FORMAL DEFINITION OF AN ABSTRACT VHDL'93 SIMULATOR BY EA–MACHINES

E. BÖRGER, U. GLÄSSER, W. MÜLLER

PUBLISHED IN:

C. Delgado Kloos and P.T. Breuer, editors,

"Formal Semantics For VHDL",

KLUWER ACADEMIC PUBLISHERS, Boston/London/Dordrecht, 1995

1

FORMAL DEFINITION OF AN ABSTRACT VHDL'93 SIMULATOR BY EA–MACHINES

Egon Börger*, Uwe Glässer, Wolfgang Müller*****

**Dipartimento di Informatica, Università di Pisa, Pisa, Italy*
***Heinz Nixdorf Institut, Universität-GH Paderborn, Paderborn, Germany*
****Cadlab, Universität-GH Paderborn, Paderborn, Germany*

ABSTRACT

We present a rigorous but transparent semantic definition for VHDL corresponding to the IEEE VHDL'93 standard. Our definition covers the full behavior of signal and variable assignments as well as the behavior of the various wait statements including delta, time, and postponed cycles. We consider explicitly declared signals, ports, local variables, and shared variables. Our specification defines an abstract VHDL'93 interpreter which comes in the form of transition rules for evolving algebra machines (EA–Machines) [18]. It faithfully reflects and supports the view given in the IEEE VHDL'93 standard language reference manual. The definition can be understood without any prior formal training. We outline our definition running a VHDL program.

1 INTRODUCTION

An approach to the definition of a formal semantics of the IEEE Std-1076 hardware description language VHDL'87 [33], as well as of the recently published VHDL'93[1] [34, 2, 22], is of great interest for hardware design and verification [5, 37].

For general hardware design, formal models are of great interest for a formal comparison of hardware descriptions which may be represented by different means, e.g., different hardware description languages such as VHDL, M, UDL/I,

[1] This standard is also known as VHDL'92

and Verilog. For the specific domain of VHDL, formal verification approaches typically deal with subsets of VHDL'87 since the domain of application is generally restricted to deterministic synchronous sequential circuits clocked by a single clock including the possibilities of resynchronization and asynchronous parts [15, 14]. It is not obvious to identify the relationship between the official VHDL IEEE standard and the subsets formalized for hardware verification since the standard language reference manual gives a natural language definition of the fairly complex behavioral semantics of VHDL. For standardization effort a formal model is not only useful for deriving valid subsets from the language but also for current language extensions. For instance, in the case of analog VHDL (VHDL-A, IEEE subPAR 1076.1) it is necessary to clearly define the interaction between the event-driven VHDL'93 simulator and the continuous simulation model.

To date, there is no coordinated effort for defining the formal semantics of full VHDL which may serve as a standard reference. The basic problem for the definition of such a standard formal model mainly lies in the complexity of VHDL'87 and yet increases with enhanced properties of the upcoming VHDL'93. In our approach we address the problem in achieving a formal, yet human–processable model covering full behavioral VHDL'93. This makes it possible to explicitly compare different definitions or implementations of the language, to evaluate them, to discuss prototypes, and to give a formal expression of standard requirements. Our definition provides the VHDL expert with a precise model of VHDL'93 along the lines of the IEEE standard reference manual [34]. Once having reached an understanding of the basic concepts of distributed EA–Machines our model may also serve as an introduction to the new concepts of VHDL'93 for VHDL users. Due to our experience, this understanding can be achieved without any major effort since the definition of distributed EA–Machines follows basic patterns of classical programming concepts.

Our rigorous definition of the VHDL'93 simulator covers full elaborated VHDL including the new features of postponed processes, rejection pulse limit, and shared variables [34]. We represent a detailed formal investigation of the VHDL'93 language reference manual in terms of Gurevich's (distributed) EA–Machines, also called *distributed evolving algebras* (EAs) [17, 18]. The definition covers the interaction of the simulation kernel process with the user defined processes, their suspension and resumption. We consider variables, explicitly declared signals, and ports including their driving, effective, and current values. Thus, this chapter is a continuation of the work presented in [3]. Out of the complete set of all syntactically correct VHDL statements we restrict attention to those which characterize VHDL and whose behavioral semantics is non–trivial. For example, we ignore features such as generics and component instantiations. The latter, for instance, can be ignored since instantiated components can be transformed into an equivalent description of hierarchical blocks (VHDL'93 LRM, §9.6.1). We also ignore syntactical constructs like concurrent signal assignments since each concurrent signal assignment can be transformed into an equivalent process statement (VHDL'93 LRM, §9.5).

The remainder of this chapter is organized as follows. Section 2 gives an overview over related work. In Section 3 we briefly review what is needed from distributed EA–Machines. In Section 4 we develop a mathematical definition of VHDL in terms of EA–Machines. For the correct understanding of that section the reader should be familiar with the basics of VHDL'87. In Section 5 we analyze the example given in Appendix B.[2] Section 6 concludes this chapter and gives an outlook to future work.

2 RELATED WORK

In the literature there are various well-known approaches to the formal semantics of VHDL for the verification of VHDL descriptions in the context of hardware design. They use temporal logic, functional semantics, denotational semantics and operational semantics, mainly applying Boyer-Moore Logic, Process Algebras, Petri-Nets, etc.

Borrione and Paillet [6] have investigated the semantics of a VHDL'87 subset in terms of a functional model. Salem in [31] defines evaluation functions based on the VHDL'87 subset P-VHDL which has been identified for formal verification in PREVAIL [32, 5]. A definition of a subset of VHDL'87 semantics in terms of Boyer-Moore Logic is presented by Read and Edwards in [27]. Russinoff [30] presents a mathematical definition of a hardware description language in terms of Boyer-Moore Logic admitting a semantics-preserving translation to a subset of VHDL'87.

A process algebra approach is presented by Bayol et al. in [1] translating a verification oriented VHDL (VOVHDL) into CCS for CCS-based verification. VOVHDL is an overlanguage of VHDL dedicated to specify the communication of processes at system level. A different algebra approach in the context of the functional specification methodology FOCUS can be found in [16]. Therein, Fuchs and Mendler define the semantics for delta-delay VHDL'87 by a transla-

[2] This example was given by the editor in [4].

tion to streams and stream-processing functions. They investigate VHDL from an abstract point of view without considering the underlying simulator.

Davis [13] has introduced a denotational semantics of the VHDL simulation cycle by the use of an intermediate language derived from a limited behavioral VHDL'87 subset. Breuer et al. define a functional and denotational semantics of the relevant behavioral VHDL'87 statements in [7, 8]. Progress in the definition of a denotational semantics can be found in [9]. [10] represents the continuation of their work.

A detailed structural operational semantics of a VHDL'87 subset, i.e., Femto-VHDL, for HOL verification is presented by Van Tassel in [35] (see also [36]). Damm et al. define the semantics of VHDL'87 through interpreted Petri-Nets [11]. In [12] detailed structural operational semantics is defined based on transition systems and used for formal verification against timing diagrams given by a linear first-order logic [12]. The formal verification is embedded in a hardware design environment which supports an extension of VHDL'87 (VHDL/S) for high-level design.

Work on interval temporal logic on a VHDL'87 subset was introduced by Wilsey in [38]. Reetz and Kropf provide a flowgraph semantics to VHDL'87 descriptions in order to facilitate the embedding of VHDL in high order logic [28] (see also [29]). Work by Marcus and Levy [21] (Core VHDL) and Levy et al. [20] considers formal verification in the context of the state delta verification system (SDVS). The internal logic (state delta logic) is a variant of temporal logic specifically tailored to be amenable to descriptions of computations and to proofs by symbolic execution. The specifications and claims of correctness are written in the state delta language which is a large subset of ISPS, Ada, and VHDL'87.

Müller introduces a modular framework defining a High-Level Semantics of behavioral VHDL'93 [23]. Therein the static semantics of TINY-VHDL is sketched by denotational means whereas the dynamic simulation semantics is sketched by partially ordered events which define Petri-Net-like structures.

Olcoz and Colom introduce a translation of full elaborated VHDL'87 to Colored Petri-Nets in [24]. In [25, 26] they provide detailed investigations of the VHDL'87 simulation cycle. In [26] they give a classification into three semantical layers: syntax checking and design library building, elaboration, and execution.

Our formal specification comes in the form of (distributed) EA–Machines. EA–Machine specifications combine the advantages of the operational and the functional approach to semantics. EA–Machines perform conditional destructive assignments which come on the abstraction level of function updates where arbitrary functions are allowed. This permits to tailor the operational view to any desired level of abstraction. In the case of VHDL this gives us the possibility to define our model along the lines of the standard language reference manual.

3 EA–MACHINES

Gurevich introduced EA–Machines in [17, 18], called there *Evolving algebras*. EA–Machines can be understood as 'pseudocode over abstract data', without any particular theoretical prerequisites. In order to make this chapter self-contained, we list here however the basic definition and refer for a rigorous formalization to [17, 18].

The abstract data come as elements of (possibly not furthermore specified) sets (domains and universes) which we denote by capitalized words. The operations allowed on universes are represented by partial functions. Thus, we have heterogeneous *structures* $(D_1, ..., D_n; f_1, ..., f_m; P_1, ..., P_r)$ with domains D_i, functions f_j, and relations P_k. For reasons of uniformity, we denote relations P by their boolean-valued function ξ_P, assuming that $BOOL = \{true, false\}$ is one of the domains. Structures without relations are traditionally called *algebras*.

We use such algebras as formal representations of *states* of the system we are going to describe. Since we want to emphasize that an algebra represents a state of a system, we call it a *static algebra*.

