13
Characterizing a portable subset of behavioral VHDL-93

Krishnaprasad Thirunarayan
Dept. of Computer Science and Engineering
Wright State University, Dayton, OH-45435
email: *tkprasad@cs.wright.edu*
phone: *(937)-775-5109* fax: *(937)-775-5133*

Robert L. Ewing
Software-Hardware Technology Branch
Wright Laboratory (Avionics Division)
Wright Patterson AFB, Dayton, OH-45433.

Abstract

Goossens defined a structural operational semantics for a subset of VHDL-87 and proved that the parallelism present in VHDL is benign. We extend this work to include VHDL-93 features such as shared variables and postponed processes that change the underlying semantic model. In the presence of shared variables, nondeterministic execution of VHDL-93 processes destroys the unique meaning property. We identify and characterize a class of *portable* VHDL-93 descriptions for which unique meaning property can be salvaged. Our specification can serve as a correctness criteria for a VHDL-93 simulator.

Keywords

VHDL, formal specification, simulation, shared variables

1 INTRODUCTION

VHDL has been designed to facilitate specification, documentation, communication and formal manipulation of hardware designs at various levels of abstraction (Bhaskar 1994). The semantics of VHDL-93 are given in English prose in (IEEE 1993). The goal of developing formal semantics is to provide a complete and unambiguous specification of the language. Adherence to this standard will contribute significantly to the sharing, portability and integration of various applications and computer-aided design tools; to the implementation of language processors; and for formal reasoning about VHDL descriptions. Furthermore, this exercise enhances our understanding of the various VHDL-93 constructs/features.

There have been a number of proposals for a formal semantics of VHDL, almost all of them dealing with subsets of VHDL-87 (Breuer *et al.* 1995, Kloos *et al.* 1995, Goossens 1995, van Tassel 1993, Wilsey 1992). In particular, Goossens (Goossens 1995) defines a structural operational semantics (Hennessy 1990) for a subset of VHDL-87 that includes almost all the fundamental behavioral constructs in a single VHDL-87 entity. Börger *et al* (Chapter 4, (Kloos *et al.* 1995)) provide a formal definition of VHDL-93 features using EA-machines. However, they do not formally prove properties of their semantics.

In this paper we build on Goossens work which deals with a subset of behavioral VHDL-87. We define a structural operational semantics for a subset of behavioral VHDL-93 that includes features such as *shared variables* and *postponed processes*, not present in VHDL-87. These VHDL-93 constructs fundamentally change the underlying semantic model of VHDL. In particular, the unique meaning (monogenicity) property proved for the subset of VHDL-87 in (Goossens 1995) no longer holds in the presence of shared variables because of nondeterministic and asynchronous nature of process executions. However, we characterize a class of *portable* VHDL-93 descriptions for which the *unique meaning property* can be salvaged. That is, we specify VHDL-93 descriptions that will always yield the same results when interpreted by different simulators or by the same simulator on different runs. The goal is to provide an approximate but formal interpretation of the following statement in Section 4.3.1.3 in the VHDL LRM (IEEE 1993).

> A description is *erroneous* if it depends on whether or how an implementation sequentializes access to shared variables.

Our definition of portability is based on the following observations: In each simulation cycle, if only one process accesses a shared variable, then the final value of the shared variable is uniquely determined (because of the sequential execution of the statements in a process). Similarly, if a shared variable is only read by some/all processes, the value of the shared variable remains unchanged. However, if multiple processes try to access a shared variable while one of them is writing into it in the same cycle, there is potential for ambiguity in the final value of the shared variable.

Our formalization can be viewed as a specification for the VHDL-93 simulators against which the correctness of an implementation can be verified. It specifies additional run-time machinery that can potentially be incorporated in a VHDL-93 simulator to flag VHDL-93 descriptions that cannot be "safely" ported. In course of this development we also explain and correct a few errors that have crept into the formal description of the VHDL-87 semantics given in (Goossens 1995).

The rest of this paper is organized as follows: Section 2 presents the abstract syntax of the VHDL-93 subset and Section 4 specifies its semantics.

The primary emphasis is on the changes to the semantics in (Goossens 1995) resulting from the introduction of shared variables and postponed processes. We explore the causes of non-portability and then formally define what we mean by *portable VHDL-93 descriptions*. Section 3 illustrates the portability problem. In Section 5 we prove some interesting properties of portable descriptions. Section 6 presents some conclusions.

2 ABSTRACT SYNTAX OF VHDL-93 SUBSET

The abstract syntax of the core subset of behavioral VHDL-93 is shown below.*

- Syntactic Categories

pgm	$\in$	$Programs$	$proc$	$\in$	$Processes$
p	$\in$	$NonPostponedProcesses$	pp	$\in$	$PostponedProcesses$
ss	$\in$	$SequentialStatements(= SSt)$	e	$\in$	$Expressions(= Expr)$
s	$\in$	$Signals(= Sig)$	S	$\in$	$SetsOfSignals$
x	$\in$	$Variables(= Var)$	sx	$\in$	$SharedVariables(= SVa$
v	$\in$	$Values(= Val)$			

- Definitions

$$\begin{array}{lcl}
pgm & ::= & \|_{i\in I}\ proc_i \\
proc_i & ::= & p_i \mid \texttt{postponed}\ pp_i \\
p_i & ::= & \texttt{while}\ true\ \texttt{do}\ ss_i \\
pp_i & ::= & \texttt{while}\ true\ \texttt{do}\ ss_i \\
ss_i & ::= & \texttt{null} \mid x := e_i \mid sx_i := e_i \mid s <= e_i\ \texttt{after}\ e_i \\
 & & \mid ss_i\ ;\ ss_i \mid \texttt{wait on}\ S\ \texttt{for}\ e_i\ \texttt{until}\ e_i \\
 & & \mid \texttt{while}\ e_i\ \texttt{do}\ ss_i \mid \texttt{if}\ e_i\ \texttt{then}\ ss_i\ \texttt{else}\ ss_i \\
e_i & ::= & \texttt{null} \mid v \mid x \mid sx_i \mid s \\
 & & \mid e_i\ bop\ e_i \mid uop\ e_i \mid s'\texttt{delayed}(e_i)
\end{array}$$

