Formal Methods in System Design, 18, 69–88, 2001

Structural Operational Semantics for a Portable Subset of Behavioral VHDL-93

KRISHNAPRASAD THIRUNARAYAN* tkprasad@cs.wright.edu
Department of Computer Science and Engineering, Wright State University, Dayton, OH-45435, USA

ROBERT L. EWING
Air Force Research Laboratory, Information Directorate Wright-Patterson Air Force Base, Dayton, Ohio 45433, USA

Received August 27, 1996; Revised June 30, 1999

Abstract. Goossens defined 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, non-deterministic 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. We analyze the computability of the portability condition and show that portability checks are neither local nor static. Our formal specification can serve as a correctness criteria for a VHDL-93 simulator or can be used as a basis for coding a simulator.

Keywords: hardware description languages (VHDL), formal semantics (of VHDL-93), formal specification (of VHDL-93 simulation cycle), shared variables (in VHDL-93) and portability

1. Introduction

VHDL has been designed to facilitate specification, documentation, communication and formal manipulation of hardware designs at various levels of abstraction [2]. The semantics of VHDL-93 are given in English prose in [11]. 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 will enhance our understanding of the various VHDL-93 constructs/features.

There have been a number of proposals for a formal semantics of VHDL [1, 3–8, 12–18, 20], many of them dealing with "unit delay" subsets of VHDL-87. To facilitate formal verification, Belhadj et al. [1] specify VHDL-87 attributes involving time in SIGNAL language. Petri-nets are used to specify the semantics of a VHDL-87 subset in [4, 6, 13]. Van Tassel [18] mechanizes reasoning about Femto-VHDL in HOL, while Breuer et al. [3] develop

*http://www.cs.wright.edu/~tkprasad.

denotational semantics for a unit delay VHDL-87 subset. Deharbe and Borrione [5] give operational semantics for a VHDL-87 subset that excludes quantitative timing information (among other things) in terms of abstract machines, while Goossens [8] defines structural operational semantics [9] for a VHDL-87 subset that includes almost all fundamental behavioral constructs in a single VHDL-87 entity. Börger et al. (Chapter 4, [7]) deal with VHDL-93 extension, while those in [12, 14, 15] deal with VHDL-AMS extension. The former provides a definition of VHDL-93 features using EA-machines, but does not formally prove properties of the semantics. Shanker and Slagle [16] provide a lucid discussion of the many tradeoffs involved in developing a formal semantics of VHDL. They then provide an intuitive semantics to a behavioral subset of VHDL using a simple polymodal logic with two modalities—one for flow of control and other for flow of time.

In this paper we build on Goossens work which deals with a subset of behavioral VHDL-87. We define 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 [8] no longer holds in the presence of shared variables and non-deterministic process execution. 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 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 [11].

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

We chose structural operational approach to VHDL semantics over other available formalisms because of (1) its compactness, clarity, and generality (at the expense of some notational complexity), (2) the need to deal with delta delays, and (3) its straightforward translation into a functional language which can serve as an executable specification. In other words, it describes abstractly, the data structures and the operations needed to code a simulator. Our formalization can be viewed as a specification that can be systematically refined into a simulator, or against which the correctness of a simulator can be verified. Realistically, the executable specification can serve as a reference against which the results of a simulator are compared. (This approach gets complicated in the presence of ambiguous programs.) Furthermore, the specification describes additional run-time machinery that can be incorporated into a VHDL-93 simulator to flag VHDL-93 descriptions that are potentially ambiguous. We believe that the complexity of the semantics specification reflects the intrinsic complexity of the concepts being modeled by VHDL. 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 [8, 18].

The rest of this paper is organized as follows: Section 2 presents the abstract syntax of the VHDL-93 subset. Section 3 explores and illustrates the causes of ambiguity informally.

Section 4 specifies the structural operational semantics of VHDL-93 subset. The primary emphasis is on the changes to the semantics in [8] resulting from the introduction of shared variables and postponed processes. Section 5 formally defines *portable VHDL-93 description*, and proves interesting properties about portability condition. 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. (The corresponding VHDL-93 concrete syntax should be obvious with the exception of the process statement: (`while` true `do` ss_i) $\equiv$ (i : `process begin` ss_i `end process` i ;).)

- **Syntactic Categories:**

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

- **Definitions:**

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

A program in this VHDL-93 subset can be viewed as a *fully elaborated behavioral VHDL-93 description* [11]. 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. To characterize portable VHDL-93 descriptions, we associate the identity of a process with each occurrence of a shared variable. (see Section 4.1.1 for details.) So we have tagged the meta-variables *proc*, *p*, *pp*, *ss*, and *e* with subscript *i* representing the index of the associated process $proc_i$. (Alternatively, this can be easily specified through the static semantics.)

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*. 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 also assume that all the signals with multiple drivers have a suitable resolution function associated with them.

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

3. The causes of ambiguity

Intuitively, a VHDL-93 description is unambiguous if it assigns the same "observable" values to all (shared) variables and signals. (This notion will be formalized in Section 5.) The following examples illustrate the causes of ambiguity and motivate restrictions required to guarantee portability of VHDL-93 descriptions. We assume that all (shared) variables of integer type are initially 0.