State transformations are reflected as transformations of static algebras. Dynamic changes of static algebras are obtained by executing *update instructions* of form

$$f(t_1, \ldots, t_n) := t$$

whose execution is to be understood as *setting* (modifying) the value of function f at the given arguments. We write $f(x) = undef$ if "f is *undefined* at x". Note that the 0-ary functions play the role of *variables* in programming languages.

A *sequential EA–Machine* is defined by a finite set of *transition rules* of form

$$\textbf{if } Cond \textbf{ then } Updates$$

where $Cond$ (condition or guard) is a first-order expression, the truth of which triggers *simultaneous* execution of all update instructions in the finite set $Updates$.

We are usually only interested in states reachable from some designated initial states, which may be specified in various ways. An EA–Machine often comes together with a set of integrity constraints, i.e., extralogical axioms and/or rules of inference, specifying the intended domains. In this chapter, our rules will always be constructed so that the guards enforce consistency of updates.

We give a simple example which illustrates sequential EA–Machines:

$$\begin{array}{l} \textbf{if } Condition \\ \textbf{then } A := B \\ \qquad\;\; B := A \end{array}$$

Example 1: Exchanging Values

This example defines the simultaneous update of the 0-ary functions A and B. Since the assignments are performed in parallel A becomes the value of B and vice versa. These updates are performed each time $Condition$ evaluates to $true$.

Besides simultaneous execution of multiple update instructions guarded by a condition, there is another form of parallelism which appears in sequential EA–Machines. This parallelism is expressed by allowing variables to appear in the update instructions of transition rules; the $DOMAIN$ over which a variable is supposed to range is declared by the condition $D \in DOMAIN$ appearing in the guard of the rules. Executing such a rule means to execute the rule for each instance D in its declared $DOMAIN$ simultaneously. The following example gives an illustration of this use of variables.

$$\begin{array}{l} \textbf{if } List \in LIST \\ \textbf{thenif } List \neq \langle\rangle \\ \quad\;\; \textbf{then } List := tail(List) \end{array}$$

Example 2: Remove the First Element of all lists

This example defines a rule specifying that each non–empty *List* from the domain *LIST* is to be replaced by the list's tail. The expression $List \in LIST$ is used as an abbreviation referring to any valid instantiation of *List* within the underlying domain *LIST*.[3] A characteristic example which we will use later has the form

$$\textbf{if } S \in SIGNAL \wedge condition(S) \textbf{ then } \ updates(S)$$

where $condition(S)$ is a condition and $updates(S)$ is a set of update instructions in which S does appear. The meaning of this rule is to simultaneously execute $updates(S)$ for each signal S which satisfies $condition(S)$.

We shall assume that we have the standard mathematical universes of booleans, integers, lists of whatever etc. (as well as the standard operations on them) at our disposal without further mention. We also use standard abbreviations like nesting of **if** 's etc.

In this chapter, we want to formalize how concurrently running VHDL processes update values under the supervision of the simulation kernel process. For this purpose we use *distributed EA–Machines*.[4] They are given by a finite number of *modules* each of which is assigned to a finite number of *agents*. For details we refer to [18]; in this context it is sufficient to say that each module is a sequential EA–Machine which is executed concurrently by the agents with which it is associated. Thus, a distributed EA–Machine can be seen as the definition of a set of concurrently running agents. Each agent is specified through a finite set of transition rules operating on a globally shared structure.

The model of distributed EA–Machines directly applies to our view of VHDL whose agents are n user defined processes and one kernel process. Our VHDL specification comes in the form of two modules, one for the kernel process and one for the asynchronously operating agents of user defined processes.

[3] In the remainder of this chapter domains are denoted by capitalized names whereas variables are represented by the same name but with only the first letter capitalized. In most cases this convention allows to skip the definition of which variable belongs to which domain, e.g., skipping the first condition in Example 2.

[4] In some articles distributed EA–Machines are referred to as concurrent evolving algebras.

4 THE FORMAL MODEL

For the correct understanding of this section the reader should be familiar with the terminology of the VHDL'87 or the VHDL'93 language reference manual. For this the reader is referred to the glossary of [33, 34]. In this section we first introduce the basic concepts for defining the simulator. Thereafter we present the formal definition of various statements, i.e., variable assignment, signal assignment, and wait statements. Finally, we give a definition of the simulation kernel process.

4.1 Basic Concepts

The Simulation Cycle

The VHDL'93 time model of event driven simulation is based on a finite number of user defined processes $P \in PROCESS$ which—under the supervision of the simulation kernel process—concurrently compute new VALUEs for given $SIGNALs$ and $VARIABLEs$. There is a mutual exclusion of the kernel process and the concurrently running user defined processes, i.e., the kernel process starts its execution if all user defined processes are suspended and vice versa (see Figure 1).

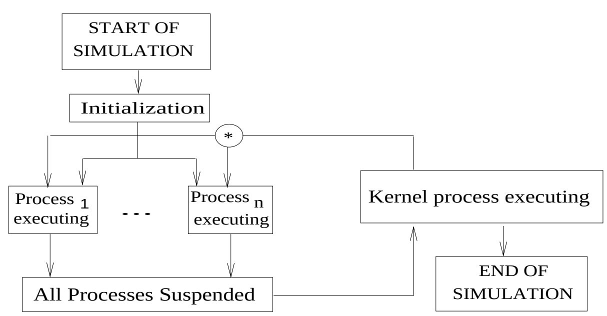

Figure 1 Execution of Simulation Cycles

Given the underlying discrete VHDL time model, the domain $TIME$ is linearly ordered and contains the distinguished element T_c for *current time*. Assignments to signals are performed by user defined processes and may cause events at specified points in time. Each user defined process is executed until it suspends. A process becomes *suspended* upon reaching a wait statement, which then delays the process execution until

a) the *timeout* expires, or
b) one of the associated signals is updated, or
c) a given expression becomes true if one of the corresponding signals is updated.

If all user defined processes are suspended, the kernel process executes and

(i) determines the value for the next time point T_n,
(ii) sets the new current simulation time T_c, if required,
(iii) updates the current values of the relevant signals, and
(iv) resumes the suspended processes which are sensitive to the signal changes or timeouts.

VHDL'87 distinguishes between so–called delta cycles and time cycles (time points). For a time cycle, the current simulation time T_c is advanced by the kernel. Delta cycles are introduces to model causality in simulation (infinitesimal delays). Between two delta cycles the kernel process does not advance T_c.

In VHDL'93, processes are further classified into so–called *nonpostponed* and *postponed* ones. The nonpostponed processes of VHDL'93 correspond to the user defined processes of VHDL'87. The postponed processes are executed only on request of time advancement, i.e., during the last delta cycle at the current time. Thus, we have to distinguish three consecutive *cycles* determined by the kernel process (see also Figure 6): in the *delta cycle*, in which the *current time* T_c does not change, nonpostponed processes are executed; in the *postponed cycle* when the resumed postponed processes are *executed*; and the *time cycle* at the beginning of which T_c is advanced. Note the careful use of the terms "execute" and "resume". Postponed processes may resume on events or expired timeouts during any simulation cycle but their execution is postponed to the next postponed cycle.

In our model, the kernel decides the type of processes which are executed during the next simulation cycle by setting *phase* $\in$ { *execute_postponed, execute_nonpostponed*}. This variable denotes the global state of the simulator. If *cycle* = *postponed_cycle*, the kernel executes the already resumed postponed processes by setting phase to *execute_postponed.* If *cycle* $\in$ {*delta_cycle,* *time_cycle*} the kernel assumes in sequence the phases: *update_driving_values, update_effective_values, update_current_values, resume_processes* (see Figure 5). Before resuming processes and executing them, the signal changes have to be propagated through the design hierarchy (blocks) by computing the driving, effective, and current values of the relevant (active) signals (see also [26]).

The VHDL'93 LRM §12.6.4 defines the initialization phase which has to be performed prior to the first simulation cycle. As initialization for our model we suppose *cycle* = *delta_cycle*, current time T_c to be set to 0, and the initialization of attributes, signals, and drivers through the simulator to be according to the definitions in [34]. Unless otherwise stated all functions are assumed to be $undef$ and sets to be empty sets. Finally, we assume that first each nonpostponed process and then each postponed process is executed until it suspends.

Signals

A user defined process P cannot immediately assign a value to a signal S. A signal assignment schedules a value val, desired at time t, into a sequence $driver(P, S)$ consisting of pairs (val, t). Those pairs are called *TRANSACTIONs* (see Figure 2). Basically, to each signal assignment exactly one driver $driver(P, S)$ is associated. The transactions of a driver are linearly ordered by their time components. By definition of the IEEE standard, the time component of the first element of each driver is $\leq T_c$. The time components of all the other transactions (in the tail of the driver) are $> T_c$. If the first element of a driver is updated in a given simulation cycle, then the driver is said to be *active* during the current simulation cycle.

Signal updates are performed by the kernel process only when all other processes are suspended. Since for each signal S its *value* is usually updated considering the set of all *sources(S)*, possible conflicts between multiple sources of S are resolved via a user defined resolution function; we represent the latter by a *resolved_value* function. In order to identify resolved and unresolved signals we set $resolved(S) \in \{true, false\}$. The domain $SIGNAL$ is partitioned into ports and explicitly declared signals. The latter are distinguished by the function $declaredSignal$. In the case of ports we distinguish them by their *mode* which is in the set {*inPort, outPort, inoutPort, buffer*}. We additionally set *mode* to a value in {*unconnected_inPort, unconnected_outPort, unconnected_inoutPort, unconnected_buffer*} if we have to identify unconnected ports.