A program in this VHDL-93 subset can be viewed as a *fully elaborated behavioral VHDL-93 description* (IEEE 1993). It is a collection of processes communicating with each other through signals and shared variables. $||$ is the parallel composition operator and I is a finite index set. As mentioned earlier, a VHDL-93 description is *portable* if one can associate a unique meaning with it. To characterize portable VHDL-93 descriptions, we associate the identity of a process with each occurrence of a shared variable. So we have tagged the

*The VHDL-93 concrete syntax for the process statement is: $(\texttt{while}\ true\ \texttt{do}\ ss_i) \equiv (i:\ \texttt{process begin}\ ss_i\ \texttt{end process}\ i;)$

meta-variables *proc*, *p*, *pp*, *ss*, and *e* with subscript *i* representing the index of the associated process $proc_i$.*

The set of processes has been partitioned into postponed processes (*pp*) and non-postponed processes (*p*). The predicate *postponed?* is true of all postponed-process indices. A process is a sequence of statements that can be executed repeatedly. The statements include assignments, wait statements, and control statements. In wait statements, whenever "**on** S", "`for` e", or "**`until`** e" are omitted, "**on** S_{ue}" (where S_{ue} is the set of signals in the **`until`** clause), "`for` ∞", or "**`until`** *true*" respectively are assumed. In signal assignments, whenever the `after`-clause is omitted, "`after` 0" is assumed. The expression syntax is standard and includes logical and arithmetic expressions.

With regards to the static semantics, we assume that the VHDL-93 descriptions are *well-typed*, and all the signals with multiple drivers have a suitable resolution function associated with them. For instance, the expression e in "`for` e" is assumed to be of integer type, while that in "**`until`** e" is of boolean type.

We now explore the semantic complications caused by the introduction of shared variables into VHDL.

3 THE CAUSES OF NON-PORTABILITY

Intuitively, a VHDL-93 description is *portable* if it assigns the same "observable" values to all (shared) variables and signals. The following examples illustrate the causes of non-portability and motivate restrictions required to guarantee portability of VHDL-93 descriptions. We assume that all variables/shared variables of integer type are initially 0.

Example 1. The following VHDL description is not portable as the value of sx after t-ns (> 0) can be either 1 or 2 (due to inherent nondeterminism).

```
while true do (sx := 1; wait for 1 ns;)
                    ||
while true do (sx := 2; wait for 1 ns;)
```

Example 2. Similarly, the following description is not portable as the value of z after t-ns can be either y or $y + 1$ (either t or $t + 1$).

```
while true do (y := y + 1; sx := y; wait for 1 ns;)
                    ||
while true do (z := sx; wait for 1 ns;)
```

Example 3. On the contrary, the following description is portable because, in each unit-time-interval, the shared variable is either only read simultaneously by both processes, or is accessed in read/write mode only by the second process. The value of sx after t-ns is $\lceil \frac{t}{2} \rceil$.

*Alternatively, this can be easily specified through the static semantics.

$$\texttt{while true do } (y := sx; \texttt{ wait for } 2\ ns;)$$
$$\|$$
$$\texttt{while true do } (z := sx; \texttt{ wait for } 1\ ns;\ sx := sx + 1; \texttt{ wait for } 1\ ns;)$$

Example 4. Similarly, the following description is portable because the two processes execute in separate (delta) cycles.

$$\texttt{while true do } (sx := sx + 1; \texttt{ wait for } 1\ ns;)$$
$$\|$$
$$\texttt{while true do (wait until } sx = 5;\ sx := 0;\ )$$

In what follows, we develop the structural operational semantics for the given VHDL-93 subset by extending the work of Goossens (Goossens 1995).

4 STRUCTURAL OPERATIONAL SEMANTICS

Let *Val*, *Sig*, *Var*, *SVar*, *Expr*, and *SSt* denote the domains of values, signals, variables, shared variables, expressions and sequential statements respectively.

4.1 Semantic Entities

The state of a computation is captured by the history of values of each signal, the value bound to each variable and each shared variable, and the "activity" status of each postponed process.

Each process has a local store *LStore* that models the persistent value bindings of the variables and the signals. Without loss of generality, we assume that each variable implicitly holds an integer or a boolean value.* $Val = \mathcal{Z} \cup \mathcal{B}$. Each signal s is interpreted as a partial function $f : \mathcal{Z} \mapsto Val_\perp$ (representable as a subset of $\mathcal{Z} \times Val_\perp$) satisfying the following constraints (Goossens 1995): for $n < 0$, $f(n)$ is the value of the signal n time steps ago; $f(0)$ is the current value of the signal s; for $n \geq 0$, $f(n+1)$ is the projected value for n time steps into future. $f(1)$ contains the value scheduled for the next delta cycle. f contains at least $\langle -\infty, i\rangle$ and $\langle 0, v\rangle$ for initial value i and current value v of s. Note that only for $n \geq 0$ is $\langle (n+1), \perp\rangle$ a valid pair in f and encodes a null transaction for time n.

The domain *SStore* models the value bindings of the shared variables. To guarantee portability of VHDL-93 descriptions, access to shared variables must be restricted. In any simulation cycle, all processes may read a shared variable, or exactly one process may read and write a shared variable, without jeopardizing portability. However, one cannot permit arbitrary reads and writes across processes. To characterize portable VHDL-93 descriptions, we

*$\mathcal{Z}$ stands for the set of integers and $\mathcal{B}$ for the set of booleans.

associate with each shared variable, its current value, the type of last access (read/write) and the index of the process accessing it. The distinguished constants $\perp$ and $\top$ denote *undefined* and *all* respectively. The constant $\perp$ represents the case where a shared variable has not yet been accessed in the current cycle, while the constant $\top$ represents the case where all processes are permitted to access the shared variable.

It is also necessary to remember whether or not a postponed process is active, ready to be run at the end of the last delta cycle for the current time. Thus, the domain *PPStat* is defined as a subset of (postponed) process indices I.