Example 1. The following VHDL description is ambiguous 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 ambiguous as the value of *z* after *t*-ns can be 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 has a unique meaning, and the value of *sx* after *t*-ns is $\lceil \frac{t}{2} \rceil$.

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

Note that, 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.

Example 4. Similarly, the following description has a unique meaning.

```
while true do (sx := sx + 1; wait for 1 ns;)
                          ||
while true do (wait until sx = 5; sx := 0; )
```

Note that the two processes execute assignments in separate (delta) cycles.

In what follows, we develop the structural operational semantics for the given VHDL-93 subset by extending the work of Goossens [8].

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 either an integer or a boolean value. $Val = \mathcal{Z} \cup \mathcal{B}$, where $\mathcal{Z}$ stands for the set of integers and $\mathcal{B}$ for the set of booleans. Each signal s is interpreted as a partial function $f : \mathcal{Z} \mapsto Val_\perp$ satisfying the following constraints [8]: 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. (In other words, negative time points correspond to physical time in the past, time point 0 corresponds to the current time now, time point 1 corresponds to delta delay, and the remaining time points correspond to physical time (advanced by 1) in the future.) 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 ensure that VHDL-93 descriptions are unambigous, access to shared variables by the processes must be restricted. For this purpose, the shared variable value is tagged with the index of the process that accessed it and the type of last/allowed access. The permitted types of accesses are: $\emptyset$, $\{r\}$, $\{w\}$ and $\{r, w\}$ representing *no access yet*, *read*-access, *write*-access, and *read/write*-access respectively. The distinguished constants $\perp$ and $\top$, when used as a process index, denote *undefined* and *any* 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.

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.

To summarize, the signatures of the *semantic domains* are:

$$LStore = (Var \mapsto Val) \times (Sig \mapsto \mathcal{P}(\mathcal{Z} \times Val_{\perp}))$$
$$SStore = (SVar \mapsto [Val \times (I \cup \{\perp, \top\}) \times \mathcal{P}(\{r, w\})])$$
$$PPStat = \mathcal{P}(I)$$

Note that $\mathcal{P}$ stands for the powerset operator, and parentheses and square brackets have been used interchangeably, to enhance readability.

4.1.1. Handling of shared variables for portability. We need to ensure that the value bound to each shared variable in every simulation cycle is well-defined (unique) in spite of the non-deterministic execution of the processes. To this end we define a deterministic finite state automaton (DFA) that keeps track of accesses to a shared variable.

A DFA is a 5-tuple [10]: (**Q**,Ω, Γ, **F**, q_0), where **Q** is the set of possible states, Ω is the alphabet, Γ is the transition function (Γ : **Q** × Ω ↦ **Q**), **F** is the set of accepting states (⊆ **Q**), and q_0 is the initial state (∈ **Q**). We customize these sets for the problem at hand as follows:

- $\mathbf{Q} = Val \times (I \cup \{\perp, \top\}) \times \mathcal{P}(\{r, w\})$.
  This set corresponds to the "states" of a shared variable. Note that I is *finite*, but V*al* is *infinite*. However, for our purposes, we make the simplifying but realistic assumption that V*al* 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, which specifies the change to a shared variable state as a result of an access, is given below (where $v, u \in Val$, $i, j \in$ I and r/w for read/write):

$$\langle v, \perp, \emptyset\rangle \xmapsto{i} \langle v, i, \{r\}\rangle \qquad \langle v, \perp, \emptyset\rangle \xmapsto{\langle i,u\rangle} \langle u, i, \{w\}\rangle$$
$$\langle v, i, \{r\}\rangle \xmapsto{i} \langle v, i, \{r\}\rangle \qquad \langle v, i, \{r\}\rangle \xmapsto{j} \langle v, \top, \{r\}\rangle \quad \text{if } i \neq j$$
$$\langle v, i, \{r\}\rangle \xmapsto{\langle i,u\rangle} \langle u, i, \{r, w\}\rangle \qquad \langle v, i, \{r\}\rangle \xmapsto{\langle j,u\rangle} \langle u, \top, \{r, w\}\rangle \quad \text{if } i \neq j$$
$$\langle v, i, \{w\}\rangle \xmapsto{i} \langle v, i, \{r, w\}\rangle \qquad \langle v, i, \{w\}\rangle \xmapsto{j} \langle v, \top, \{r, w\}\rangle \quad \text{if } i \neq j$$
$$\langle v, i, \{w\}\rangle \xmapsto{\langle i,v\rangle} \langle v, i, \{w\}\rangle \qquad \langle v, i, \{w\}\rangle \xmapsto{\langle i,u\rangle} \langle u, i, \{r, w\}\rangle \quad \text{if } u \neq v$$
$$\langle v, i, \{w\}\rangle \xmapsto{\langle j,v\rangle} \langle v, \top, \{w\}\rangle \quad \text{if } i \neq j \qquad \langle v, i, \{w\}\rangle \xmapsto{\langle j,u\rangle} \langle u, \top, \{r, w\}\rangle \quad \text{if } i \neq j \wedge u \neq v$$

$$\langle v, i, \{r, w\}\rangle \xrightarrow{i} \langle v, i, \{r, w\}\rangle \qquad \langle v, i, \{r, w\}\rangle \xrightarrow{j} \langle v, \top, \{r, w\}\rangle \quad \text{if } i \neq j$$