When only considering non–hierarchical descriptions, i.e., excluding ports, the sources of a signal are drivers only (see Figure 2). In that case, the value of each signal S is determined from the first elements of $drivers(S)$ eventually applying a resolution function.

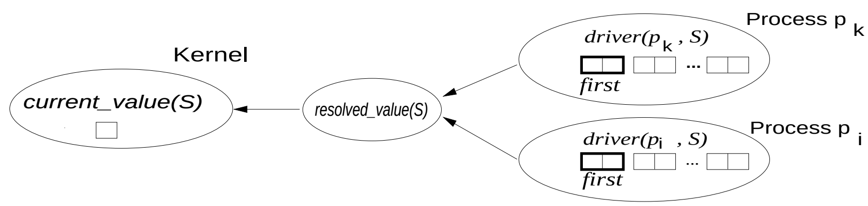

Figure 2 Non–Hierarchical Signal Updates

When considering hierarchical connections defined by port associations, signal values have to be propagated through a so–called *net*.[5] A net is a value propagation graph for propagating the driving and the effective values of signals through the design hierarchy represented by embedded blocks (see also Figure 7). The root of a net is a signal whose value has to be determined. Inner nodes are either associated with a signal or a resolution function; the leaves are given by drivers. The edges represent the signal–source relationships where a type conversion function may be assigned to each edge [22].

When computing new driving and new effective values, only active signals are considered. A signal in a net is active if one of the following holds:[6] (i) at least one active driver is reachable from that signal, (ii) it is a connected port (there exists an actual part given by a port association element) and its actual part is active, (iii) if one of its subelements is active in the case that the signal is of composite type. More formally,

$$
\begin{aligned}
&active(S) :\Leftrightarrow \\
&\quad (\exists d \in D(S) :\ active(d)) \text{ or } (actual(S) \neq undef \wedge active(actual(S))) \text{ or} \\
&\quad (\exists i \in subelements(S) : active(i)),
\end{aligned}
$$

$$
\text{where } D(S) = \{d \mid d \in drivers(S)\} \bigcup_{S' \in sources(S)} D(S')
$$

[5] VHDL'93 LRM, Page 166
[6] VHDL'93 LRM, Page 164

identifies the set of drivers within the subtree of signal S which are reachable from S.

Variables

Alternative means for interprocess communication are provided by shared variables in VHDL'93. The value $value(SV)$ of a shared variable SV can be updated from multiple concurrently running processes. Since no concept for the resolution of possibly conflicting write accesses is prescribed by VHDL'93 this introduces explicit non-determinism into the language. In order to distinguish shared variables from local variables we set $kind(V) \in \{local, shared\}$ for each $V \in VARIABLE$. For a local variable LV we write $value(P, LV)$ associating it with the process P it is declared in. In contrast to signal assignment, a *variable assignment* is immediately executed by a user defined process. Consequently, no timing model is associated with variables.

4.2 User Defined Processes

The rules P1-P6 in this section constitute the program of an agent, one for each user defined process P, and define the semantics of *variable assignment*, *signal assignment*, and of the various *wait statements*.

Processing Statements

In order to concentrate on the essential behavioral semantics of VHDL'93, we assume that the control flow of each (sequential) iterative process is determined by the environment which provides the dynamic changes of values for the external function[7] *program_counter*. The *program_counter* of each process is initialized by pointing to the first statement of that process. After having processed the last statement it returns to the first statement again.

In order to express that a user defined process P can be executed only when it is not suspended and when all processes of the same type as P are enabled to execute, we use the following abbreviation:

[7] An external function in the sense of [17] is a function which is not updated by the rules of the system under consideration; nevertheless such a function might be updated by the environment and thus represents a precise interface for the system.

$$
\begin{array}{l}
Process\ does\ \texttt{statement} \equiv \\
\quad program\_counter(Process) = \texttt{statement} \wedge suspended(Process) = false \wedge \\
\quad ((type(Process) = postponed \wedge phase = execute\_postponed) \vee \\
\quad\ (type(Process) = nonpostponed \wedge phase = execute\_nonpostponed))
\end{array}
$$

Recall that *phase* can assume sequential states of the kernel as well as *execute_nonpostponed* and *execute_postponed* indicating that the agents of that type are the ones currently executed.

Variable Assignments

The semantics of variable assignment is given by the transition rule P1.[8] If the target variable V is of kind local ($\neq$ *shared*), the $value(Process, V)$ of that variable V declared in *Process* is given by the *value* computed from the right-hand side expression *Expr*.

P1: SHARED/LOCAL VARIABLE ASSIGNMENT

$$
\begin{array}{l}
\textbf{if } Process\ does\ \langle V\ :=\ Expr\rangle \\
\textbf{thenif } kind(V) \neq shared \\
\quad \textbf{then } value(Process, V) :=\ value(Expr) \\
\quad \textbf{else }\ value(V) := resolve(competingValues(V))
\end{array}
$$

In the case of shared variables[9] *resolve* denotes the implementation defined resolution[10] of the concurrent update requests to $value(V)$ of variable V. *competingValues(V)* denotes the set of values competing for the update of V, i.e., all values $value(Expr_p)$ for each right-hand side expression $Expr_p$ of all processes p which are currently performing an assignment to variable V. Note, that for composite variables the VHDL'93 standard permits a possibly interleaving update of the subelements of a shared variable. That means, that for composite variables $competingValues(S)$ is applied subelement-wise and *resolve* is defined on the competing values of the subelements, i.e.,

$$resolve(V_1, ..., V_r) = resolve(V_1), ..., resolve(V_r).$$

[8] In order to concentrate on the relevant parts of behavioral semantics of VHDL statements we use abstract syntax by template-like descriptions enclosed by $\langle\rangle$.

[9] VHDL'93 LRM, §4.3.1.3

[10] Since shared variables introduce non–determinism there will be different implementations of VHDL'93 resolving this non–determinism.

Signal Assignments

The intuitive meaning of a transport delay instruction[11] $\langle S \Leftarrow \underline{\text{TRANSPORT}}\ Expr_1\ \underline{\text{AFTER}}\ Time_1, ...\rangle$[12] when carried out by some process P is to schedule, on the driver identified through *driver(P,S)* of process P of signal S, for each $1 \leq i \leq n$, the (possibly new) value $X_i = value(Expr_i)$ for the time point $T_c + Time_i$.[13] Note that due to the discrete VHDL time model, the sequence of time points $Time_i$ is required to be strictly increasing. For signal assignment the VHDL standard defines a preemptive scheduling. That is, all old values which were scheduled for time points $\geq T_c + Time_1$ are deleted. We describe this using a function $|_<$: $DRIVER \times TIME \rightarrow DRIVER$, which for given driver d and time t yields the driver containing precisely those transactions in d which have time component $< t$.

In the special case, when a new value is scheduled for the current time T_c, i.e., when $Time_1 = 0$, this means that the whole driver is replaced by the list $\langle (X_1, Time'_1), \ldots, (X_n, Time'_n)\rangle$ where $Time'_i = Time_i + T_c$ denotes the absolute time with respect to the current simulation time T_c. Since this means that the first transaction is replaced, the driver is set to be *active* then.

In the other case, when $Time_1 > 0$, the waveform, which by definition is linearly ordered, is simply appended to the previously shrunken driver. For the concatenation of sublists we use the ^-operator.

<u>P2: TRANSPORT DELAY</u>

if $Process$
  $does\ \langle S \Leftarrow \underline{\text{TRANSPORT}}\ Expr_1\ \underline{\text{AFTER}}\ Time_1, \ldots, Expr_n\ \underline{\text{AFTER}}\ Time_n\rangle$
thenif $Time_1 = 0$
  then $driver(Process, S) := Waveform$
    $active(driver(Process, S)) := true$
  else $driver(Process, S) := (driver(Process, S)\ |_< T_c + Time_1)\hat{}\ Waveform$
where $Waveform = \langle (X_1, Time'_1), \ldots, (X_n, Time'_n)\rangle \wedge Time'_j = Time_j + T_c \wedge$
  $X_j = value(Expr_j)$

The inertial delay[14] $\langle S \Leftarrow \underline{\text{INERTIAL}}\ Expr_1\ \underline{\text{AFTER}}\ Time_1, ...\rangle$ has the same effect as a transport delay, in the case that a new value is scheduled for the

[11] VHDL'93 LRM, Page 117
[12] $Time_i$ are placeholders for the time values computed from the corresponding time expressions.
[13] Within our model time components of transactions and timeouts represent the absolute time.
[14] VHDL'93 LRM, Page 117

current time, i.e., if $Time_1 = 0$. In other cases, all new values are scheduled for time points following the time point of the first transaction; as in the transport delay the new waveform $Waveform$ is appended. In addition to the transport delay's behavior, which realizes a preemption for scheduled transactions $\geq Time'_1$, the inertial delay manipulates the driver for elements with $time < Time'_1$. The VHDL'87 LRM defines this by a further 3 step algorithm in terms of marking elements and removing the unmarked elements thereafter.[15] This algorithm defines that the first element of the driver is kept (Step 3). Step 2 defines that a transaction is kept if

> "... it immediately precedes an unmarked transaction and its value component is the same as that of the marked transaction."