Thus, the signatures of the *semantic domains* are*:

$$\begin{array}{lcl} \mathit{LStore} & = & (\mathit{Var} \mapsto \mathit{Val}) \times (\mathit{Sig} \mapsto \mathcal{P}(\mathcal{Z} \times \mathit{Val}_{\perp})) \\ \mathit{SStore} & = & (\mathit{SVar} \mapsto (\mathit{Val} \times (I \cup \{\perp, \top\}) \times \mathcal{P}(\{r, w\}))) \\ \mathit{PPStat} & = & \mathcal{P}(I) \end{array}$$

(a) Handling of Shared Variables for Portability

We now propose a scheme to ensure that the value bound to each shared variable in every cycle is well-defined (unique) in spite of the nondeterministic execution of the processes. For this purpose, we tag each shared variable with two additional pieces of information — the index of the process accessing it and the type of last access (read/write). One can capture the constraints for portability by defining a suitable transition function on the "states" of the shared variable as explained below:

- At the beginning of each simulation cycle, the state of a shared variable can be denoted by $\langle v, \perp, \emptyset \rangle$, where $\emptyset$ signifies that the variable has not yet been accessed. Assume that a read by process i is denoted by i, while the action of writing u is denoted by $\langle i, u \rangle$.

  If process i issues a read, the state of the shared variable changes to $\langle v, i, \{r\} \rangle$. The corresponding state transition is written as: $\langle v, \perp, \emptyset \rangle \xmapsto{i} \langle v, i, \{r\} \rangle$.

  If process i *now* writes a u, the state of the shared variable changes to $\langle u, i, \{r, w\} \rangle$ and the state transition is written as: $\langle v, i, \{r\} \rangle \xmapsto{\langle i,u \rangle} \langle u, i, \{r, w\} \rangle$.
- If the current state of the shared variable is $\langle v, i, \{r\} \rangle$ and process j issues a read, all *subsequent* accesses to the shared variable can only be reads, to ensure portability. This is because, a subsequent write to the shared variable by a process i (resp. j) can potentially affect the value of the shared variable read by the remaining statements in process j (resp. i). To capture this restriction, the following state transitions are defined, where $\top$ means *any process*:

  $\langle v, i, \{r\} \rangle \xmapsto{j} \langle v, \top, \{r\} \rangle$ and $\langle v, \top, \{r\} \rangle \xmapsto{\langle j,u \rangle} \langle u, \top, \{r, w\} \rangle$.

*$\mathcal{P}$ stands for the powerset operator.

The state $\langle v, \top, \{r\}\rangle$ should permit only reads by any process, while the state $\langle v, \top, \{r, w\}\rangle$ signifies a non-portable computation. This is mirrored by the following transitions:

$\langle v, \top, \{r\}\rangle \xmapsto{j} \langle v, \top, \{r\}\rangle$ and $\langle v, \top, \{r\}\rangle \xmapsto{\langle j,u\rangle} \langle u, \top, \{r, w\}\rangle$.

$\langle v, \top, \{r, w\}\rangle \xmapsto{j} \langle v, \top, \{r, w\}\rangle$ and $\langle v, \top, \{r, w\}\rangle \xmapsto{\langle j,u\rangle} \langle u, \top, \{r, w\}\rangle$.

- Now consider all possible transitions from the state $\langle v, i, \{w\}\rangle$.

  If process i issues a read, then only i should be allowed subsequent access, for portability. However, if process j issues a read, the code is not portable, because there is potential for ambiguity in the value that process j reads. In particular, it could be v or the value the shared variable had prior to v.

  $\langle v, i, \{w\}\rangle \xmapsto{i} \langle v, i, \{r, w\}\rangle$ and $\langle v, i, \{w\}\rangle \xmapsto{j} \langle v, \top, \{r, w\}\rangle \quad$ if $i \neq j$.

  If process i writes v, there is no change in the state. However, if process i writes u, then process i should have exclusive access, for portability.

  $\langle v, i, \{w\}\rangle \xmapsto{\langle i,v\rangle} \langle v, i, \{w\}\rangle$ and $\langle v, i, \{w\}\rangle \xmapsto{\langle i,u\rangle} \langle u, i, \{r, w\}\rangle \quad$ if $u \neq v$.

  If process j writes v, all processes can be permitted to write the same value, for portability. However, if process j writes u, then the code is not portable because the final value of the shared variable can be either v or u depending on how the processes are scheduled.

  $\langle v, i, \{w\}\rangle \xmapsto{\langle j,v\rangle} \langle v, \top, \{w\}\rangle \quad$ if $i \neq j$.

  $\langle v, i, \{w\}\rangle \xmapsto{\langle j,u\rangle} \langle u, \top, \{r, w\}\rangle$ if $i \neq j \wedge u \neq v$.

We crystallize and complete the above description by formally defining a deterministic finite state automaton that keeps track of accesses to a shared variable, to distinguish access-sequences that are portable from those that are potentially non-portable.

A deterministic finite-state automaton (DFA) is a 5-tuple (Hopcroft and Ullman 1979): $(\mathbf{Q}, \Omega, \Gamma, \mathbf{F}, q_0)$, where $\mathbf{Q}$ is the set of possible states, Ω is the alphabet, Γ is the transition function ($\Gamma : \mathbf{Q} \times \Omega \mapsto \mathbf{Q}$), $\mathbf{F}$ is the set of accepting states ($\subseteq \mathbf{Q}$), and q_0 is the initial state ($\in \mathbf{Q}$). We customize these sets for the problem at hand as follows:

- $\mathbf{Q} = Val \times (I \cup \{\perp, \top\}) \times \mathcal{P}(\{r, w\})$. *

  Recall that the shared variable value is tagged with the index of the process that accesses it and the type of last/allowed access. The possible types of accesses are: $\emptyset$, $\{r\}$, $\{w\}$ and $\{r, w\}$ representing *no access yet*, *read*-access, *write*-access, and *read/write*-access respectively. The $\perp$ value for the index signifies that no process has yet accessed the shared variable in the given simulation cycle, while the $\top$ value means that all processes are allowed access.

*I is *finite*, but *Val* is *infinite*. However, for our purposes, we make the simplifying but realistic assumption that *Val* is arbitrarily large but finite. (Overflow will trigger a run-time error.)