$$\langle v, i, \{r, w\}\rangle \xrightarrow{\langle i,u\rangle} \langle u, i, \{r, w\}\rangle \qquad \langle v, i, \{r, w\}\rangle \xrightarrow{\langle j,u\rangle} \langle u, \top, \{r, w\}\rangle \quad \text{if } i \neq j$$

$$\langle v, \top, \{r\}\rangle \xrightarrow{j} \langle v, \top, \{r\}\rangle \qquad \langle v, \top, \{r\}\rangle \xrightarrow{\langle j,u\rangle} \langle u, \top, \{r, w\}\rangle$$

$$\langle v, \top, \{w\}\rangle \xrightarrow{j} \langle v, \top, \{r, w\}\rangle$$

$$\langle v, \top, \{w\}\rangle \xrightarrow{\langle j,v\rangle} \langle v, \top, \{w\}\rangle \qquad \langle v, \top, \{w\}\rangle \xrightarrow{\langle j,u\rangle} \langle u, \top, \{r, w\}\rangle \quad \text{if } u \neq v$$

$$\langle v, \top, \{r, w\}\rangle \xrightarrow{j} \langle v, \top, \{r, w\}\rangle \qquad \langle v, \top, \{r, w\}\rangle \xrightarrow{\langle j,u\rangle} \langle u, \top, \{r, w\}\rangle$$

- $\mathbf{F} = (Val \times \{\bot\} \times \{\emptyset\}) \cup (Val \times I \times \{\{r\}, \{w\}, \{r, w\}\}) \cup (Val \times \{\top\} \times \{\{r\}, \{w\}\})$
  The accepting states are meant to capture "portable" sequences of reads and writes.
- $q_0 = \langle v, \bot, \emptyset\rangle$.
  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. The following lemma summarizes the kinds of reads and writes that lead to dead states and is intended to capture potential causes of ambiguity.

Lemma 4.1. *Every string* (*of read/write actions*) *in the language* $\mathcal{L}(\langle y, \top, \{r, w\}\rangle)$ (*where* $y \in Val$) *contains a substring that matches one of the following regular expressions* (*where*, "*" *(resp. "+") represents Kleene star* (*resp. plus*) *operation, and* $i \neq j$ *and* $u \neq v$):

$$\text{(a) } i\ j^*\ \langle j, v\rangle \qquad \text{(b) } \langle i, v\rangle\ i^*\ (j\ \cup\ \langle j, u\rangle) \qquad \text{(c) } \langle i, v\rangle\ \langle j, v\rangle^+\ (k\ \cup\ \langle k, u\rangle).$$

Proof: This can be verified by considering all the transitions that cause the DFA to enter a dead state for the first time.

(a) The pattern $i\ j^*\ \langle j, v\rangle$ matches strings that lead to $(Val \times \{\top\} \times \{\{r, w\}\})$ via $(Val \times i \times \{\{r\}\})$ or $(Val \times \{\top\} \times \{\{r\}\})$.
(b) The pattern $\langle i, v\rangle\ i^*\ (j\ \cup\ \langle j, u\rangle)$ matches strings that lead to $(Val \times \{\top\} \times \{\{r, w\}\})$ via $(Val \times \{i\} \times \{\{w\}\})$ or $(Val \times \{i\} \times \{\{r, w\}\})$.
(c) The pattern $\langle i, v\rangle\ \langle j, v\rangle^+\ (k\ \cup\ \langle k, u\rangle)$ matches strings that lead to $(Val \times \{\top\} \times \{\{r, w\}\})$ via $(Val \times \{\top\} \times \{\{w\}\})$. □

Let $?_{Val} \in Val$ and assume that multiple occurrences of $?_{Val}$ in a string can have different values.

Lemma 4.2. *Every string* (*of read/write actions*) *in the language of the DFA and the resulting value of the shared variable satisfies one of the following conditions*:
(a) *Every action in the string is a read action, that is, it is in I, and 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* $\langle i, ?_{Val}\rangle$, *and 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* $I \times \{\{v\}\}$, *and the final value of the shared variable is the value written.*

Proof: The result follows straightforwardly, using induction on the length of the string, by considering the labels on the paths from the start state to the final states in the DFA.

(a) Every action in the string that takes the DFA from the start state to $\langle v, i, \{r\}\rangle$ and $\langle v, \top, \{r\}\rangle$ is a read action because the relevant transitions are:

$$\langle v, \bot, \emptyset\rangle \overset{i}{\longmapsto} \langle v, i, \{r\}\rangle \quad \langle v, \top, \{r\}\rangle \overset{j}{\longmapsto} \langle v, \top, \{r\}\rangle$$
$$\langle v, i, \{r\}\rangle \overset{i}{\longmapsto} \langle v, i, \{r\}\rangle \quad \langle v, i, \{r\}\rangle \overset{j}{\longmapsto} \langle v, \top, \{r\}\rangle \quad \text{if } i \neq j$$