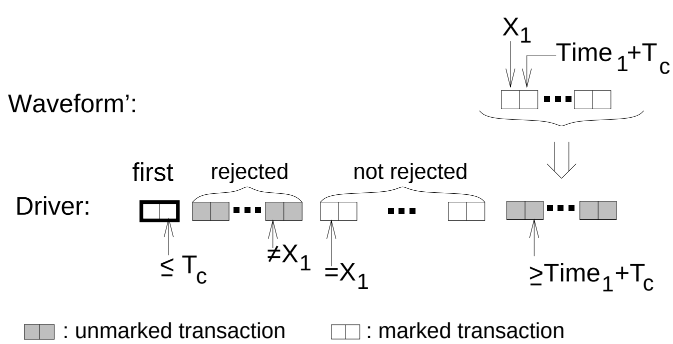

Figure 3 Preemptive scheduling for inertial delay

In our model this is realized by the function *reject*, which only keeps transactions whose value is equal to the value of the first new transaction ($X_1 = value(Expr_1)$), i.e., it rejects transactions with $value \neq X_1$. The resulting driver is obtained as a composition of three separate lists: the first element, the not rejected rest restricted by $Time_1 + T_c$, and the new transactions. This directly corresponds to the last update in the following rule.

[15] VHDL'87 LRM, Page 8–5

P3: INERTIAL DELAY

if $Process$
$\quad does\ \langle S \Leftarrow \underline{\text{INERTIAL}}\ Expr_1\ \underline{\text{AFTER}}\ Time_1, \ldots, Expr_n\ \underline{\text{AFTER}}\ Time_n \rangle$
thenif $Time_1 = 0$
then $driver(Process, S) := Waveform$
$\quad active(driver(Process, S)) := true$
else $driver(Process, S) :=$
$\quad first(driver(Process, S))\hat{\ }reject(driver', X_1)\hat{\ }Waveform$

where $Waveform = \langle (X_1, Time'_1), \ldots, (X_n, Time'_n) \rangle \wedge Time'_j = Time_j + T_c \wedge$
$X_j = value(Expr_j) \wedge driver' = tail(driver(Process, S) \mid_< (Time_1 + T_c))$

The function $reject$ is specified by:

$reject(TransList, Val) \equiv$
if $TransList = \langle\rangle \vee value(last(TransList)) \neq Val$
then return $\langle\rangle$
else return $reject(front(TransList, Val))\hat{\ }last(TransList)$

We have a similar rule for the refined inertial signal assignment statement appearing in VHDL'93:[16]

$$\langle S \Leftarrow \underline{\text{REJECT}}\ Pulse\ \underline{\text{INERTIAL}}\ Expr_1\ \underline{\text{AFTER}}\ Time_1, \ldots \rangle.$$

By the use of this statement it is possible to define an explicit pulse rejection limit *Pulse*, which may be different from the limit given by the first waveform element. This means that compared to the above rule the function *reject* is not applied to the whole tail of the restricted driver but, instead, some front transactions may be kept. Transactions starting from the first element of a driver up to $(Time_1 - Pulse) + T_c$ are not rejected. For $Pulse = Time_1$, the behavior of the statement is the same as a transport signal assignment. When $Pulse = 0$ the statement is equivalent to the VHDL'87 inertial signal assignment. Formally this is reflected in our model by the new driver being composed of the first transaction, followed by those whose time component is less that the rejection pulse, followed by those which are filtered by *reject*, followed by the new waveform elements; formally:

$driver(Process, S) :=$
$first(driver(Process, S))\hat{\ }driver''\hat{\ }reject(driver''', X_1)\hat{\ }Waveform,$

[16] VHDL'93 LRM, Page 117

where $driver'' = (driver' \mid_{<} ((Time_1 - Pulse) + T_c))$ and $driver'''$ denotes the driver elements of $driver'$ without the elements of $driver''$.

Wait Statements

The rules for wait statements define how processes are suspended due to wait requirements for a specified time period, a signal, or the truth of a condition.

For modeling WAIT FOR statements we use the concept of timeouts. Timeouts are set when a WAIT FOR statement is executed. They are reset by the kernel if the process is resumed. That is, if $Process$ WAITs FOR $Time$,[17] then $timeout(Process)$ is set to $T_c + Time$ and $Process$ is suspended by $suspended(Process) := true$.

<u>P4: WAIT FOR</u>

$$\begin{array}{l}\textbf{if } Process\ does\ \langle \underline{\text{WAIT}}\ \ \underline{\text{FOR}}\ Time \rangle \\ \textbf{then } timeout(Process) := Time + T_c \\ \qquad suspended(Process) := true\end{array}$$

If a Process WAITs ON a set of *Signals* or UNTIL an *Expression* becomes true, then the $Process$ becomes suspended and added to the set of processes which are waiting for changes of a signal in the sensitivity set. Each signal holds in $waiting(S)$ the set of processes which are sensitive to the signal value change (see also Figure 4).

<u>P5: WAIT ON</u>

$$\begin{array}{l}\textbf{if } Process\ does\ \langle \underline{\text{WAIT}}\ \ \underline{\text{ON}}\ Signals \rangle \\ \textbf{then} \\ \quad suspended(Process) := true \\ \quad \textbf{if } S \in Signals \\ \quad \textbf{then } waiting(S) := \\ \qquad\qquad waiting(S) \cup \{Process\}\end{array}$$

<u>P6: WAIT UNTIL</u>

$$\begin{array}{l}\textbf{if } Process\ does\ \langle \underline{\text{WAIT}}\ \ \underline{\text{UNTIL}}\ Expr \rangle \\ \textbf{then} \\ \quad waitcond(Process) := Expr \\ \quad suspended(Process) := true \\ \quad \textbf{if } S \in condsignals(Expr) \\ \quad \textbf{then } waiting(S) := \\ \qquad\qquad waiting(S) \cup \{Process\}\end{array}$$

If a Process WAITs UNTIL an expression $Expr$ it is resumed when the expression evaluates to true. The expression (current wait condition) is stored in *waitcond(Process)*. *waitcond(Process)* is set to *undef* when the kernel resumes the individual $Process$ suspended on that condition. The evaluation of the current waiting condition is performed by the kernel if at least one signal in this

[17] $Time$ is a placeholder for the value computed from a time expression.

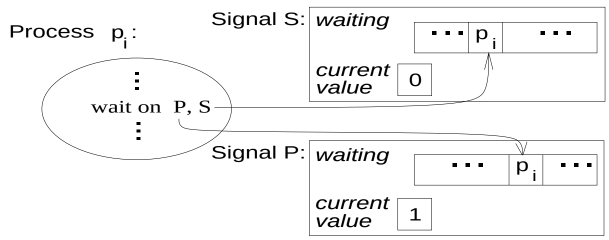

Figure 4 Processes Suspend on Signals

expression changes. The signals representing the sensitivity set are extracted from the expression by $condsignals(Expr)$. Again, the $Process$ is suspended and added to the set of processes of each signal in the sensitivity set. Note that if the function $condsignals$ returns an empty list the process suspends forever.

Finally, the special case of a wait statement without a clause which suspends the process for the rest of the simulation is defined by the following rule.

P7: WAIT FOREVER

$$\begin{array}{l} \textbf{if } Process\ does\ \langle\underline{\text{WAIT}}\rangle \\ \textbf{then } suspended(Process) := true \end{array}$$

4.3 The Kernel Process

The kernel is an agent the execution of which is enabled as soon as all user defined processes are suspended (see also Figure 1 and [26]). We abbreviate this by:

$$\begin{array}{l} AllProcessesSuspended \equiv \text{ either (i) or (ii) holds :} \\ \quad \text{(i) } phase = execute\_nonpostponed \wedge \forall P \in PROCESS : \\ \quad\quad \text{if } type(P) = nonpostponed \text{ then } suspended(P) = true \\ \quad \text{(ii) } phase = execute\_postponed \wedge \forall P \in PROCESS : \\ \quad\quad \text{if } type(P) = postponed \text{ then } suspended(P) = true \end{array}$$

Kernel actions are defined by the rules K1-K3. In these rules, the kernel first determines the next time point T_n. With respect to T_n and the current $cycle \in \{delta\_cycle, postponed\_cycle, time\_cycle\}$ the kernel sets $phase$ and next $cycle$. Recall, that if $cycle = postponed\_cycle$ the kernel executes the already resumed postponed processes by setting phase to $execute\_postponed$. If $cycle \in \{delta\_cycle, time\_cycle\}$, the kernel runs sequentially through the further phases as given in Figure 5: update driving values, update effective values, update current values, resume processes.[18]

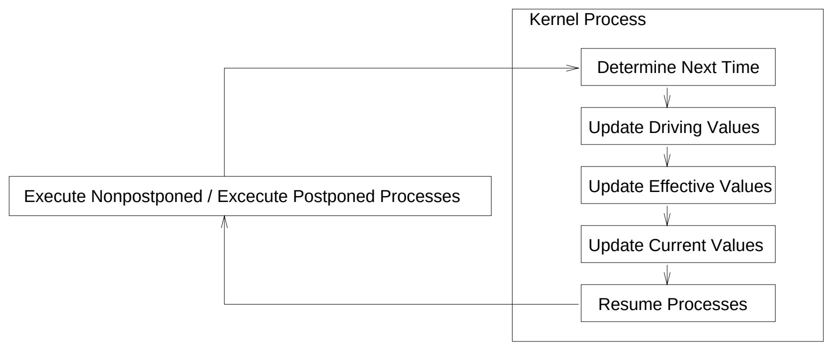

Figure 5 Different Phases of the VHDL Simulator

Determine Next Time Point

If the expected next time T_n is equal to the current time T_c[19] the kernel goes into cycle $delta\_cycle$, i.e., the next cycle will be a delta cycle. T_n is computed by taking the minimum of all timeouts $\geq T_c$, the time of all current transactions of active drivers, and the future time points of inactive drivers. Otherwise, if $T_n > T_c$, the kernel goes either from $delta\_cycle$ to $postponed\_cycle$ or from the latter to $time\_cycle$ (see Figure 6). Note that for the transition from $postponed\_cycle$ to $time\_cycle$ the condition $T_n > T_c$ still holds. "It is an error if the execution of any postponed process causes a delta cycle to occur immediately after the current simulation cycle"(VHDL'93 LRM, Page 169).