- $\Omega = I \cup (I \times Val)$.
  The state of a shared variable changes when it is accessed. A read-action is represented by the index of the process from which the read has been issued, while a write-action is represented by a pair consisting of the value to be written and the index of the process from which the write has been issued.
- The deterministic transition function Γ is given below:

$$\begin{array}{ll}
\langle v, \bot, \emptyset\rangle \xmapsto{i} \langle v, i, \{r\}\rangle & \langle v, \bot, \emptyset\rangle \xmapsto{\langle i,u\rangle} \langle u, i, \{w\}\rangle \\
\langle v, i, \{r\}\rangle \xmapsto{i} \langle v, i, \{r\}\rangle & \langle v, i, \{r\}\rangle \xmapsto{j} \langle v, \top, \{r\}\rangle \quad \text{if } i \neq j \\
\langle v, i, \{r\}\rangle \xmapsto{\langle i,u\rangle} \langle u, i, \{r, w\}\rangle & \langle v, i, \{r\}\rangle \xmapsto{\langle j,u\rangle} \langle u, \top, \{r, w\}\rangle \quad \text{if } i \neq j \\
\langle v, i, \{w\}\rangle \xmapsto{i} \langle v, i, \{r, w\}\rangle & \langle v, i, \{w\}\rangle \xmapsto{j} \langle v, \top, \{r, w\}\rangle \quad \text{if } i \neq j \\
\langle v, i, \{w\}\rangle \xmapsto{\langle i,v\rangle} \langle v, i, \{w\}\rangle & \langle v, i, \{w\}\rangle \xmapsto{\langle i,u\rangle} \langle u, i, \{r, w\}\rangle \quad \text{if } u \neq v \\
\langle v, i, \{w\}\rangle \xmapsto{\langle j,v\rangle} \langle v, \top, \{w\}\rangle \quad \text{if } i \neq j & \langle v, i, \{w\}\rangle \xmapsto{\langle j,u\rangle} \langle u, \top, \{r, w\}\rangle \quad \text{if } i \neq j \wedge u \neq v \\
\langle v, i, \{r, w\}\rangle \xmapsto{i} \langle v, i, \{r, w\}\rangle & \langle v, i, \{r, w\}\rangle \xmapsto{j} \langle v, \top, \{r, w\}\rangle \quad \text{if } i \neq j \\
\langle v, i, \{r, w\}\rangle \xmapsto{\langle i,u\rangle} \langle u, i, \{r, w\}\rangle & \langle v, i, \{r, w\}\rangle \xmapsto{\langle j,u\rangle} \langle u, \top, \{r, w\}\rangle \quad \text{if } i \neq j \\
\langle v, \top, \{r\}\rangle \xmapsto{j} \langle v, \top, \{r\}\rangle & \langle v, \top, \{r\}\rangle \xmapsto{\langle j,u\rangle} \langle u, \top, \{r, w\}\rangle \\
\langle v, \top, \{w\}\rangle \xmapsto{j} \langle v, \top, \{r, w\}\rangle & \\
\langle v, \top, \{w\}\rangle \xmapsto{\langle j,v\rangle} \langle v, \top, \{w\}\rangle & \langle v, \top, \{w\}\rangle \xmapsto{\langle j,u\rangle} \langle u, \top, \{r, w\}\rangle \quad \text{if } u \neq v \\
\langle v, \top, \{r, w\}\rangle \xmapsto{j} \langle v, \top, \{r, w\}\rangle & \langle v, \top, \{r, w\}\rangle \xmapsto{\langle j,u\rangle} \langle u, \top, \{r, w\}\rangle
\end{array}$$

- $\mathbf{F} = (Val \times \{\bot\} \times \{\emptyset\}) \bigcup (Val \times I \times \{\{r\}, \{w\}, \{r, w\}\}) \bigcup (Val \times \{\top\} \times \{\{r\}, \{w\}\})$
  Informally, the set of accepting states characterizes the safe sequences of reads and writes for portability.
- $q_0 = \langle v, \bot, \emptyset\rangle$.
  v is the value of the shared variable at the beginning of a simulation cycle. The index $\bot$ and the type of access $\emptyset$ signify that the shared variable has not yet been accessed.

The states in $(Val \times \{\bot\} \times \{\{r\}, \{w\}, \{r, w\}\}) \cup (Val \times I \times \{\emptyset\}) \cup (Val \times \{\top\} \times \{\emptyset\})$ are *unreachable* from q_0, and those in $Val \times \{\top\} \times \{\{r, w\}\}$ are the dead states.

Lemma 41 *Every string (of read/write actions) in the language of the DFA satisfies one of the following properties:*

(a) *Every action in the string is a read action, that is, it is in I. Furthermore, the value of the shared variable remains unchanged.*

(b) *Every action in the string contains the same index i, that is, it is either i or $\langle i, ?v_{al}\rangle$. Furthermore, the final value of the shared variable is the last value written.*

(c) *Every action in the string is a write action with the same value component, that is, it is in $I \times \{\{v\}\}$. Furthermore, the final value of the shared variable is the value written.*

Proof Sketch: It is easy to see the result by starting from the final states and tracing all the relevant transitions in reverse. •

Lemma 41 lays the foundation for defining portability. Let $Size(rs)$ return the size of the set of indices in the read sequence rs. ($size(ijkjij) = 3$.)

Lemma 42 *Let $q, q_1, q_2 \in Q$, and $rs_1, rs_2 \in I^*$ be two sequences of reads that are* permutations *of each other. Then, the relation $(q \xmapsto{rs_1}_* q_1 \wedge q \xmapsto{rs_2}_* q_2 \Rightarrow q_1 = q_2)$ holds.*

Proof: We consider two cases: (a) $Size(rs_1) <= 1$. Trivial. (b) $Size(rs_1) > 1$. Follows straightforwardly from the definition of the transition function. •

(b) Advancing time

A program is evaluated with respect to the global structure *Store* defined as follows:

$$Store \;=\; \mathcal{P}(LStore) \times SStore \times PPStat$$

$$\begin{array}{lll} \sigma, \sigma_i & \in & LStore \\ \psi & \in & SStore \end{array} \qquad \begin{array}{lll} \Sigma, \Sigma_i & \in & \mathcal{P}(LStore) \\ \xi & \in & PPStat \end{array}$$

Two functions — $\mathcal{T}, \mathcal{U} : Store \mapsto Store$ — are defined to advance time and delta time respectively (Goossens 1995).

The function $\mathcal{T}$ effects only the value of the signals, the state of the shared variables, and the status of the postponed processes. It leaves unchanged the values of the variables and the shared variables.