(b) Every action in the string that takes the DFA from the start state to $\langle u, i, \{w\}\rangle$ and $\langle u, i, \{r, w\}\rangle$ has the same process index because the relevant transitions are:

$$\langle v, \bot, \emptyset\rangle \overset{i}{\longmapsto} \langle v, i, \{r\}\rangle \qquad \langle v, \bot, \emptyset\rangle \overset{\langle i,u\rangle}{\longmapsto} \langle u, i, \{w\}\rangle$$
$$\langle v, i, \{r\}\rangle \overset{i}{\longmapsto} \langle v, i, \{r\}\rangle \qquad \langle v, i, \{r\}\rangle \overset{\langle i,u\rangle}{\longmapsto} \langle u, i, \{r, w\}\rangle$$
$$\langle v, i, \{w\}\rangle \overset{i}{\longmapsto} \langle v, i, \{r, w\}\rangle \qquad \langle v, i, \{w\}\rangle \overset{\langle i,v\rangle}{\longmapsto} \langle v, i, \{w\}\rangle$$
$$\langle v, i, \{w\}\rangle \overset{\langle i,u\rangle}{\longmapsto} \langle u, i, \{r, w\}\rangle \quad \text{if } u \neq v$$
$$\langle v, i, \{r, w\}\rangle \overset{i}{\longmapsto} \langle v, i, \{r, w\}\rangle \quad \langle v, i, \{r, w\}\rangle \overset{\langle i,u\rangle}{\longmapsto} \langle u, i, \{r, w\}\rangle$$

(c) Every action in the string that takes the DFA from the start state to $\langle u, \top, \{w\}\rangle$ has the same value component because the relevant transitions are:

$$\langle v, \bot, \emptyset\rangle \overset{\langle i,u\rangle}{\longmapsto} \langle u, i, \{w\}\rangle \quad \langle u, \top, \{w\}\rangle \overset{\langle j,u\rangle}{\longmapsto} \langle u, \top, \{w\}\rangle$$
$$\langle u, i, \{w\}\rangle \overset{\langle i,u\rangle}{\longmapsto} \langle u, i, \{w\}\rangle \quad \langle u, i, \{w\}\rangle \overset{\langle j,u\rangle}{\longmapsto} \langle u, \top, \{w\}\rangle \quad \text{if } i \neq j$$

For the start state, which is also an accepting/final state, the conditions hold trivially.

□

Let *size*(*rs*) return the cardinality of the set of indices in the string of reads *rs*. (*size*(*ikjjij*) = | {*i*, *j*, *k*} |= 3.)

Lemma 4.3. *Let* $q, q_1, q_2 \in Q$, *and* $rs_1, rs_2 \in I^*$ *be two strings of reads that are* permutations *of each other. Then, the relation* $(q \overset{rs_1}{\longmapsto}_* q_1 \wedge q \overset{rs_2}{\longmapsto}_* q_2 \Rightarrow q_1 = q_2)$ *holds.*

Proof: We consider two cases: (a) $size(rs_1) < 1$. Trivial. (b) $size(rs_1) > 1$. For this case, observe that a read-transition has one of the following forms:

$$\langle v, \bot, \emptyset\rangle \overset{i}{\longmapsto} \langle v, i, \{r\}\rangle \qquad \langle v, \top, A\rangle \overset{i}{\longmapsto} \langle v, \top, A \cup \{r\}\rangle$$

$$\langle v, i, A\rangle \overset{i}{\longmapsto} \langle v, i, A \cup \{r\}\rangle \quad \langle v, i, A\rangle \overset{j}{\longmapsto} \langle v, \top, A \cup \{r\}\rangle \quad \texttt{if } i \neq j$$

Furthermore, for all computations originating from the start state, the second component of the state is i if the string $rs_1, rs_2 \in i^*$, otherwise it is $\top$. □

4.1.2. Extending the DFA. The reads/writes generated by the following two VHDL-93 descriptions (with the shared variables initialized to 0) drive the DFA corresponding to the shared variable *sx* to a (non-accepting) dead state (in every simulation cycle):

- `while true do` (*sx := sx;* `wait for` *1 ns;*)
  ‖
  `while true do` (*sx := 0;* `wait for` *1 ns;*)
- $\|_I$ `while true do` (*sx := sx;* `wait for` *1 ns;*)

This is because, in general, a read and a write from two different processes in the same cycle is regarded as potentially ambiguous. However, it is obvious that these VHDL-93 descriptions can be given unique meaning as they do not change the values of any shared variable. In order to accomodate these "portable" programs, we modify the DFA to accept strings of reads/writes that leave the shared variable value unaltered. A new type of access, denoted as s, is introduced, to record the fact that the shared variable value is unchanged. (The w-type access now signifies writing a different value.)