In the case of a $postponed\_cycle$, postponed processes (which have already been resumed) are executed by setting $phase := execute\_postponed$. In the case of a $time\_cycle$, also the drivers are updated with respect to the new time T_n. This

[18] VHDL'93 LRM, §12.6.4

[19] This is the case if there are some active drivers or if at least one of the processes has been suspended by $\langle \underline{\text{WAIT}}\ \underline{\text{FOR}}\ 0\,ns \rangle$. The latter one is a trick to enforce a synchronization with the kernel process.

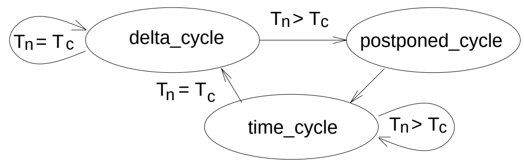

Figure 6 Different Cycles of the VHDL Simulator

causes at least one driver to become active. Additionally, T_c is advanced to T_n. In the case that the new time exceeds the simulation time limit $TIME'HIGH$, further execution is stopped by setting $phase := undef$ (see also Figure 1).

<u>K1: DETERMINE NEXT TIME POINT</u>

if $AllProcessesSuspended$
thenif $T_n = T_c$
  then $cycle := delta\_cycle$
    $phase := update\_driving\_values$
  elsif $cycle = delta\_cycle$
    then $cycle := postponed\_cycle$
      $phase := execute\_postponed$
    else $cycle := time\_cycle$
      $phase := update\_driving\_values$
      $AdvanceTime$
      $UpdateDrivers(T_n)$
where $T_n = min\{mindriver,\ mintimeout\}$

$AdvanceTime \equiv$ **if** $T_n \leq TIME'HIGH$ **then** $T_c := T_n$ **else** $phase := undef$

T_n is computed by taking the minimum of all timeouts $\geq T_c$ (*mintimeout*) if being defined and of times of all drivers (*mindriver*). In the case of active drivers, the time of the newly scheduled first element has to be considered. In the case of inactive drivers, the time of the second element has to be considered.

$$\begin{aligned} mintimeout &= min\{timeout(p)\ | p \in PROCESS \wedge timeout(p) \neq undef \wedge \\ &\quad timeout(p) \geq T_c\} \\ mindriver &= min\{time(t) \mid \exists d \in DRIVER : t = t^i, active(d) = i\}, \\ &\quad where\ t^{true} = first(d)\ and\ t^{false} = second(d). \end{aligned}$$

UpdateDrivers is applied in the case of a *time_cycle* in order to update all drivers with respect to the new time T_n. If any transaction is scheduled in any driver for the new T_n these drivers are updated to their tails, i.e., the first element is removed. These drivers become active by definition.[20] Due to the ordering of drivers, we can determine these drivers by comparing their second element with T_n.

$$
\begin{array}{l}
UpdateDrivers(Time) \equiv \\
\quad \mathbf{if}\ d \in DRIVER \wedge tail(d) \neq \langle\rangle \wedge time(second(d)) \ = \ Time \\
\quad \mathbf{then}\ d := tail(d) \\
\qquad\quad active(d) := true
\end{array}
$$

Propagation of Signal Values

When in *time_cycle* or in *delta_cycle*, the kernel process switches in sequence to the subphases for evaluating the driving values, the effective values, and the current values. This is called the propagation of signal values (see also VHDL'93 LRM, §12.6.2). The values of the signals have to be propagated through the design hierarchy by computing their different values (see also the outlines on Page 11). Figure 7 gives an example of a net and the corresponding VHDL program.

We specify the sequential states for propagating the signal values by the rules K2a–K2c. Each of these rules determines the value only for those signals which have become *active* during the current simulation cycle (for the definition of *active* see Page 12). The kernel switches to the next state by setting *phase* in each of these rules.

First we consider the computation of driving values.

K2a: UPDATE DRIVING VALUES

$$
\begin{array}{l}
\mathbf{if}\ cycle \in \{delta\_cycle, time\_cycle\} \wedge phase = update\_driving\_values \\
\mathbf{then}\ SetDrivingValues \\
\qquad phase := update\_effective\_values
\end{array}
$$

When computing the driving value *driving_value*(S) of each active signal S, ports of *mode in* are ignored since for an in port a driving value is not defined. This explains the following definition:

[20] VHDL'93 LRM, Page 164

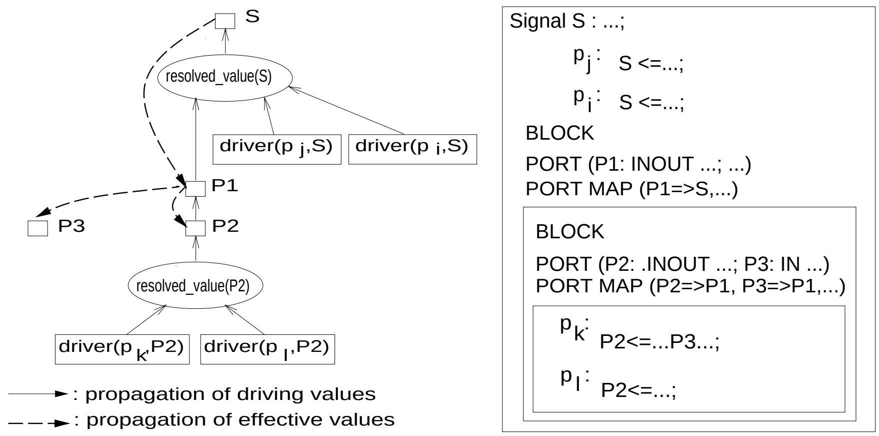

Figure 7 Signal Propagation Net Example

$$
\begin{array}{l}
SetDrivingValues \equiv \\
\quad \textbf{if } S \in SIGNAL \wedge active(S) = true \wedge mode(S) \neq inPort \\
\quad \textbf{then } driving\_value(S) := dv(S)
\end{array}
$$

$$
where\ dv(s) = \begin{cases} resolved\_value(dv(s_1'), ..., dv(s_n')), & \text{if } resolved(s) \wedge \\ & \quad sources(s) \text{ is defined} \\ dv(s_1'), & \text{if } not\ resolved(s) \wedge \\ & \quad sources(s) \text{ is defined} \\ value(s), & if\ s \text{ is a driver} \\ default\_value(s), & \text{otherwise} \end{cases}
$$

dv is a recursive definition on the signal sources, where $s_1', ..., s_n' = sources(s)$, $n \geq 1$, and $value(s)$ is the value of the first component for driver s. Note, that the second case in the definition of dv defines the value propagation for a signal s with only one source. This case also covers the propagation for inout, buffer, and out ports propagating the value from the formal to the actual part of the particular port association element.

After computing the driving value the effective values of the active signals have to be determined.

<u>K2b: UPDATE EFFECTIVE VALUES</u>
if $cycle \in \{delta\_cycle, time\_cycle\} \wedge phase = update\_effective\_values$
then $SetEffectiveValues$
$phase := update\_current\_values$

When determining the effective value $effective\_value(S)$ of each active signal S, ports of *mode out* are ignored since for an out port an effective value is not defined. This explains the following definition:

$SetEffectiveValues \equiv$
if $S \in SIGNAL \wedge active(S) = true \wedge mode(S) \neq outPort$
then $effective\_value(S) := ev(S)$

$$where\ ev(s) = \begin{cases} ev(actual(s)), & \text{if } mode(s) = inPort \vee mode(s) = inoutPort \\ driving\_value(s), & \text{if } declaredSignal(s) \vee mode(s) = buffer \vee \\ & \quad mode(s) = unconnected\_inoutPort \\ default(s), & \text{if } mode(s) = unconnected\_inPort \end{cases}$$

ev is defined by a recursion on port association elements from ports to signals. Recall, that $declaredSignal(s) = true$ denotes that s is an explicitly declared signal. Otherwise s is a port.

Finally, the current values are computed from the effective values.

<u>K2c: UPDATE CURRENT VALUES</u>
if $cycle \in \{delta\_cycle, time\_cycle\} \wedge phase = update\_current\_values$
then $SetCurrentValues$
$phase := resume\_processes$

The current value is determined for all active signals except for ports of mode out. If the newly determined effective value is different from the current value then the current value is updated by this value. This sets an event on the updated signal which is expressed by $event(S) := true$.

$$
\begin{array}{l}
SetCurrentValues \equiv \\
\qquad \textbf{if } S \in SIGNAL \wedge active(S) = true \wedge mode(S) \neq outPort \\
\qquad\quad \wedge\ effective\_value(S) \neq current\_value(S) \\
\qquad \textbf{then } current\_value(S) := effective\_value(S) \\
\qquad\qquad event(S) := true \\
\qquad\qquad SetEventTrueAttributes(S) \\
\qquad \textbf{else } \ SetEventFalseAttributes(S)
\end{array}
$$

Note that, as an example, we sketch the update of the corresponding predefined attributes of signal S without giving a further specification. In the case of an event $SetEventTrueAttributes$ is applied and $SetEventFalseAttributes$ otherwise.

