Flexible Timing Specification in a VHDL Synthesis Subset

A. Stoll \hspace{1cm} J. Biesenack \hspace{1cm} S. Rumler

Siemens AG, Corporate Research and Development
Otto-Hahn-Ring 6, W-8000 München 83, Germany

Abstract

We present a VHDL subset for high-level synthesis allowing a flexible timing specification of the circuit interface such that the optimization potential of classical scheduling and allocation techniques can be fully utilized in the other hand, the algorithmic circuit specification can be validated by a conventional VHDL simulator, if the description style follows the proposed guidelines.

1 Introduction

Because VHDL was primarily designed for simulation, there are some difficulties in using VHDL as input for high-level synthesis (cf. [4]). This especially concerns the timing specification, because conventional simulators work with a precisely determined time, whereas common high-level synthesis techniques (cf. [14]) need some freedom for the scheduling of operations into time steps in order to produce optimal results.

Our first solution to this problem was to fix the execution cycles of the I/O operations only (cf. [9, 17]). Then algorithmic simulation before synthesis must coincide at the interface with register-transfer simulation after synthesis; notice that both simulations are cycle-based and ignore combinational delays. This principle, called "What you simulate is what you synthesize", allows the user to validate the algorithmic specification by simulation and the synthesis system to schedule at least the internal operations; their execution cycles are constrained only by data dependencies and fixed I/O operations, so there is a certain optimization potential.

Most scheduling algorithms, however, include also the interface operations. In particular, if such a sophisticated technique like path-based scheduling [2] is used, it can happen that the execution cycle of an interface read or write operation is rather irregular and unpredictable for the user, because it is path-dependent; we shall show that in general it is even impossible to backannotate the synthesized timing by simply inserting cycle delays in the original algorithmic description. Clearly, such an irregular interface behavior is unwelcome in many applications; in those applications where it is acceptable the communication of the circuit with the environment must be done by handshake. For such a description style, any specification error that can be detected by register-transfer simulation can already be detected by algorithmic simulation (supposed that the synthesis from algorithm to structure is correct), even though these simulations need not coincide.

Therefore, under certain conditions, validation of the algorithmic specification by simulation is compatible with a more flexible interpretation of the simulated timing by the synthesis system. In fact, the chosen method of communication between circuit and environment determines how close the implementation must come to the interface timing simulated at algorithmic level. For covering a wide range of applications, we extended our VHDL synthesis subset to support various communication paradigms, which can even be mixed in one algorithmic specification. The timing constructs in the subset must make the designer aware of the associated paradigm, because then the value of algorithmic simulation depends on the description style.

At algorithmic level a timing abstraction is necessary that separates the functional aspect of timing, which guarantees that the circuit in the given environment works correct "in principle", from the performance aspect of timing, which could be overcome by a technology that is fast enough. The latter aspect can be expressed by VHDL attributes constraining the optimizations of the synthesis system, whereas the first aspect must be represented in a way that permits validation of the specification ("Is what I asked for really what I want?"); for this, simulation is still the method of choice. Separating function from performance is the key idea of the timing concept we shall present; thereby we enable validation by simulation, but avoid over-specification for synthesis.

This paper is organized as follows: First, we briefly set the context of our considerations. Then, in Section 3, we discuss our three basic timing paradigms, in particular, their implications on validation and synthesis. In Section 4, we show how our timing concept enters into the internal design representation. The question whether such a timing specification can be satisfied by an implementation is answered in Section 5 by a necessary and sufficient condition. Finally, we round off the paper with detailing our VHDL subset.

2 Preliminaries

High-level synthesis traditionally starts from an imperative programming language; therefore, sequential statements inside a VHDL process are a natural candidate for the algorithmic specification (cf. [4]). A VHDL process communicates with its environment by signals; so, we rely on a one-to-one correspondence between the signals read or written by a process and the interface of the resulting circuit, whereas the inner life of the circuit has to be described with variables. Since timing-related constructs of VHDL such as the WAIT statement affect only signals, operations on variables can be executed at any time that fits the data dependencies. This reflects our opinion that it is sufficient to consider timing constraints at the circuit interface.

The result of high-level synthesis (cf. [7]) is a structural...
circuit description at register-transfer level, composed of a data path and a controller for switching multiplexors and loading registers. For completing the design, a low-level synthesis system can be used, which maps the components in the structural description (logic blocks, parameterized functional units etc.) onto a given target library. The synthesized circuit will always be synchronous, driven by the predefined signals clock and reset; combinational circuits will be omitted in the following. The VHDL semantics suggest to use registers for the output signals. Then the complete circuit is a Moore automaton, although the controller by itself can be realized as a Mealy automaton.

