformations to run on a Sparc. In all cases, the signals and waits were successfully converted to sequential program constructs easily handled by high-level synthesis tools.

6 Conclusions

In this paper we have presented a transformation which will enable integration of VHDL behavioral specification and synthesis tools. The W/S transformations increase the expressive power of VHDL specifications that are synthesizable by enlarging the synthesizable subset of VHDL. This reduces the restrictions which are placed on designers writing VHDL behavioral descriptions intended as input to high-level synthesis tools. The W/S transformations are easy to incorporate into existing synthesis methodologies. In addition they can provide a path from VHDL to software which can be mapped to a processor, thus enabling the designer to generate hardware or software from the same VHDL description.

References

[1] J. Lis and D. Gajski, "VHDL synthesis using structured modeling," in *DAC*, pp. 606–609, 1989.

[2] R. Camposano, L. Saunders, and R. Tabet, "VHDL as input for high level synthesis," *IEEE Design & Test of Computers*, pp. 43–49, March 1991.

[3] P. Eles, K. Kuchcinski, Z. Peng, and M. Minea, "Compiling vhdl into a high-level synthesis design representation," in *EuroDAC*, pp. 604–609, 1992.

[4] A. Stoll, J. Biesenack, and S. Rumler, "Flexible timing specification in a vhdl synthesis subset," in *EuroDAC*, pp. 610–615, 1992.

[5] W. Ecker and S. Marz, "Subtype concept of vhdl for synthesis constraints," in *EuroDAC*, pp. 720–725, 1992.

[6] N. Wehn, J. Biesenack, and P. Duzy, "Scheduling of behavioral vhdl by retiming techniques," in *EuroDAC*, 1994.

[7] R. Walker and R. Camposano, *A Survey of High-Level Synthesis Systems*. Kluwer Academic Publishers, 1991.

[8] D. Gajski, F. Vahid, and S. Narayan, "A system-design methodology: Executable-specification refinement," in *EDAC*, 1994.

[9] M. Chiodo, P. Giusto, A. Jurecska, L. Lavagno, H. Hsieh, and A. Sangiovanni-Vincentelli, "Synthesis of mixed software-hardware implementations from cfsm specifications," in *International Workshop on Hardware-Software Co-Design*, 1993.

[10] G. DeMicheli and D. Ku, "HERCULES - a system for high-level synthesis," in *DAC*, 1988.

[11] A. Orailoglu and D. Gajski, "Flow graph representation," in *DAC*, pp. 503–509, 1986.

[12] D. Thomas, E. Langese, R. Walker, J. Nestor, J. Rajan, and R. Blackburn, *Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench*. Kluwer Academic Publishers, 1990.

[13] D. Gajski, N. Dutt, C. Wu, and Y. Lin, *High-Level Synthesis: Introduction to Chip and System Design*. Boston, Massachusetts: Kluwer Academic Publishers, 1991.

[14] L. Ramachandran, F. Vahid, S. Narayan, and D. Gajski, "Semantics and synthesis of signals in behavioral VHDL," in *EuroDAC*, 1992.

[15] R. Walker, *Design Representation and Behavioral Transformation for Algorithmic Level Integrated Circuit Design*. PhD thesis, Carnegie Mellon Unversity., April 1988.