Resume Processes

After phase *update_current_values*, the kernel continues in phase *resume_processes* in order to resume processes on signals and on expired timeouts. After that, the resumed nonpostponed processes are executed by setting *phase* to *execute_nonpostponed*.

K3: RESUME AND EXECUTE PROCESSES

$$
\begin{array}{l}
\textbf{if } cycle \in \{delta\_cycle, time\_cycle\} \wedge phase = resume\_processes \\
\textbf{then } ResetActiveDrivers \\
\qquad ResumeOnTimeouts \\
\qquad ResumeOnSignals \\
\qquad phase := execute\_nonpostponed
\end{array}
$$

The active drivers are first reset to their initial values since their being active holds only for one simulation cycle.

$$
ResetActiveDrivers \equiv \ \textbf{if } d \in DRIVER \textbf{ then } active(d) := false
$$

$ResumeOnTimeouts$ resumes all *waiting* processes whose timeout is set and equals the new T_c. The timeout is finally reset to $undef$.

$ResumeOnTimeouts \equiv$
if $Process \in PROCESS \wedge timeout(Process) \neq undef \wedge timeout(Process) = T_c$
then $suspended(Process)\ :=\ false$
$\quad timeout(Process)\ :=\ undef$

ResumeOnSignals resumes the processes which are sensitive to signals S with $event(S) := true$, i.e., all processes in $waiting(S)$ are resumed for each signal S in the case of an event on S. In case of suspension by WAIT UNTIL, i.e., if *waitcond* is defined, the corresponding condition *condvalue* has to be evaluated. Note that, when applying this function, each appearance of each signal S now refers to the previously updated current value of S. Finally, *event* and *waitcond* have to be initialized for the next simulation cycle and the individual resumed *Process* is deleted from the set of *waiting* processes.

$ResumeOnSignals \equiv$
if $S \in SIGNAL \wedge event(S) = true$
then $event(S) := false$
$\quad$ **if** $Process \in waiting(S)$
$\quad$ **thenif** $waitcond(Process) =\ undef$
$\quad\quad$ **then** $suspended(Process) := false$
$\quad\quad\quad waiting(S) := waiting(S) \backslash \{Process\}$
$\quad\quad$ **elsif** $value(waitcond(Process)) = true$
$\quad\quad\quad$ **then** $waitcond(Process) :=\ undef$
$\quad\quad\quad\quad suspended(Process) := false$
$\quad\quad\quad\quad waiting(S) := waiting(S) \backslash \{Process\}$

5 EXAMPLE

This section gives a detailed insight into the basic concepts of the IEEE standard VHDL simulator by simulating the VHDL program introduced in Appendix B. We outline the definition of our previously defined VHDL'93 simulator by running the first 11 simulation cycles which advance the simulation time to 23 nanoseconds.

Since our definition presumes elaborated VHDL we run the simulation on the elaborated model in Appendix A of the VHDL program in Appendix B.[21] By

[21] This example was given by the editor in [4].

this elaboration, component instantiations are transformed into hierarchical blocks, concurrent signal assignments are transformed into process statements, and positional port associations are transformed into named port associations. Additionally, each block and process is given a label. Since the port's identifiers are not unique within the model we prefix these identifiers with the label of the block they are defined in, e.g., the out port Y of the block labeled *example* is referred to as *example.Y*.

For our simulation we introduce a further process p_5 and a signal for the stimuli. The signal *stimuli* implements a clock with a half-period of 5 nanoseconds. Table 1 shows the values of the signals *stimuli*, $S(0..2)$, and *example.Y* for each nanosecond time point between 0 *ns* and 23 *ns*. The values of this table are partitioned with respect to the 5 nanosecond half-period of signal *stimuli*.

Signal \ *Time*	0 *ns*	5 *ns*	10 *ns*	15 *ns*	20 *ns*
stimuli	0,0,0,0,0	1,1,1,1,1	0,0,0,0,0	1,1,1,1,1	0,0,0,0,...
$S(0)$	0,0,0,0,0	0,0,0,0,0	0,1,1,1,1	1,1,1,1,1	1,0,0,0,...
$S(1)$	0,0,0,0,0	0,0,0,0,0	0,0,0,0,0	0,0,0,0,0	0,0,1,1,...
$S(2)$	0,0,0,0,0	0,0,0,0,0	0,0,0,0,0	0,0,0,0,0	0,0,0,0,...
example.Y	0,0,0,0,0	0,0,0,0,0	0,0,1,1,1	1,1,1,1,1	1,1,0,2,...

Table 1

The net representation of the elaborated model is illustrated in Figure 8. This figure shows three nets with the signals *stimuli*, S, and *example.Y* at their roots.

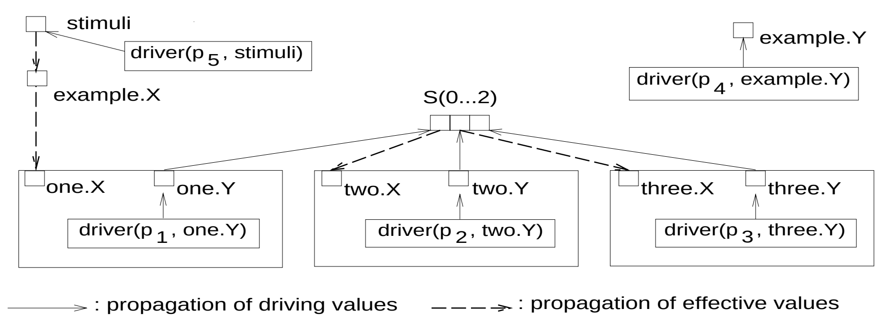

Figure 8 Value Propagation Nets

Initialization

For initialization we set the time–scale to nanoseconds and the current time T_c to 0 ns. All functions are set to $undef$ and all sets are assumed to be empty sets unless stated otherwise. $active(d)$ and $event(s)$ are set to $false$ for all $d \in DRIVER$ and $s \in SIGNAL$. $cycle$ is initially set to $delta\_cycle$. The first transaction of each driver is initialized by the current time (0 ns) and by the value 0: $(0 \mid 0ns)$. The current value of each (subelement of each) signal is initialized by 0.

By definition of the program all processes are nonpostponed processes, i.e., $type(p) = nonpostponed$ for all $p \in PROCESS$. Finally, these processes are executed by setting $phase = execute\_nonpostponed$. This enables p_5 to execute a signal assignment which fires Rule P3 updating the driver of $stimuli$ which becomes $active$. Thereafter $driver(p_5, stimuli)$ holds the following waveform elements:

$$(0 \mid 0ns)\ (1 \mid 5ns)\ (0 \mid 10ns)\ (1 \mid 15ns)\ (0 \mid 20ns)\ (1 \mid 25ns)\ \ldots$$

Furthermore, p_4 schedules the value 0 to $driver(p_4, example.Y)$ at 1 ns when firing Rule P3. We obtain:

$$driver(p_4, example.Y) = (0 \mid 0ns)(0 \mid 1ns)$$

All other processes are suspended when executing their first statement. The processes are suspended with the configuration given in Table 2. The leftmost column of this table shows the location of the program counters. The rightmost column enumerates the sensitivity sets of the individual processes.

$Process \setminus Function$	$program\_counter$	$waitcond$	$condsignals$
p_1	⟨ wait until X = '0' ⟩	one.X = '0'	$\{one.X\}$
p_2	⟨ wait until X = '0' ⟩	two.X = '0'	$\{two.X\}$
p_3	⟨ wait until X = '0' ⟩	three.X = '0'	$\{three.X\}$
p_4	⟨ wait on S ⟩	$undef$	$\{S\}$
p_5	⟨ wait ⟩	$undef$	$\{\}$

Table 2

As an additional result, the set $waiting(s)$ of each signal s holds the processes which are sensitive to events on s. The signals with a non–empty $waiting$ set are listed in Table 3. Note that process p_5 has been suspended forever firing Rule P7 and thus is not sensitive to any timeout or signal change.

Function \Signal	*one.X*	*two.X*	*three.X*	*S*
waiting	$\{p_1\}$	$\{p_2\}$	$\{p_3\}$	$\{p_4\}$

Table 3

Cycle 1

Firing rule K1, the next simulation time T_n is computed. Since during initialization $driver(p_5, stimuli)$ has been set active the time minimum for T_n is given by the time component of its first transaction $(0 \mid 0ns)$, i.e., 0 ns. $cycle$ is set to *delta_cycle* and *phase* is set to *update_driving_values* since $T_c = T_n$. During the next phases the driving, effective, and current values are propagated through the net $driver(p_5, stimuli)$ is associated with firing the rules K2a–K2c. This net is given by Figure 9.

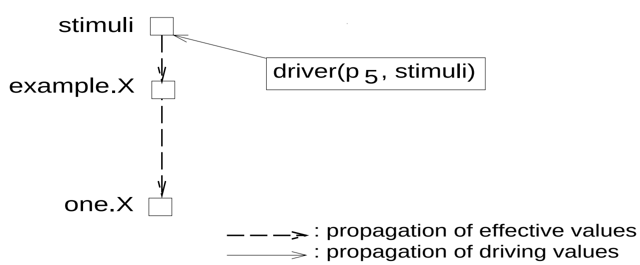

Figure 9 Value Propagation Net for *stimuli*

First, the driving value of *stimuli* is set to the value 0, i.e., to the value of $driver(p_5, stimuli)$. Thereafter, the effective value of *stimuli* is set to its driving value. The effective value is then propagated to *example.X* and *one.X* setting both to 0. Since the effective values of these signals all equal their current values no events are generated for those signals when firing Rule K2c. Rule K3 resets *active* of $driver(p_5, stimuli)$. No process is resumed since no event has occurred on any signal. Thus, when setting *phase* to *execute_nonpostponed* in Rule K3 the condition *AllProcessesSuspended* immediately evaluates to true which initiates the next simulation cycle.