At register-transfer level, combinational delays are ignored and timing is expressed in terms of cycles. Nevertheless, the high-level synthesis system should estimate real delays of functional units and balance the hardware that is active during each cycle. In this paper, we shall concentrate on the topic how register-transfer simulation after synthesis can be related to algorithmic simulation before synthesis.

3 Methods of timing specification

For the specification of the interface timing, we provide three different methods:

- **Exact I/O determination** at the cycle level; the actual time in clock ticks can be advanced by calling a procedure cycles(n), which basically waits n times on the rising edge of the clock.

- **Time parameters:** As argument of the procedure cycles, certain integers for constants (e.g., \(tc.<name>\)) can be used for expressing a positive number of cycles that will be determined during synthesis. Choosing values for the time parameters typically amounts to solving a minimization problem under given constraints.

- **Synchronization anchors:** A call of the procedure anchor, which simply hides the statement cycles(1), indicates complete freedom for scheduling the preceding operations. Although both calls mean the same for the simulator, the interpretation for synthesis is different. First an arbitrary and even path-dependent number of cycles may be inserted. The end of this time period is characterized by asserting all signal changes, which should include a valid signal for handshake. Then these signals are kept constant for one cycle. Finally, succeeding time expressions are considered to be relative to this synchronization anchor. Indeed, this proposal was inspired by relative scheduling [12].

We point out that all three methods can be mixed consistently. But each method is distinguished by its favorite application domain, its impact on synthesis techniques, and its associated validation strategy.

3.1 Exact I/O determination

Typically, when certain protocols have to be synthesized, it is necessary to determine the I/O behavior completely at the cycle level. Then validation is most convenient, but there is only limited freedom for the scheduler. As only the I/O operations are fixed, it is possible to move internal operations as long as no data dependencies are violated. This degree of freedom can be used for optimizing area and cycle length. Nevertheless, in the worst case, arbitrary many operations have to be chained. But in many practical cases, where exact I/O determination is required, e.g. protocol synthesis, not much chaining will be necessary. If more cycles have to be spent for avoiding violations of the cycle length, we recommend to use the method of time parameters first. After the minimal cycle numbers under the constraint of the given cycle length have been found, synthesis can be rerun with the method of exact I/O determination.

3.2 Time parameters

In many applications involving time-multiplexing or digital signal processing, exact cycle numbers between I/O operations are not so important, as long as the distances — the time parameters — are kept constant and perhaps certain ratios are observed. In these applications it is important that reading or writing of interface signals occurs at fixed frequencies, which are divisors of the clock frequency; in particular, the synthesis system does not have the freedom to schedule the interface operations just as it would be convenient.

The method of time parameters presupposes that the functionality of the circuit does not depend on the exact values of the time parameters. Although the environment may need to know the values, it can be adapted — e.g., by iterative design — to the concrete values chosen by synthesis. Therefore, various scheduling and allocation techniques can be employed, which try to optimize performance properties, e.g., area under timing constraints such as upper bounds on cycle length and cycle numbers. Notice that the functional aspect of timing is observed by specifying with time parameters guaranteeing constant frequencies and given ratios, whereas the performance aspect of timing is taken into account by choosing values for the time parameters.

After synthesis, the concrete values obtained for the time parameters can easily be backannotated in the original algorithmic description. Then another simulation is possible with the true values, whereas before synthesis one may simulate with estimated values. Although, for a proper specification, the circuit functionality should not depend on the values of the time parameters, any doubt can be removed by a second simulation after synthesis.

```
if cond1 then
  x = a * c
else
  y = b * c
end if;

if cond2 then
  z = x * c
end if;
```

Figure 1: Backannotation

In general, even the possibility to backannotate the synthesized timing represents a constraint for the synthesis system. In the example of Figure 1, a synthesis system may achieve to spend one cycle for three of the four paths and two cycles for the fourth path. If, however, at any position in this algorithmic description, cycle statements are inserted such that three paths have the same cycle count, then also the fourth path will have this cycle count.
3.3 Synchronization anchors

Mainly in processor-like applications, AFAP-scheduling is wanted (cf. [2]), i.e., synthesis has the objective to minimize the number of cycles for all paths. The communication is done by handshake; therefore, it is not necessary to determine by the specification how long a path will be. Thus, we introduce a special time construct anchor and combine it with the semantics that the synthesis system is allowed to insert an arbitrary, data-dependent number of cycles before the signal update.