The new DFA $= (\mathbf{Q}, \Omega, \Gamma, \mathbf{F}, q_0)$ is obtained by modifying the old DFA $= (\mathbf{Q}', \Omega', \Gamma', \mathbf{F}', q_0')$ as follows:

- The new set of states $\mathbf{Q} = \mathbf{Q}' \cup (Val \times (I \cup \{\top\}) \times \{\{s\}\})$.
- The new transition function Γ is obtained from old Γ' as follows:
  - The write-transitions out of $\langle v, \bot, \emptyset\rangle$, $\langle v, i, \{r\}\rangle$ and $\langle v, \top, \{r\}\rangle$ are split.

$$\langle v, \bot, \emptyset\rangle \overset{\langle i,v\rangle}{\longmapsto} \langle v, i, \{s\}\rangle \qquad \langle v, \bot, \emptyset\rangle \overset{\langle i,u\rangle}{\longmapsto} \langle u, i, \{w\}\rangle \quad \text{if } u \neq v$$

$$\langle v, i, \{r\}\rangle \overset{\langle i,v\rangle}{\longmapsto} \langle v, i, \{s\}\rangle \qquad \langle v, i, \{r\}\rangle \overset{\langle j,v\rangle}{\longmapsto} \langle v, \top, \{s\}\rangle \quad \text{if } i \neq j$$

$$\langle v, i, \{r\}\rangle \overset{\langle i,u\rangle}{\longmapsto} \langle u, i, \{r, w\}\rangle \quad \text{if } u \neq v \qquad \langle v, i, \{r\}\rangle \overset{\langle j,u\rangle}{\longmapsto} \langle u, \top, \{r, w\}\rangle \quad \text{if } u \neq v \wedge i \neq j$$

$$\langle v, \top, \{r\}\rangle \overset{\langle j,v\rangle}{\longmapsto} \langle v, \top, \{s\}\rangle \qquad \langle v, \top, \{r\}\rangle \overset{\langle j,u\rangle}{\longmapsto} \langle u, \top, \{r, w\}\rangle \quad \text{if } u \neq v$$

Note that it does not make sense to split the write-transitions out of $\langle v, i, \{w\}\rangle$, $\langle v, i, \{r, w\}\rangle$, $\langle v, \top, \{w\}\rangle$ and $\langle v, \top, \{r, w\}\rangle$ because a different value has already been written.

– The transitions for $\langle v, i, \{s\}\rangle$, and $\langle v, \top, \{s\}\rangle$ are added:

$$\langle v, i, \{s\}\rangle \stackrel{i}{\longmapsto} \langle v, i, \{s\}\rangle \qquad \langle v, i, \{s\}\rangle \stackrel{j}{\longmapsto} \langle v, \top, \{s\}\rangle \qquad \text{if } i \neq j$$

$$\langle v, i, \{s\}\rangle \stackrel{\langle i,v\rangle}{\longmapsto} \langle v, i, \{s\}\rangle \qquad \langle v, i, \{s\}\rangle \stackrel{\langle j,v\rangle}{\longmapsto} \langle v, \top, \{s\}\rangle \qquad \text{if } i \neq j$$

$$\langle v, i, \{s\}\rangle \stackrel{\langle i,u\rangle}{\longmapsto} \langle u, i, \{r, w\}\rangle \quad \text{if } u \neq v \qquad \langle v, i, \{s\}\rangle \stackrel{\langle j,u\rangle}{\longmapsto} \langle u, \top, \{r, w\}\rangle \quad \text{if } u \neq v \wedge i \neq j$$

$$\langle v, \top, \{s\}\rangle \stackrel{i}{\longmapsto} \langle v, \top, \{s\}\rangle \qquad \langle v, \top, \{s\}\rangle \stackrel{\langle i,v\rangle}{\longmapsto} \langle v, \top, \{s\}\rangle$$

$$\langle v, \top, \{s\}\rangle \stackrel{\langle i,u\rangle}{\longmapsto} \langle u, \top, \{r, w\}\rangle$$

– All the remaining transitions in Γ' are included in Γ as is.

- The new set of final states $\mathbf{F} = \mathbf{F}' \cup (Val \times (I \cup \{\top\}) \times \{\{s\}\})$.

Lemma 4.4. *Every action in the string* (*of read/write actions*) *that takes the DFA from the start state* $q_0 = \langle v, \perp, \emptyset\rangle$ *to the states* $\langle v, i, \{s\}\rangle$ *or* $\langle v, \top, \{s\}\rangle$ *is either a read action or a write action with the value component equal to* v. *Furthermore*, *the value of the shared variable remains unaltered.*

Proof: The result follows straightforwardly by considering the labels on the sequence of transitions from the start state to the states $\langle v, i, \{s\}\rangle$ and $\langle v, \top, \{s\}\rangle$ in the DFA given above (Section 4.1.2). □

Given two strings of reads and writes, such as $rws_1, rws_2 \in (I \cup I \times Val)^*$, that are *permutations* of each other, it is *not* always the case that $(q_0 \stackrel{rws_1}{\longmapsto}_* q_1 \wedge q_0 \stackrel{rws_2}{\longmapsto}_* q_2 \Rightarrow q_1 = q_2)$ holds. This is because, different permutations of writes from a single process can result in different final values for the shared variable (cf. Lemma 4.3). Fortunately, we can salvage a weaker result (see Lemma 4.5) by suitably restricting allowable permutations, which attests to the soundness of the DFA.

Definition 4.1. An *interleaving* of a set of action strings $\{s_1, s_2, \ldots, s_n\}$ is a permutation of the concatenation of $s_1, s_2, \ldots, s_n$ that preserves the ordering of the actions in each string s_i, for $1 \leq i \leq n$.