The function $\mathcal{T}$ transforms a *Store* as follows:

- The (local) variables are unchanged: $\mathcal{T}(\sigma_i)(x) = \sigma_i(x)$.
- For signals: $\mathcal{T}(\sigma_i)(s) \;=\; \{\langle n-1, v\rangle \mid \langle n, v\rangle \in \sigma_i(s)\} \cup \{\langle 0,\; \sigma_i(s)(2)\; else\; \sigma_i(s)(0)\rangle\}$ Here x *else* y means "if x is defined then x else y". Note that there is an error in (Goossens 1995) since it has 1 in place of 2, and as shown later, $\sigma_i(s)(1)$ is always undefined when $\mathcal{T}$ is applied.
- For shared variables: $\mathcal{T}(\psi)(sx) = \langle v, \perp, \emptyset\rangle$, where $\psi(sx) = \langle v, \_, \_\rangle$.
- For the status of the postponed-processes: $\mathcal{T}(\xi) = \emptyset$.

A signal s is *active* if $\exists \sigma_i \in \Sigma_I, v \in Val_\perp : \langle 1, v\rangle \in \sigma_i(s)$. A process can *resume* if it is sensitive to an active signal or it has been timed-out. (See Section 4.4.)

The function $\mathcal{U}$ effects only the value of the *active* signals, the state of the shared variables, and the status of the postponed processes. It leaves unchanged the values of the variables, the shared variables, and the inactive signals.

- For shared variables: $\mathcal{U}(\psi)(sx) = \langle v, \perp, \emptyset\rangle$, where $\psi(sx) = \langle v, \_, \_\rangle$.
- For active signals s, the current value is replaced by $r_s \in Val$, obtained through the signal resolution function f_s applied to the driving values of the signal (Goossens 1995):

$$r_s = f_s\{\{v_i \mid \exists i \in I : \langle 1, v_i\rangle \in \sigma_i(s) \wedge v_i \neq \mathsf{null}\}\}$$
$$\mathcal{U}(\sigma_i)(s) = (\sigma_i(s) \setminus \{\langle 0, \sigma_i(s)(0)\rangle, \langle 1, \sigma_i(s)(1)\rangle\}) \cup \{\langle 0, r_s\rangle\}$$

  Here, $\{\{.\}\}$ denotes a multiset. f_s is assumed to be a commutative resolution function. null signifies disconnection. Note that inactive signals do not participate in determining the final resolved value.
- The determination of the status of the postponed processes is described in Section 4.4.

The signatures of the relevant *semantic functions* are:

$\mathcal{E} : Expr \mapsto LStore \times SStore \mapsto Val_\perp \times SStore$
$\rightarrow_{ss}, \rightarrow_{proc}: (LStore \times SStore \times SSt) \mapsto (LStore \times SStore \times SSt)$
$\rightarrow_{pgm} : (Store \times SSt) \times (Store \times SSt)$

An expression is evaluated with respect to the local/shared store and it returns a value and a (possibly modified) shared store. A program (resp. statement) and a store evolve into a new program (resp. statement) and an (resp. unique) updated store.

4.2 Semantics of Expressions

Let *fst* stand for the function that extracts the first component of a pair and the set *dom(f)* stand for the domain of a partial function f. Let $\psi_v(sx) \in Val$ denote the first (value) component of the triple $\psi(sx)$ associated with the shared variable sx. For concreteness, we specify the rules for variables, signals and for compound expressions involving a binary operator. Also, $\psi[sx \mapsto st] \;=\; (\lambda sy.\ \texttt{if}\ sx \equiv sy\ \texttt{then}\ st\ \texttt{else}\ \psi(sy))$.

$$\begin{array}{rcll}
\mathcal{E}\,[\![x]\!]\,\langle\sigma,\psi\rangle & = & \langle\sigma(x),\psi\rangle & \\
\mathcal{E}\,[\![sx_i]\!]\,\langle\sigma,\psi\rangle & = & \langle\psi_v(sx_i),\psi[sx_i \mapsto st]\rangle, & \text{if } \psi(sx_i) \overset{i}{\longmapsto} st \\
\mathcal{E}\,[\![s]\!]\,\langle\sigma,\psi\rangle & = & \langle\sigma(s)(0),\psi\rangle & \\
\mathcal{E}\,[\![s'\mathtt{delayed}(e_i)]\!]\,\langle\sigma,\psi\rangle & = & \langle\sigma(s)(n),\psi\rangle & n = max\{m \mid m \in dom(\sigma(s)) \wedge \\
 & & & \quad m \leq -\mathsf{fst}(\mathcal{E}\,[\![e_i]\!]\,\langle\sigma,\psi\rangle) \leq 0\} \\
\mathcal{E}\,[\![e_i\ bop\ e_i']\!]\,\langle\sigma,\psi\rangle & = & \langle v\ bop\ v',\psi''\rangle & \text{if } \mathcal{E}\,[\![e_i]\!]\,\langle\sigma,\psi\rangle = \langle v,\psi'\rangle \\
 & & & \mathtt{and}\ \mathcal{E}\,[\![e_i']\!]\,\langle\sigma,\psi'\rangle = \langle v',\psi''\rangle
\end{array}$$

The value of the delayed expression is required to be nonnegative. (There is a minor error in (Goossens 1995) here.) $s'\mathtt{delayed}(0\ ns) \neq s$ during any simulation cycle where there is a change in the value of s. (See Section 14.1 in the LRM (IEEE 1993).) For correct handling of `delayed`-attribute we also need to store the previous value of each signal in the *LStore*.

Theorem 41 *The meaning of an expression is independent of the order of evaluation of its subexpressions.*

Proof Sketch: The meaning of an expression consists of its value and the shared store. As the expressions only inspect (read) the values bound to variables, shared variables and signals, and never modify (write) them, the value component is independent of the order of evaluation. So the result follows from Lemma 42 and structural induction. •

4.3 Semantics of Statements

The semantic rules for all but the signal assignment statement and the wait statement are more or less standard.