The arising discrepancies between simulation and synthesis can be compensated by a proper specification style. If the communication of the circuit with the environment is based on handshake instead of relying on certain execution times, then a specification error that can be found by simulation at register-transfer level after synthesis can also be found by simulation at algorithmic level before synthesis. Of course, this conclusion assumes that the specification error does not consist in a deviation from the proposed communication style. The special construct anchor, however, reminds the user of the necessary handshake. Moreover, a plausibility check can be done by simulating the algorithmic description after a statement wait for t with a random time t is inserted before each anchor.

Our strict semantics, when input signals have to be read and output signals have to be written, will sometimes cause some I/O register overhead. But exploiting this optimization potential, i.e., to formalize the use of sequential don't cares, is beyond the scope of this paper.

A synchronization anchor makes it possible to apply serial-to-parallel transformations that cannot be described adequately within a single VHSIC process. If \( p_1, p_2 \) are algorithms working on disjoint sets of variables without any timing construct, then a sequence

\[
\ldots; p_1; p_2; \text{anchor}; \ldots
\]

can be implemented using separate hardware with parallel controllers for \( p_1 \) and \( p_2 \) respectively. This kind of parallelization is orthogonal to path-based scheduling [2], i.e., path-based scheduling should be applied to \( p_1 \) and \( p_2 \) individually, because a sequential order of \( p_1 \) and \( p_2 \) would be preserved by path-based scheduling.

4 Internal representation

The first step of high-level synthesis is to translate the algorithmic specification, given in VHDL or another language, into an internal representation, which is either an annotated parse tree (cf. [6, 15]) or, more commonly, a graph representation of data and control flow, e.g., VT [13], DDS [11], YIF [1], or DFBS [16]. In this section, we shall describe how our timing specification can be expressed in such an internal representation.

We shall base our discussion on a design representation as given in [6], i.e., there is a separation between the behavioral and the structural domain, as well as between the control and data-flow. This results in four different views (see Figure 2), each of them represented by a directed graph:

- The control-flow graph CFG (behavior, control)
- The data-flow graph DFG (behavior, data)
- The control automation graph CAG (structure, control)
- The data path graph DPG (structure, data)

The behavioral graphs are built directly from the algorithmic VHDL specification, whereas the structural graphs are the result of high-level synthesis.

![Figure 2: Design representation](image)

In the following, we shall concentrate on the control-flow graph \((V, E)\), because there the timing specification will be represented. The nodes \( v \in V \) are the operations of the VHDL source, and the edges \( e \in E \) represent the predecessor-successor relationship among them, as implied by the sequential nature of a VHDL process. Nodes and edges are attributed, e.g., by the branching information. Of course, composed expressions in VHDL will result in a sequence of nodes, given in one of the orders allowed by the semantics.

Each timing construct will also be represented by a node \( v \in V \) having a special time attribute, which can be one of the following:

- A positive integer
- A symbol \( t_i \) (\( i \in \mathcal{N} \)) for a positive integer to be determined
- A symbol \( \infty \) for "indefinite"

The conversion of the timing specification in VHDL into these time nodes is obvious. The time nodes primarily constrain the schedule of the interface read/write operations, so their nodes are specially marked as I/O nodes; recall that our VHDL subset allows the use of signals only for the circuit interface.

Clearly, the intended semantics of the timing specification can be retrieved from the control-flow graph. This timing specification expresses the functional aspect of timing; in addition, one can subject high-level synthesis also to timing constraints representing the performance aspect of timing.

In practice, a real synthesis system may be well suited only for a certain subset of all possible timing specifications. But the focus of this paper is not on the problem how to synthesize a timing specification optimally, because the first step is to precisely state the problem that has to be solved. Then, a high-level synthesis system can--no matter how efficient its synthesis algorithms are--inspect the produced control automation graph and determine automatically whether the problem has been solved.

5 Consistency

In this section, we shall discuss the problem under which conditions a timing specification as described above can be satisfied in principle. Unrestricted use of arbitrary
In cycles. This number may depend on the path history and be executed. Then the values of all output signals must be kept scheduled at the same time plus an infinitesimal delta, in consistency of a timing specification.

The specification determines the schedule of all I/O nodes in such a bridge from v to w. The read nodes have to be executed immediately before the time that is obtained by adding up all time attributes until the time attribute of v inclusively. On the other hand, all write nodes must be scheduled at the same time plus an infinitesimal delta, in which all necessary operations on the bridge have to be executed. Then the values of all output signals must be kept constant for the time given by the time attribute of w. So far, we just described the simulation-oriented semantics of VHDL. But in the case that w has the time attribute ∞, we allow the synthesis system to increase the infinitesimal delay for the output calculation by an arbitrary number of cycles. This number may depend on the path history and the values of the input signals, but it must be the same for all output signals, and one of them should be specified by the user as a real signal. Then, after assertion, these signals are kept constant for one cycle.