Consequently, s_i results from deleting actions contributed by all s_j's such that $i \neq j$ from an interleaving of $\{s_1, s_2, \ldots, s_n\}$. In the sequel, we abuse the terminology and call two valid interleavings of the same set of strings, interleavings of each other. For instance, the read/write string $\langle i, v\rangle j\langle j, v\rangle\langle i, u\rangle$ obtained from the set $\{\langle i, v\rangle\langle i, u\rangle, j\langle j, v\rangle\}$ is an interleaving of $\langle i, v\rangle j\langle i, u\rangle\langle j, v\rangle$, but is not an interleaving of $\langle i, u\rangle\langle j, v\rangle j\langle i, v\rangle$.

Lemma 4.5. *Let* $q_1, q_2 \in Q$, *and* $rws_1, rws_2 \in (I \cup I \times Val)^*$ *be two strings of reads and writes that are* interleavings *of each other. Then, the following relation holds*:
$q_0 \stackrel{rws_1}{\longmapsto}_* q_1 \wedge q_0 \stackrel{rws_2}{\longmapsto}_* q_2 \Rightarrow (q_1 \in \mathbf{F} \text{ iff } q_2 \in \mathbf{F})$.

Proof Sketch: The proof is simple but tedious and involves extensive case analysis. So we just sketch our approach. We partition strings in $(I \cup I \times Val)^*$ in such a way that, for each partition, the result will follow straightforwardly by observing the relevant transitions in the DFA. The partitions we consider are:

1. Strings that do not modify the value of the shared variable. In this case, $q_1, q_2 \in \mathbf{F}$.
2. Strings that contain a "modifying" write, and furthermore, all reads/writes are from a single process. In this case, $q_1, q_2 \in \mathbf{F}$.
3. Strings that contain a "modifying" write, and a read from another process.
   These are further analysed based on the relative order of the two actions in the string. In all cases, $q_1, q_2 \notin \mathbf{F}$.
4. Strings that contain a "modifying" write and a write from another process.
   These are further analysed based on the relative order of the two actions in the string. In case all the writes have the same value component, $q_1, q_2 \in \mathbf{F}$. In all other cases, $q_1, q_2 \notin \mathbf{F}$. □

Finally, the signature of *SStore* is modified to reflect the new access type:

$$SStore = (SVar \mapsto [Val \times (I \cup \{\bot, \top\}) \times (\mathcal{P}(\{r, w\}) \cup \{\{s\}\})])$$

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

$$\begin{aligned} Store &= \mathcal{P}(LStore) \times SStore \times PP\,Stat \\ \sigma, \sigma_i &\in LStore \quad \Sigma, \Sigma_i \in \mathcal{P}(LStore) \\ \psi &\in SStore \qquad \xi \in PP\,Stat \end{aligned}$$

Two functions—$\mathcal{T}, \mathcal{U} : Store \mapsto Store$—are defined to advance (global) time and delta time respectively [8]. 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 [8] since it has 1 in place of 2, and as seen later, $\sigma_i(s)(1)$ is always undefined when $\mathcal{T}$ is applied.
- For shared variables: $\mathcal{T}(\psi)(sx) = \langle v, \bot, \emptyset\rangle$, where $\psi(sx) = \langle v, i, A\rangle$.
- For the status of the postponed-processes: $\mathcal{T}(\xi) = \emptyset$.

A signal *s* is *active* if $\exists \sigma_i \in \Sigma_I, v \in Val_\bot : \langle 1, v\rangle \in \sigma_i(s)$. 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 variables, shared variables, and inactive signals.

- For shared variables: $\mathcal{U}(\psi)(sx) = \langle v, \bot, \emptyset\rangle$, where $\psi(sx) = \langle v, i, A\rangle$.
- 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 [8]:

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

Here, $\{\{\cdot\}\}$ 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* for the various language constructs are:

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

An expression is evaluated with respect to the local and shared stores 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*. Also, $\psi[sx \mapsto st] = (\lambda sy.\texttt{if } sx \equiv sy \texttt{ then } st \texttt{ else } \psi(sy))$.

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

The change in the state of the shared variable is dictated by the corresponding transition in the DFA. The value of the delayed expression is required to be non-negative. (There is a minor error in [8] here.) $s'\texttt{delayed}(0\ ns) \neq s$ during any simulation cycle where there is a change in the value of *s*. (see Section 14.1 in the LRM [11].) For correct handling of `delayed`-attribute we also need to store the previous value of each signal in the *LStore*.

Theorem 4.1. *The meaning of an expression is independent of the order of evaluation of its subexpressions.*

Proof: 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. The result follows from Lemma 4.3 and structural induction. □

Consequently, the semantics does not change if we altered the last rule as follows:

$$\mathcal{E}\,[\![e_i \ bop \ e_i']\!]\,\langle\sigma,\psi\rangle = \langle v \ bop \ v',\psi''\rangle \quad \texttt{if}\ \ \mathcal{E}\,[\![e_i']\!]\,\langle\sigma,\psi\rangle = \langle v',\psi'\rangle \ \texttt{and}\ \ \mathcal{E}\,[\![e_i]\!]\,\langle\sigma,\psi'\rangle = \langle v,\psi''\rangle$$

4.3. Semantics of statements

The semantic rules for all but the signal assignment statement and the wait statement are more or less standard. Recall that, $\sigma[x \mapsto v] = (\lambda y.\ \texttt{if}\ x \equiv y\ \texttt{then}\ v\ \texttt{else}\ \sigma(y))$.