Cycle 2

On the condition for *AllProcessesSuspended* the kernel becomes active firing Rule K1. The minimum time for T_n is given by the second element of $driver(p_4, example.Y)$: 1 *ns*. Consequently, *cycle* is set to *postponed_cycle* and *phase* is set to *execute_postponed*. Since by definition of the example there are no postponed processes K1 fires once more.[22] $T_n = 1\ ns$ remains greater than $T_c = 0\ ns$ and thus *cycle* is set to *time_cycle* and *phase* to *update_driving_values*. By *UpdateDrivers* with 1 *ns*, $driver(p_4, example.Y)$ becomes updated and thereby active:

$$driver(p_4, example.Y) = (0 \mid 1ns)$$

The current time is advanced to 1 *ns* in *AdvanceTime*. Thereafter, when firing the rules K2a-K2c, the driving, effective, and current values of all active signals are determined in sequence, i.e., the value of $driver(p_4, example.Y)$ is propagated in Rule K2a to the driving value of signal *example.Y*. Since *example.Y* is an out port neither an effective nor a current value is defined for this signal. For that reason and since a value change from 0 to 0 does not cause an event, no process is resumed when firing Rule K3. Consequently, the condition *AllProcessesSuspended* immediately evaluates to true.

Cycle 3

On the condition for *AllProcessesSuspended* the kernel becomes active firing Rule K1. The minimum time for T_n is given by 5 *ns*, i.e., the time component of the second element of $driver(p_5, stimuli)$. Consequently, *cycle* is set to *postponed_cycle* and *phase* is set to *execute_postponed*. This causes an immediate second firing of Rule K1 since by definition of the model there are no postponed processes. *cycle* is set to *time_cycle* and *phase* to *update_driving_values* in Rule K1. By *UpdateDrivers* with 5 *ns*, $driver(p_5, stimuli)$ becomes updated and thereby active. Thus, we obtain:

$$driver(p_5, stimuli) = (1 \mid 5ns)\ (0 \mid 10ns)\ (1 \mid 15ns)\ (0 \mid 20ns)\ (1 \mid 25ns)\ \ldots$$

The current time is advanced to 5 *ns* in *AdvanceTime*. The driving, effective, and current values of all active signals are determined when firing the rules K2a–K2c. Thereby the value of $driver(p_5, stimuli)$ is propagated through the corresponding net given by Figure 9. Table 4 presents the results after applying

[22] At the beginning of each cycle of the remaining simulation K1 fires twice. The first firing is an attempt to execute the (not existing) postponed processes.

the rules K2a–K2c. In this table we indicate the change of a value by an arrow, i.e., the current values of all of these signals change from 0 to 1 which sets an event for each of these signals.

$Value \backslash Signal$	$stimuli$	$example.X$	$one.X$
$driving$	1	-	-
$effective$	1	1	1
$current$	$0 \rightarrow 1$	$0 \rightarrow 1$	$0 \rightarrow 1$
$event$	$true$	$true$	$true$

Table 4

In $ResumeOnSignals$ of Rule K3, for each of the signals with an event the wait condition of the processes in $waiting$ (see Table 3) is checked by the kernel. p_1 is not resumed since the condition *waitcond(waiting(one.X))* evaluates to false ($one.X \neq$ $'0'$). Rule K3, in addition, resets $active$ and $event$ for the given signals and all drivers. Rule K1 immediately fires since $suspended$ remains true for all processes.

Cycle 4

In Rule K1, T_n is given by the second transaction of $driver(p_5, stimuli)$ evaluating to 10 ns. After the attempt to execute the resumed postponed processes K1 fires again and determines a time cycle advancing the current simulation time to $T_n = 10\ ns$. In $UpdateDrivers$ to 10 ns, $driver(p_5, stimuli)$ is updated and set to $active$. Thus, we obtain:

$$driver(p_5, stimuli) = (0 \mid 10ns)\ (1 \mid 15ns)\ (0 \mid 20ns)\ (1 \mid 25ns)\ ...$$

Again, all signals of the net $driver(p_5, stimuli)$ is associated with are updated. The result of this update is shown in Table 5.

The $waiting$ sets of all signals with an event are checked for processes to resume (see Table 3). In the current cycle, process p_1 resumes on signal $one.X$ since $waitcond(p_1)$ evaluates to true (see Table 2). The kernel sets $phase$ in order to execute the resumed nonpostponed processes in Rule K3. This enables process p_1 to execute the signal assignment $\langle$ *Y <= '1'* AFTER *1 ns*$\rangle$. The signal assignment fires Rule P3 scheduling one transaction at time $T_c + 1\ ns$:

$$driver(p_1, one.Y) = (0 \mid 0ns)\ (1 \mid 11ns)$$

$Value \backslash Signal$	$stimuli$	$example.X$	$one.X$
$driving$	0	-	-
$effective$	0	0	0
$current$	$1 \rightarrow 0$	$1 \rightarrow 0$	$1 \rightarrow 0$
$event$	$true$	$true$	$true$

Table 5

p_1 finally suspends with wait condition $one.X = '0'$ enabling the kernel process again.

Cycle 5

The new time T_n is given by the second element of $driver(p_1, one.Y)$, i.e., $11ns$. After a second firing of Rule K1 a time cycle is determined. When being updated to 11 ns, $driver(p_1, one.Y)$ becomes *active* and obtains a new value which is propagated through the subnet of $S(0)$ (see Figure 10).

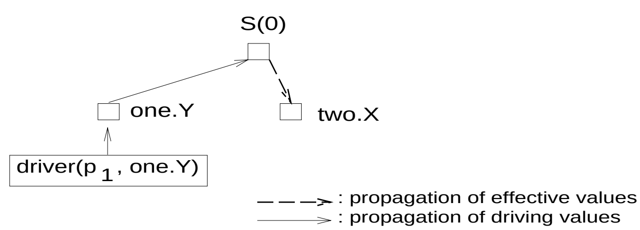

Figure 10 Value Propagation Subnet for $S(0)$

Table 6 shows the values after the update.

Process p_4 is resumed since there is an event on the subelement $S(0)$ of signal S and $p_4 \in waiting(S)$ (see Table 3). p_2 is not resumed since $two.X \neq '0'$.

When $phase = execute\_postponed$, p_4 performs the signal assignment statement $\langle$ $Y <= S$ AFTER *1 ns*$\rangle$. This assignment schedules the value 1 at time 12 ns to $driver(p_4, example.Y)$. Thereafter, the wait statement suspends p_4 on signal S.

$Value \backslash Signal$	$one.Y$	$S(0)$	$two.X$
$driving$	1	1	-
$effective$	-	1	1
$current$	-	0→1	0→1
$event$	$false$	$true$	$true$

Table 6

Cycle 6

Since all user defined processes are currently suspended Rule K1 fires twice.[23] The current simulation time is advanced to 12 ns which is given by the previously scheduled transaction of $driver(p_4, example.Y)$. Consequently, the driving value of $example.Y$ computes to 1.

Cycle 7 – Cycle 11

Cycle 7 performs equal to Cycle 3 advancing the current simulation time to 15 ns. Cycle 8 performs equal to Cycle 4 and sets the current time to 20 ns. As a result of the event on $one.X$, which equals 0, p_1 is resumed. The signal assignment schedules the value 0 at 21 ns to $driver(p_1, one.Y)$. Thus, in Cycle 9, T_c is set to 21 ns. The value of $driver(p_1, one.Y)$ is propagated to $S(0)$ and $two.X$ which both become 0. p_2 is resumed since an event is set for $two.X$ and the condition $two.X = '0'$ evaluates to true. The signal assignment performed by p_2 schedules the value 1 at 22 ns to $driver(p_2, two.Y)$. In Cycle 10 this value is propagated to $S(1)$ whose value change resumes process p_4. The signal assignment performed by p_4 schedules the value 2 to $driver(p_4, example.Y)$ at 23 ns. The propagation to the driving value of $example.Y$ finally takes place in Cycle 11. Thus, during the last three cycles, the (driving) value of the out port $example.Y$ increments from 1 at time 21 ns to 2 at time 23 ns with the intermediate value of 0 at time 22 ns (see also Table 1).

[23] Recall that the first firing is the attempt to execute the postponed processes.

6 CONCLUSION & FUTURE DIRECTIONS

We have given a formal definition of the VHDL'93 simulator which is introduced without any extraneous formal methodological overhead. We have strictly defined our model with regard to the nameing of the VHDL'93 language reference manual which makes the formalization of this manual directly visible to the reader. Our aim was to provide the community with a formal but yet human–processable model of the relevant behavioral constructs of VHDL'93.

The next investigations will be undertaken from this rigorous basis to build tools for machine assisted analysis and verification. Additional investigations in deriving an implementation of a VHDL simulator might be sensible since most of the syntactical representation of our EA–Machines should be easily implementable. Due to the inherent parallelism of EA-Machines a distributed implementation might be achieved.

Further work will concentrate on a specification of UDL/I [19] with the ultimate goal to close the formal gap between VHDL's simulation–oriented semantics and UDL/I's hardware–oriented semantics for language comparison. As a first step we have to investigate implicit signals within our VHDL specification.

Acknowledgements

We would like to thank Peter T. Breuer, Uschi Hudson, Yuri Gurevich, Leo Marcus, Christel Oczko, Serafin Olcoz, Franz J. Rammig, and Simon Read for their valuable comments when reviewing this work.