Now it is obvious that a timing specification cannot be satisfied by real hardware, if there is a bridge in (V, E) that does not end with an co-attributed time node, but contains an inner loop, i.e., the bridge visits a node \( z \in V \) twice and \( z \) is different from the start and end nodes of the bridge, what means that \( z \) is not a time node. Otherwise, we say that the property (acyclic) holds. Of course, a loop could be expanded by loop unrolling, and there may also be the pathological case that a loop only exists as "dead code". We think, however, that such situations have already been detected and eliminated by compiler optimizations, if at all. So, the property (acyclic) is necessary only in the sense of static analysis.

Vice versa, the property (acyclic) is also sufficient, as we shall show. Let us first consider a timing specification that contains no co-attributed time nodes and has the property (acyclic). Without loss of generality, we may assume that each time attribute has the value one. Then each time node can be realized as a state \( S_i \) in the control automaton graph, and the state transitions are given by the possible paths in the control-flow graph. Now an equivalent VHDL specification can be obtained by adding a state variable \( \text{state} \) and describing explicitly the state transitions in a CASE statement. The result is sketched in Figure 3.

The transition functions \( F_i \) essentially contain the code pieces between the statements cycles(1) in the original specification; we also use auxiliary variables \( \text{invar}, \text{hvar}, \text{outvar} \) for all signals \( \text{in}, \text{out} \). The VHDL code following the CASE statement contains no timing-relevant constructs, is free of loops, and therefore describes a combinatorial function. According to the semantics of the initial WAIT statement, registers have to be inserted for all feedback variables \( \text{hvar} \). In fact, such a VHDL specification can be synthesized by a commercially available register-transfer synthesis system.

Besides other disadvantages, the synthesis method just proposed may lead to unlimited chaining in the synthesized hardware. If a synthesis system observes further constraints such as an upper bound on the estimated cycle length, then a timing specification cannot always be satisfied, but it is still possible to preserve the information about the specified timing in the generated control automaton graph, as has been shown in [17]. In the control automaton graph, which can be considered as a compacted control-flow graph, the requested timing can be represented by expressions, which are attached to some edges (Mealy automaton) and are obtained by adding up time attributes in the control-flow graph. Since a control automaton graph implicitly represents an actual timing, both timings can be compared.

In [17], we defined a property well-timed, which is a necessary condition for the control automaton graph that its actual timing can be specified in the algorithmic description. Essentially, well-timed means that all bridges with the same start edge also have the same length. Moreover, we described an algorithm how to introduce wait states in the control automaton graph such that the resulting graph is well-timed, but no path is longer than necessary. If this well-timed control automaton graph violates the specified timing, then the collection of all timing constraints was too restrictive.

This situation, however, can be avoided by using the specification method of time parameters. Then instead of a timing violation, an ILP formulation for these free parameters \( t_i \) is obtained, having the following structure:

\[
a_{0k} + \sum_{i=1}^{n} a_{ik} t_i \geq c_k \quad (k = 1, \ldots, m)
\]

with natural numbers \( a_{ik}, c_k \). The right hand values \( c_k \) are the lengths of the bridges, whereas the left hand expressions are the values of the time attributes after transformation from control-flow graph to control automaton graph. By the nature of this transformation, a left hand expression will quite often amount to a single time parameter \( t_i \). If for each \( k \) there is at least one positive \( t \) with \( a_{ik} > 0 \), a positive integer solution \( (t_1, \ldots, t_n) \) exists, and well-known algorithms can be applied for determining a solution that minimizes a given cost function of the time parameters \( t_i \).

Finally, we have to consider a timing specification that contains co-attributed time nodes and has the property

```vhdl
process
variable state : state.type;
variable invar, hvar, outvar : \ldots;
begin
wait until clock = '1';
invar := inig;
case state is
... when S_i \Rightarrow (state, hvar, outvar) := F_i(invar, hvar);
... end case;
outvar <= outvar;
end process;
```

Figure 3: Folded VHDL specification
is cut by an additional statement includes conversion functions synthesis and can be mapped directly to abstract cells. The basic logic attributes initialization only. We do not yet include a value 'Z' for time. This method generates a timing specification for a more restrictive timing specification by replacing all attributes \( \omega \) by the time attribute one; moreover, any loop is by the example of Figure 6. First, synchronization phase is run through. Therefore, a VHDL description of a sequential circuit in our synthesis subset has the structure shown in Figure 5. For simplifying such a description, a preprocessor like \texttt{cpp} is useful; actually, the example of Figure 6 uses the macros \texttt{nbegin(n)}, \texttt{step(t)}, \texttt{sync}, \texttt{nend} given in Figure 5.