$$\frac{\textit{true}}{\langle\sigma,\psi,\textit{null};ss\rangle \to_{ss} \langle\sigma,\psi,ss\rangle}$$

$$\frac{\mathcal{E}\,[\![e]\!]\,\langle\sigma,\psi\rangle = \langle v,\psi'\rangle}{\langle\sigma,\psi,x := e;ss\rangle \to_{ss} \langle\sigma[x \mapsto v],\psi',ss\rangle}$$

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

$$\frac{\langle\sigma,\psi,ss'\rangle \to_{ss} \langle\sigma',\psi',ss''\rangle}{\langle\sigma,\psi,ss';ss\rangle \to_{ss} \langle\sigma',\psi',ss'';ss\rangle}$$

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

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

$$\frac{\mathcal{E}\,[\![e]\!]\,\langle\sigma,\psi\rangle = \langle\mathsf{true},\psi'\rangle}{\langle\sigma,\psi,\texttt{if}\ e\ \texttt{then}\ ss'\ \texttt{else}\ ss''\rangle \to_{ss} \langle\sigma,\psi',ss'\rangle}$$

$$\frac{\mathcal{E}\,[\![e]\!]\,\langle\sigma,\psi\rangle = \langle\mathsf{false},\psi'\rangle}{\langle\sigma,\psi,\texttt{if}\ e\ \texttt{then}\ ss'\ \texttt{else}\ ss''\rangle \to_{ss} \langle\sigma,\psi',ss''\rangle}$$

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. This corresponds to the *transport delay* interpretation with preemptive scheduling and is formalized similarly to Goossens [8] as follows:

$$\textit{update}(\sigma,s,v,t) = \sigma[s \mapsto (\sigma(s)\backslash\{\langle n,\sigma(s)(n)\rangle \mid n > t\}) \cup \{\langle t+1,v\rangle\}].$$

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

From Theorem 4.1, it also follows that the order of evaluation of the expressions e and et does not effect the final value of the signal s.

An alternative semantics for the signal assignment statement, which happens to be the default in VHDL, uses the *inertial delay* interpretation. Inertial delay differs from transport delay in that it requires retaining the sequence of time-value pairs that immediately precede the new transaction and that have the same value component. (See Section 8.4 of the LRM [11] for details. Also note that there is minor discrepancy between the LRM and Van Tassel's formalization [Chapter 3, [7]] in that the latter retains all transactions with the same value component, not just the contiguous ones.)

This interpretation can be formalized by modifying *update* as follows, where the pending transactions involving signal s, from time *stableFrom(σ,s,v,t)* to time t, have the same value component v.

$$\begin{aligned} \mathit{stableFrom}(\sigma, s, v, t) = \mathit{min}\{t' \mid t' > 0 \wedge \forall \tau \in \mathit{dom}(\sigma(s)) : (t' \leq \tau \leq t) \\ \Rightarrow (\sigma(s)(\tau) = v)\} \\ \mathit{else}\ t+1 \\ \mathit{update}(\sigma, s, v, t) = \sigma[s \mapsto (\sigma(s) \backslash \{\langle n, \sigma(s)(n)\rangle \mid n > t\}) \\ \cup \{\langle n, v\rangle \mid \mathit{stableFrom}(\sigma, s, t, v) \\ \leq n \leq t+1\}]. \end{aligned}$$

Recall that *x else y* means "if *x* is defined then *x* else *y*", and *min*{ } is undefined.

4.4. *Semantics of processes and programs*

The sematic 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 $\|_{i\in I}\ \langle \sigma_i, \psi, \xi, ss_i\rangle$ for $\langle\langle\|_{i\in I}\ \sigma_i, \psi, \xi\rangle, \|_I\ ss_i\rangle$.)

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 (or 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:

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

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

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

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

A delta cycle (that is, the simulation cycle where the global time is not advanced) is initiated when a process can resume, and a non-postponed process is executed if it is either timed-out or if the condition in the `wait`-statement holds.
Rule 3:

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

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

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

$$run(\sigma_i, \sigma_i', \psi, te_i, be_i) \equiv (timeout(\sigma_i, \psi, te_i) \vee [\exists s \in S_i : event(\sigma_i, \sigma_i', s) \wedge fst(\mathcal{E}\ [\![be_i]\!]\ \langle \sigma_i', \psi\rangle)])$$
$$event(\sigma, \sigma', s) \equiv \sigma(s)(0) \neq \sigma'(s)(0)$$

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. Note that, for a process to be able to run, it is *necessary but not sufficient* that the condition in the `wait`-statement hold. 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 [11]). It is an error if the execution of a postponed process initiates another delta-cycle.

Rule 4:

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

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 now define *portability* formally. Let $\rightarrow_{pgm}^*$ be the reflexive transitive closure of $\rightarrow_{pgm}$, and $(\mathbf{Q}, \Omega, \Gamma, \mathbf{F}, q_0)$ be the DFA described in Sections 4.1.1 and 4.1.2.