REFERENCES

[1] C. Bayol, B. Soulas, F. Corno, P. Prinetto, and D. Borrione. A process algebra interpretation of a verification oriented overlanguage of vhdl. In *EURO-DAC'94/EURO-VHDL'94*. IEEE CS Press, 1994.

[2] J.-M. Berge, A. Fonkoua, S. Maginot, and J. Rouillard. *VHDL'92*. Kluwer Academic Publishers, Dordrecht, Netherlands, 1993.

[3] E. Börger, U. Glässer, and W. Müller. The semantics of behavioral VHDL'93 descriptions. In *EURO-DAC'94/EURO-VHDL'94*. IEEE CS Press, 1994.

[4] E. Börger, U. Glässer, and W. Müller. Formal definition of an abstract VHDL'93 simulator by EA–Machines. In C. Delgado Kloos and P. T. Breuer, editors, *Formal Semantics For VHDL*. Kluwer, Boston/London/Dordrecht, 1995.

[5] D. Borrione, H. Eveking, and L. Pierre. Formal proofs from HDL descriptions. In J.P. Mermet, editor, *Fundamentals and Standards in Hardware Description Languages*. Kluwer Academic Publishers, Dordrecht, Netherlands, 1993.

[6] D. Borrione and J.L. Paillet. An approach to the formal verification of VHDL descriptions. Technical Report 683-I, IMAG/ARTEMIS, Grenoble, 1987.

[7] P.T. Breuer, L. Sanchez, and C. Delgado Kloos. Clean formal semantics for VHDL. In *European Design and Test Conference '94*, 1994.

[8] P.T. Breuer, L. Sanchez, and C. Delgado Kloos. Hard models for hardware: relating functional and denotational semantics for VHDL. In *PROCOMET '94*, 1994. Submitted preliminary version.

[9] P.T. Breuer, L. Sanchez Fernandez, and C. Delgado Kloos. Proof theory and a validation condition generator for VHDL. In *EURO-DAC'94/EURO-VHDL'94*. IEEE CS Press, 1994.

[10] T. Breuer, L. Sanchez Fernandez, and C. Delgado Kloos. A functional semantics for macro-time VHDL. In C. Delgado Kloos and P. T. Breuer, editors, *Formal Semantics For VHDL*. Kluwer, Boston/London/Dordrecht, 1995.

[11] W. Damm, B. Josko, and R. Schloer. A net-based semantics for VHDL. In *EURO-DAC'93/EURO-VHDL'93*. IEEE CS Press, 1993.

[12] W. Damm, B. Josko, and R. Schloer. Specification and verification of VHDL-based system-level hardware designs. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, Oxford, 1994. to appear.

[13] K.C. Davis. A denotational definition of the VHDL simulation kernel. In *Computer Hardware Description Languages and their Application*, Amsterdam, 1993. North-Holland.

[14] A. Debreil and Ph. Oddo. Synchronous designs in VHDL. In *EURO-DAC'93/EURO-VHDL'93*. IEEE CS Press, 1993.

[15] H. Eveking. (V)HDL-based verification of heterogeneous synchronous/asynchronous systems. In *EURO-DAC'94/EURO-VHDL'94*. IEEE CS Press, 1994.

[16] M. Fuchs and M. Mendler. Functional semantics for delta-delay VHDL based on FOCUS. In C. Delgado Kloos and P. T. Breuer, editors, *Formal Semantics For VHDL*. Kluwer, Boston/London/Dordrecht, 1995.

[17] Y. Gurevich. Evolving algebras – a tutorial introduction. In *Bulletin of the EATCS*, volume 43, pages 264–284. EATCS, 1991.

[18] Y. Gurevich. Evolving algebra 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, Oxford, 1994.

[19] Japan Electronic Industry Development Association. *UDL/I Language Reference Manual—Version 2.0.3*, September 16 1993. Translation from the Japanese Language Reference Manual.

[20] B. H. Levy, I. V. Filippenko, L. G. Marcus, and T. K. Menas. Using the state delta verification system (SDVS) for hardware verification. In *IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 337–360. North-Holland, 1992.

[21] L. G. Marcus and B. H. Levy. Specifying and proving core VHDL descriptions in the state delta verification system (SDVS). Technical Report ATR-89(4778)-5, The Aerospace Corporation, 1989.

[22] P.J. Menchini. VHDL. In J.P. Mermet, editor, *Fundamentals and Standards in Hardware Description Languages*. Kluwer Academic Publishers, Dordrecht, Netherlands, 1993.

[23] W. Mueller. Approaching the denotational semantics of behavioral VHDL92 descriptions. In *First Asian Pacific Conference on Hardware Description Languages, Standards & Applications*, 1993.

[24] S. Olcoz and J.M. Colom. Toward a Formal Semantics of IEEE Std. VHDL 1076. In *EURO-VHDL/EURO-DAC'93*. IEEE CS Press, 1993.

[25] S. Olcoz and J.M. Colom. VHDL through the looking glass. In *VHDL-FORUM*, 1993.

[26] S. Olcoz and J.M. Colom. The discrete event simulation semantics of VHDL. In *International Conference on Simulation and Hardware Description Languages*, San Diego, CA, 1994. SCSI.

[27] S. Read and M. Edwards. A formal semantics of VHDL in Boyer-Moore logic. In *2nd International Conference on Concurrent Engineering & EDA*, San Diego, CA, 1994. SCSI.

[28] R. Reetz and Th. Kropf. Simplifying deep embedding: A formalised code generator. In *International Workshop on High Order Logic Theorem Proving and its Applications*. Springer, 1994.

[29] R. Reetz and Th. Kropf. Correct system level design with VHDL. In C. Delgado Kloos and P. T. Breuer, editors, *Formal Semantics For VHDL*. Kluwer, Boston/London/Dordrecht, 1995.

[30] D.M. Russinoff. Specification and verification of gate-level VHDL models of synchronous and asynchronous circuits. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, Oxford, 1994.

[31] A. Salem. *Verification formelle des circuits digitaux decrits en VHDL*. PhD thesis, Universite Joseph Fourier, Grenoble, October 1992.

[32] A. Salem and D. Borrione. Formal semantics of VHDL timing constructs. In J.P. Mermet, editor, *VHDL for simulation, synthesis, and formal proof*. Kluwer Academic Publishers, London, 1993.

[33] The Institute of Electrical and Electronics Engineers, New York, NY, USA. *IEEE Standard VHDL Language Reference Manual–IEEE Std 1076-1987*, 1988. Order Code SH11957.

[34] The Institute of Electrical and Electronics Engineers, New York, NY, USA. *IEEE Standard VHDL Language Reference Manual–IEEE Std 1076-1993*, 1994. Order Code SH16840.

[35] J.P. van Tassel. *Femto-VHDL: The Semantics of a Subset of VHDL and its Embedding in the HOL Proof Assistant.* PhD thesis, University of Cambridge, July 1993.

[36] P. Van Tassel, J. An operational semantics for a subset of VHDL. In C. Delgado Kloos and P. T. Breuer, editors, *Formal Semantics For VHDL.* Kluwer, Boston/London/Dordrecht, 1995.

[37] E. Villar and P. Sanchez. Synthesis applications of VHDL. In J.P. Mermet, editor, *Fundamentals and Standards in Hardware Description Languages.* Kluwer Academic Publishers, Dordrecht, Netherlands, 1993.

[38] P.A. Wilsey. Developing a formal semantics description of VHDL. In *First European Conference on VHDL, Vol.2*, 1990.

APPENDIX A

ELABORATED EXAMPLE

```
p5:      process begin
           stimuli <= '0' after 0 ns, '1' after 5 ns, '0' after 10 ns,
                     '1' after 15 ns, '0' after 20 ns, '1' after 25 ns, ...;
           wait;
         end process;
example: block port (X: in bit; Y: out bit_vector(0 to 2));
              port map (X=>stimuli);
         signal S: bit_vector(0 to 2);
         begin
         p4:   process begin
                 Y <= S after 1 ns;
                 wait on S;
               end process;
         one:  block
               port((X: in bit; Y: out bit); port map(X=>X,Y=>S(0));
               begin
               p1: process begin
                     wait until X='0';
                     Y<='1' after 1 ns;
                     wait until X='0';
                     Y<='0' after 1 ns;
                   end process;
               end block one;
         two:  block
               port((X: in bit; Y: out bit); port map(X=>S(0),Y=>S(1));
               begin p2: process begin ... end process;
               end block two;
         three: block
               port((X: in bit; Y: out bit); port map(X=>S(1),Y=>S(2));
               begin p3: process begin ... end process;
               end block three;
         end block example;
```

APPENDIX B

VHDL EXAMPLE

```
entity cont_1 is
   port (X: in bit; Y: out bit);
end cont_1;

architecture beh of cont_1 is
begin
   process
      begin
         wait until X='0';
         Y <= '1' after 1 ns;
         wait until X='0';
         Y <= '0' after 1 ns;
      end process;
end beh;

entity cont_3 is
   port (X: in bit; Y: out bit_vector(0 to 2));
end;

architecture structural of cont_3 is
component cont_1
   port(X: in bit; Y: out bit);
end component;
for all: cont_1 use entity work.cont_1(beh);
signal S:bit_vector (0 to 2);
begin
   Y <=S after 1 ns;
   one:   cont_1 port map(X, S(0));
   two:   cont_1 port map(S(0), S(1));
   three: cont_1 port map(S(1), S(2));
end structural;
```