For concreteness, the rules for assignment to a shared variable and for while-loop can be specified as follows: (Recall that, $\sigma[x \mapsto v] \;=\; (\lambda y.\ \texttt{if}\ x \equiv y\ \texttt{then}\ v\ \texttt{else}\ \sigma(y))$.)

$$\frac{\mathcal{E}\,[\![e]\!]\,\langle\sigma,\psi\rangle = \langle v,\psi'\rangle \quad \wedge \quad \psi'' = \psi'[\,sx_i \mapsto \Gamma(\psi'(sx_i),\langle i,v\rangle)\,]}{\langle\sigma,\psi,\ \ sx_i := e\ ;\ ss\ \rangle \rightarrow_{ss} \langle\sigma,\psi'',\ ss\ \rangle}$$

$$\frac{\mathcal{E}\,[\![e]\!]\,\langle\sigma,\psi\rangle = \langle\mathsf{true},\psi'\rangle}{\langle\sigma,\psi,\ \texttt{while}\ e\ \texttt{do}\ ss'\ \rangle \rightarrow_{ss} \langle\sigma,\psi',\ ss'\ ;\ \texttt{while}\ e\ \texttt{do}\ ss'\ \rangle}$$

$$\frac{\mathcal{E}\,[\![e]\!]\,\langle\sigma,\psi\rangle = \langle\mathsf{false},\psi'\rangle}{\langle\sigma,\psi,\ \texttt{while}\ e\ \texttt{do}\ ss'\ ;\ ss\rangle \rightarrow_{ss} \langle\sigma,\psi',\ ss\ \rangle}$$

The signal assignment statement changes the value of a signal by adding a time-value pair and eliminating all other pairs that are scheduled for a later time. Let $update(\sigma,s,v,t) = (\sigma(s) \setminus \{\langle n,\sigma(s)(n)\rangle \mid n > t\}) \cup \{\langle t+1,v\rangle\}$. (There is a minor error in (Goossens 1995) here.)

$$\frac{\mathcal{E}\,[\![e]\!]\,\langle\sigma,\psi\rangle = \langle v,\psi'\rangle \;\wedge\; \mathcal{E}\,[\![et]\!]\,\langle\sigma,\psi'\rangle = \langle t,\psi''\rangle \;\wedge\; t \geq 0}{\langle\sigma,\psi,\ s\ <=\ e\ \texttt{after}\ et\ ;\ ss\rangle \rightarrow_{ss} \langle update(\sigma,s,v,t),\psi'',\ ss\ \rangle}$$

4.4 Semantics of Processes and Programs

The semantic rules for processes/postponed processes (that is, for $\rightarrow_{proc}$) are similar to those for statements (that is, $\rightarrow_{ss}$). A process unwinds into a potentially infinite sequence of statements.

A program (that is, fully elaborated behavioral VHDL-93 description) consists of a collection of sequential processes that execute independently. Global synchronization and (synchronous) communication through (common) signals takes place when all the processes reach a **wait**-statement. Otherwise, these processes execute asynchronously between **wait**-statements and can communicate (asynchronously) through shared variables. (We use $\|_I \langle\sigma_i,\psi,\xi,ss_i\rangle$ for $\langle\ \langle\|_I\ \sigma_i,\psi,\xi\rangle, \|_I\ ss_i\ \rangle$.)

Rule 1:

$$\frac{\langle\sigma_j,\psi,\ ss_j\ \rangle \rightarrow_{ss} \langle\sigma'_j,\psi',\ ss'_j\ \rangle}{\|_{I\cup\{j\}}\ \langle\sigma_i,\psi,\xi,\ ss_i\ \rangle \ \rightarrow_{pgm}\ \|_{I\cup\{j\}}\ \langle\sigma'_i,\psi',\xi,\ ss'_i\ \rangle}$$

where $\sigma'_i = \sigma_i \wedge ss'_i = ss_i$ for all $i \neq j$, and $\sigma'_i = \sigma'_j \wedge ss'_i = ss'_j$ for $i = j$. This rule is applicable as long as the first statement of ss_j is not a **wait**-statement.

In the presence of shared variables, the nondeterministic execution of processes embodied in this rule may yield different results. However we can define restrictions that ensure that all possible executions are "equivalent", as explained later.

If no processes can resume (and there are no postponed processes that can run in the last delta cycle), then the global simulation time is advanced by one. To achieve this, the store is updated using $\mathcal{T}$ and the timeout value in the wait-statment is decremented by one. We use $ws_i[te_i,be_i]$ for (`wait on` S_i `for` te_i `until` be_i).

Rule 2:

$$\frac{\neg resume(\|_I\ \langle\sigma_i,\psi,\xi,\ ws_i[te_i,be_i]\ ;ss_i\ \rangle) \;\wedge\; \forall\, i \in I :\ \langle tv_i,\psi'\rangle = \mathcal{E}\ [\![te_i]\!]\ \langle\sigma_i,\psi\rangle}{\|_I\ \langle\sigma_i,\psi,\xi,\ ws_i[te_i,be_i]\ ;ss_i\ \rangle \ \rightarrow_{pgm}\ \|_I\ \langle\mathcal{T}(\sigma_i),\mathcal{T}(\psi),\mathcal{T}(\xi),\ ws_i[tv_i-1,be_i]\ ;ss_i\ \rangle}$$

$$\begin{aligned} & resume(\|_I\ \langle\sigma_i,\psi,\xi,\ ws_i[te_i,be_i]\ ;ss_i\ \rangle) \\ \equiv\ & \exists i \in I : resume(\sigma_i,\psi,te_i) \vee (\xi \neq \emptyset) \end{aligned}$$

A process can *resume* if it contains a signal that is active or it has been timed out.

$$\begin{aligned}
resume(\sigma_i, \psi, te_i) &\equiv active(\sigma_i) \vee timeout(\sigma_i, \psi, te_i)\\
active(\sigma) &\equiv \exists s \in dom(\sigma),\ \exists v \in Val_\perp : \langle 1, v\rangle \in \sigma(s)\\
timeout(\sigma, \psi, te) &\equiv fst(\mathcal{E}\ [\![te]\!]\ \langle \sigma, \psi\rangle) = 0
\end{aligned}$$

A delta cycle* is initiated in the above situation. Non-postponed processes are executed if they are timed-out or if the condition in the wait-statement holds.