\begin{figure}[h]
\begin{verbatim}
procedure cycles
  (signal reset, clock : in bit; n : in natural) is
begin
  for i in 1 to n loop
    wait until (clock = '1');
    exit when (reset = '1');
  end loop;
end;

procedure anchor
  (signal reset, clock : in bit) is
begin
  cycles(reset, clock, 1);
end;
\end{verbatim}
\end{figure}

Figure 4: Procedures \texttt{cycles} and \texttt{anchor}

Finally, the package \texttt{callas} contains the definition of the procedures \texttt{cycles} and \texttt{anchor}, as shown in Figure 4. According to this definition, the procedure \texttt{cycles} reacts on a synchronous reset by termination. So, we explicitly describe the effect of the \texttt{reset} signal. This further requires to embed the algorithmic description into an outermost loop, which is only left by a reset. In this case, the initialization phase is run through. Therefore, a VHDL description of a sequential circuit in our synthesis subset has the structure shown in Figure 5. For simplifying such a description, a preprocessor like \texttt{cpp} is useful; actually, the example of Figure 6 uses the macros \texttt{nbegin(n)}, \texttt{step(t)}, \texttt{sync}, \texttt{nend} given in Figure 5.

\begin{figure}[h]
\begin{verbatim}
process
  constant tc = 1;
  variable ... ;
begin
  <init-block>
  -- nbegin(n);
  cycles(reset, clock, n);
  if (reset = '1') then
    else
      outestloop: loop
        ...; cycles(tc);
        exit outestloop when (reset = '1');
      anchor(reset, clock);
      exit outestloop when (reset = '1');
      nend;
    end loop outestloop;
  end if;
end process;
\end{verbatim}
\end{figure}

Figure 5: Skeleton for synthesis

High-level synthesis may involve an encoding of abstract data types (e.g., enumeration types) as bit strings, because the quality of the result may depend on the chosen encoding (cf. [16]), which on the other hand is irrelevant for the functionality of the circuit. Because the synthesized structure works on bit strings only, its VHDL description must be supplemented by binding-elements, which we realize as extra processes that do the necessary data type conversions. This method can be extended to handle the synthesis of entities consisting of several communicating processes. Each process is synthesized separately as a combinational or sequential circuit. This results in entities that are connected with each other and the original entity by binding elements. For more details, we refer to [9]. For procedures and functions, we plan to support two different synthesis strategies, i.e., either inline expansion or handling as a single resource.

Finally, we illustrate our methods of timing specification by the example of Figure 6. First, synchronization on the start signal \texttt{instart} is described, dispensing with the construct "wait until". In the first cycle with \texttt{instart} = '1', the input signals \texttt{insig} are valid; their values are used for determining the initial values of the internal variables \texttt{inout}. In the main loop, new values for \texttt{inout} and the output signals \texttt{outsig} are continuously computed; e.g., one may produce a solution to a differential equation. The specification requires that the output signals are asserted with a constant frequency, which can be optimized by the synthesis system. So we use a time parameter \texttt{tc}.

Notice that the timing specification of Figure 6 allows
the synthesis system to apply the technique of loop winding, i.e., overlapping of successive loop iterations (cf. [8]). This technique requires a certain setup time when the loop is entered. The necessary degree of freedom is granted by the anchor \texttt{sync}. Another time parameter would have the same effect, but an anchor allows a data-dependent setup time, what can also be exploited in the calculation of the initial function \texttt{F_0}. The exit of the main loop is data dependent, and the signal \texttt{output} is used together with \texttt{instant} for handshake with the environment.

7 Conclusion

We extended algorithmic VHDL-description for high-level synthesis by a precise and adequate timing specification. It is adequate, because common high-level synthesis techniques get the degree of freedom they need for optimal results; and it is so precise that even validation of the specification by simulation is possible ("Is what I asked for really what I want?"). This validation depends on the proper description style, but our methods of timing specification allow an adequate low-level description of higher communication primitives such as the input and output commands in CSP [10]. Therefore, one possibility to prevent improper specifications consists in starting from an even higher level of abstraction. Moreover, since it is clearly defined, under which conditions an implementation satisfies a specification, one can use formal methods for verifying whether a given implementation belongs to the class of implementations defined by the specification ("Is what I got really what I asked for?").

References