Rule 3:

$$\frac{\exists i \in I : \neg postponed?(i) \ \wedge\ resume(\sigma_i, \psi, te_i)}{\|_I\ \langle \sigma_i, \psi, \xi,\ ws_i[te_i, be_i]\ ; ss_i\ \rangle\ \rightarrow_{pgm}\ \|_I\ \langle \mathcal{U}(\sigma_i), \mathcal{U}(\psi), \xi',\ \mathcal{F}(ws_i[te_i, be_i]\ ; ss_i)\ \rangle}$$

Informally, the function $\mathcal{F}$ executes the wait-statements for those non-postponed processes that can run.

$$\mathcal{F}(ws_i[te_i, be_i]\ ; ss_i) = \begin{cases} ss_i & \text{if } \neg postponed?(i) \ \wedge\ run(\sigma_i, \mathcal{U}(\sigma_i), \psi, te_i, be_i)\\ ws_i[tv_i, be_i]\ ; ss_i & \text{otherwise, where } tv_i = fst(\mathcal{E}\ [\![te_i]\!]\ \langle \sigma_i, \psi\rangle) \end{cases}$$

$$\begin{aligned}
run(\sigma_i, \sigma_i', \psi, te_i, be_i) \equiv\ &(\ timeout(\sigma_i, \psi, te_i) \ \vee\\
&[\ \exists s \in S_i : event(\sigma_i, \sigma_i', s) \ \wedge\ fst(\mathcal{E}\ [\![be_i]\!]\ \langle \sigma_i', \psi\rangle)\ ]\ )
\end{aligned}$$

$$event(\sigma, \sigma', s) \equiv \sigma(s)(0) \neq \sigma'(s)(0)$$

Effectively, the timeout expression is evaluated only once in the first delta-cycle, while the condition in the wait-statement is evaluated in every delta cycle in which there is an event on a signal that the process/condition is "sensitive" to. Whether or not a postponed process can run in the last delta cycle is determined as follows.

$$\xi' \equiv \xi \cup \{i \in I \ \mid\ postponed?(i) \ \wedge\ run(\sigma_i, \mathcal{U}(\sigma_i), \psi, te_i, be_i)\}$$

The postponed processes that can run are executed only when no non-postponed process can resume. The condition that causes a postponed process to run may no longer hold in the state in which the postponed process is actually executed. (See Section 8.1 in the LRM (IEEE 1993).) It is an error if the execution of a postponed process initiates another delta-cycle.

*A delta cycle is a simulation cycle where the global time is not advanced.

Rule 4:

$$\frac{\begin{array}{c}\neg(\exists i \in I : \neg \mathit{postponed?}(i) \wedge \mathit{resume}(\sigma_i, \psi, te_i)) \quad \wedge \quad \xi \neq \emptyset \quad \wedge \\ \forall i \in \xi : (\ \langle \mathcal{U}(\sigma_i), \mathcal{U}(\psi),\ ss_i\ \rangle \rightarrow_{ss} \langle \sigma'_i, \psi',\ ws'_i[te'_i, be'_i]\ ; ss'_i\ \rangle\ ) \quad \wedge \\ \forall i \in I - \xi : (\ (\sigma_i = \sigma'_i) \wedge (ws_i[te_i, be_i]\ ; ss_i \ \equiv\ ws'_i[te'_i, be'_i]\ ; ss'_i)\ ) \\ \wedge \quad \forall i \in I : \neg \mathit{ready}(\sigma'_i, \mathcal{U}(\sigma'_i), \psi', te'_i, be'_i)\end{array}}{\|_I \ \langle \sigma_i, \psi, \xi,\ ws_i[te_i, be_i]\ ; ss_i\ \rangle \ \rightarrow_{pgm}\ \|_I \ \langle \sigma'_i, \psi', \emptyset,\ ws'_i[te'_i, be'_i]\ ; ss'_i\ \rangle}$$

Again, the well-definedness of $\rightarrow^*_{pgm}$ depends on the portability restrictions we impose. Also recall that $\rightarrow_{ss}$ is transitive.

5 PROPERTIES OF THE OPERATIONAL SEMANTICS

We are now ready to formally define the notion of *portability*. Let $\rightarrow^*_{pgm}$ be the reflexive transitive closure of $\rightarrow_{pgm}$, and ($\mathbf{Q}$,Ω,Γ,$\mathbf{F}$, q_0) be the DFA described in Section a.

Definition 51 *A program (*$\|_I$ while true do ss_i*) is a* portable VHDL-93 description *if, for every computation of the form*

$$\|_I \ \langle \sigma_i, \psi, \xi, \text{while true do } ss_i \rangle \ \rightarrow^*_{pgm}\ \|_I \ \langle \sigma'_i, \psi', \xi', ss'_i \rangle,$$

we have $\forall sx \in SVar : \ (\psi(sx) = q_0) \ \Rightarrow\ \psi'(sx) \in \mathbf{F}$.

From Lemma 41 this implies permitting arbitrary interleaving of statement-executions as long as each shared variable is accessed either by all processes in read-mode, or by all processes in write-mode and the same value is written in, or by the same process in read/write mode, *between two successive synchronization points.*

We now investigate properties about the semantics of the portable VHDL-93 descriptions, to gain deeper understanding and to increase our confidence in the formalization of the semantics.

Theorem 51 *A process that does not contain a* wait*-statement loops forever.*

Theorem 52 *The semantics of expressions $\mathcal{E}$ (resp. statements $\rightarrow_{ss}$) is* determisnitic.

Theorem 53 *The statement* `wait on` $\emptyset$ `for` ∞ `until true;` *causes the enclosing process to suspend forever.*

We now show that the portable VHDL-93 descriptions can be given a unique meaning.

Theorem 54 *The values bound to variables, shared variables, and signals of the processes of a* portable VHDL-93 description *sampled when all of them are waiting are unique.*

Proof Sketch: Effectively, we need to show that, if

$$||_I \ \langle \sigma_i, \psi, \xi,\ ws_i[te_i, be_i]\ ; ss_i\ ;\ ws'_i[te'_i, be'_i]\ ; ss'_i\ \rangle \ \rightarrow^*_{pgm}\ ||_I \ \langle \sigma'_i, \psi', \xi',\ ws'_i[te'_i, be'_i]\ ; ss'_i\ \rangle$$

holds, then σ'_i, ψ', and ξ' are unique, where each ss_i *does not* contain any wait-statements.

Now consider the four semantic rules for $\rightarrow_{pgm}$ given in Section 4.4, which have disjoint antecedents. The application of **Rule 1** and **Rule 4** for portable descriptions yields unique result because of Definition 51 and Lemma 41. The application of **Rule 2** and **Rule 3** for the wait-statement define a unique transformation because the resolution functions f_s and $\mathcal{U}$, and the time increment function $\mathcal{T}$ are one to one and total. •

Theorem 55 *The portability condition given in Definition 51 is* sufficient but not necessary *for VHDL-93 descriptions to have a unique meaning.*

Proof: There exist *trivial* descriptions such as $||_I$ `while true do` *sx* := *sx*; that have a unique meaning, but violate the portability definition. •

Theorem 56 *The portability condition given in Definition 51 is* nonlocal.

Proof: Consider the two processes **PS** (with *sflag* initially true)

`while true do (if` *sflag* `then` *sx := 1* `else` *sx := 2;* `wait for` *2 ns;*)

||

`while true do (`*sx := 1;* `wait for` *2 ns;*)

executing in parallel with each of the following processes:

P1: `while true do (` `wait for` *1 ns; sflag :=* true; `wait for` *1 ns;*)

P2: `while true do (` `wait for` *1 ns; sflag :=* false; `wait for` *1 ns;*)

PS with **P1** is portable; **PS** with **P2** is not portable. •

As a consequence of this nonlocality it is not possible to incrementally check VHDL-93 descriptions for portability.

Theorem 57 *Given a VHDL-93 description, it is not possible to determine* statically *(that is, at compile time) whether or not it is portable.*

Proof Sketch: If the VHDL-93 description contains a "free" shared variable whose value is not known at compile-time, then it is obvious that portability check cannot be made statically. The program **PS** and the shared variable *sflag* given in the proof of Theorem 56 exemplify this situation.

Interestingly, the result holds even when all the variables, shared variables and signals are completely defined. The test for portability can then be reduced to determining whether or not two programs compute the same function.

```
while true do (  ...sx := Func1(x1) ...;  x1 := x1 + 1;  wait for 1 ns;)
                                     ||
while true do (  ...sx := Func2(x2) ...;  x2 := x2 + 1;  wait for 1 ns;)
```

Let *x1* and *x2* be initially 0; *Func1* and *Func2* abbreviate the effect of the code that computes *sx* from *x1* and *x2*. The above program is portable if and only if the value written into *sx* by the two processes in every step is identical. That is, *Func1* and *Func2* stand for the same function. However, since equivalence problem for Turing-complete languages is undecidable, the portability cannot be determined at compile-time. •

In order to detect lack of portability at run-time, the simulator can be augmented with additional information specified in the DFA described in Section a. One can view this as a new implementation of the abstract data type *shared variable.*

6 CONCLUSIONS

The designers of VHDL-93 extended VHDL-87 by introducing shared variables and postponed processes into the language. Here, we developed a structural operational semantics for a behavioral subset of VHDL-93 along the lines of Goossens' work. In particular, we extended the underlying semantic model to accomodate new VHDL-93 features. This formal specification can serve as a guide to the implementor and as a correctness criteria for the VHDL-93 simulator. Furthermore, VHDL-93 LRM stipulates that the VHDL-93 descriptions that generate different behaviors on different simulators are erroneous. In this paper, we explored causes of non-portability through examples and later proposed sufficient conditions for a VHDL-93 behavioral description with shared variables and postponed processes to have unique meaning. We also specified how a simulator can be augmented with additional information to detect and flag non-portability. We then stated some basic properties about VHDL-93 descriptions, and showed that test for portability is neither local nor static.

Pragmatically, our approach has some limitations compared to the recent proposal for introducing *protected types* into the language to deal with shared variables (Willis 1996). In this proposal, Hoare's *monitors* are used as the basis for implementing shared variable mutual exclusion semantics. Each procedure defined in the monitor can be thought of as encapsulating "atomic" sequence of reads and writes. Any arbitrary permutation of these procedure calls from various processes in a simulation cycle is assumed to be acceptable to the designer as long as each call is executed atomically. Thus, this construct enables expression of *algorithmic nondeterminism* (Willis 1996).

REFERENCES

Bhasker, J. (1994) *A VHDL Primer*, Second Edition, Prentice Hall, Inc..

Breuer, P., Sanchez, L., and Kloos, C. D. (1995) A simple denotational semantics, proof theory and validation condition generator for unit delay VHDL, *Formal Methods in System Design,* **7(1-2)**.

Kloos, C. D., and Breuer, P., eds. (1995) *Formal Semantics of VHDL,* **307**, Kluwer Academic Publishers.

Goossens, K. G. W. (1995) Reasoning about VHDL using operational and observational semantics, *Advanced Research Workshop on Correct Hardware Design Methodologies,* ESPRIT CHARME, Springer Verlag.

Hennessy, M. (1990) *The Semantics of Programming Languages: An Elementary Introduction using Structural Operational Semantics*, John Wiley & Sons.

Hopcroft, J., and Ullman, J. (1979) *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley Co.

Institute of Electrical and Electronics Engineers, 345 East 47th Street, New York, USA. *IEEE Standard VHDL Language Reference Manual, Std 1076-1993.*

van Tassel, J. P. (1993) *Femto-VHDL: The Semantics of a Subset of VHDL and its Embedding in the HOL Proof Assistant,* Ph. D. Dissertation, University of Cambridge.

Willis, J., *et al* (1996) *Shared Variable Language Change Specification,* 1996. (Draft)

Wilsey, P. A. (1992) Developing a formal semantic definition of VHDL, In: Mermet, J., eds, *VHDL for Simulation, Synthesis and Formal Proofs of Hardware,* Kluwer Academic Publishers, 243–256.

7 BIOGRAPHY

K. Thirunarayan is an Associate Professor in the Dept. of Computer Science and Engineering at Wright State University, Dayton, Ohio. He received his B. Tech degree in Electrical Engineering from the Indian Institute of Technology, Madras, in 1982, and his M.E. degree in Automation from the Indian Institute of Science, Bangalore, in 1984. He received his Ph.D. in Computer Science from the State University of New York at Stony Brook in 1989. His current research interests are in formal specification and verification of hardware, programming languages, and knowledge representation.

Dr. Robert L. Ewing works in the areas of hardware description languages (VHDL), formal verification, analog synthesis and design, built-in testability and electronic devices at Wright Laboratory, Wright-Patterson Air Force Base. He is an adjunct professor at Wright State University and also at the Air Force Institute of Technology, in the area of electronic devices.