Definition 5.1. A program $(\|_{i\in I}$ `while` true `do` $ss_i)$ is a *portable* VHDL-93 description if, for every computation of the form

$$(\|_{i\in I} \ \langle \sigma_i, \psi, \xi, \texttt{while true do } ss_i\rangle \rightarrow_{pgm}^* \|_{i\in I} \ \langle \sigma_i', \psi', \xi', ss_i'\rangle),$$

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

Lemma 4.2 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*. Similarly, Lemma 4.4 implies permitting executions that do not alter the value of the shared variable.

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

Theorem 5.1. *A process that does not contain a* wait*-statement loops forever.*

Proof: If *ss* is a sequence of statements that contains *n* simpler statements (none of which is a `wait`-statement), then the following computations require at least *n*-steps.

$$\begin{array}{c}\langle \sigma, \psi, ss; ss'\rangle \rightarrow_{ss} \langle \sigma', \psi', ss'\rangle \\ \langle \sigma, \psi, \xi, \texttt{while true do } ss\rangle \rightarrow_{pgm}^* \langle \sigma', \psi', \xi, \texttt{while true do } ss\rangle\end{array}$$

Furthermore, the latter computation repeats all over again, leading to infinite loop.

Theorem 5.2. *The semantics of expressions* $\mathcal{E}$ *(resp. statements* $\rightarrow_{ss}$*) is* deterministic.

Proof: We need to show that

$$\begin{array}{l}\mathcal{E}\ [\![e_i]\!]\ \langle \sigma, \psi\rangle = \langle v', \psi'\rangle \wedge \mathcal{E}\ [\![e_i]\!]\ \langle \sigma, \psi\rangle = \langle v'', \psi''\rangle \Rightarrow (v' = v'') \wedge (\psi' = \psi'') \\ \langle \sigma, \psi, ss\rangle \rightarrow_{ss} \langle \sigma', \psi', ss'\rangle \wedge \langle \sigma, \psi, ss\rangle \rightarrow_{ss} \langle \sigma'', \psi'', ss'\rangle \\ \quad \Rightarrow (\sigma' = \sigma'') \wedge (\psi' = \psi'')\end{array}$$

Given Theorem 4.1, this result follows from the theory of sequential programs [9, 21]. □

Theorem 5.3. *The statement* `wait on` $\emptyset$ `for` ∞ `until` true; *causes the enclosing process to suspend forever.*

Proof: We need to show that

$$\langle \sigma, \psi, \xi, ws_i[\infty, \text{true}]; ss\rangle \rightarrow^*_{pgm} \langle \sigma, \psi, \xi, ws_i[\infty, \text{true}]; ss\rangle$$

This is a consequence of the definition of *run*-predicate used in defining the semantics of the `wait`-statement. In particular, it requires the time-out expression to become zero *or* an event to occur on a signal in the condition expression. □

Recall also that the statements `wait;`, `wait for` ∞`;`, `wait for` ∞ `until` true; and `wait on` $\emptyset$ `for` ∞ `until` true; are all syntactic variants of each other, and hence, semantically equivalent.

We now show that portable VHDL-93 descriptions can be given a unique meaning. (Recall that Lemma 4.5 assures us of the soundness of the DFA for shared variables.)

Theorem 5.4. *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: Effectively, we need to show that, if $\|_{i\in I}\langle \sigma_i, \psi, \xi, ws_i[te_i, be_i]; ss_i; ws'_i[te'_i, be'_i]; ss'_i\rangle \rightarrow^*_{pgm} \|_{i\in I}\ \langle \sigma'_i, \psi', \xi', ws'_i[te'_i, be'_i]; ss'_i\rangle$ 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 5.1, Lemma 4.2 and Lemma 4.4. The application of *Rule* 2 and *Rule* 3 for the `wait`-statement define a unique transformation because the resolution functions f_s and the time increment functions $\mathcal{T}$ and $\mathcal{U}$ are all one to one and total. So the result follows by induction on the length of the computation (number of simulation cycles). □

Theorem 5.5. *The portability condition given in Definition* 5.1 *is* non-local.

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 separately:

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;*)

Running with **P1**, **PS** is portable; while running with **P2**, **PS** is not portable. □

As a consequence of this non-locality, it is not possible to *incrementally* check VHDL-93 descriptions for portability. In other words, portable programs cannot be composed.

Theorem 5.6. *The portability condition given in Definition* 5.1 *is* sufficient but not necessary *for VHDL-*93 *descriptions to have a unique meaning.*

Proof: In general, arbitrary writes from two processes to a shared variable can destroy portability. However, there exist *trivial* descriptions such as

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

that have a unique meaning, but violate the portability condition. □

Even though the portability condition seems conservative, we now show that it is futile to search for a necessary and sufficient condition for portability that can be *mechanized*.

Theorem 5.7. *Given a VHDL-*93 *description*, *it is not possible to determine* statically (*that is*, *at compile time*) *whether or not a VHDL-*93 *description 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 done statically. The program **PS** and the shared variable *sflag* given in the proof of Theorem 5.5 exemplifies this situation.

The negative result holds even when all the variables, shared variables and signals are completely defined in the program. This is because, the test for portability can be reduced to the problem of 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, and let *Func1* and *Func2* abbreviate the effect of the code that computes values for *sx* from *x1* and *x2* respectively. The above program is portable if and only if the value written into *sx* by the two processes is identical in every cycle. That is, *Func1* and *Func2* stand for the same function. However, since the equivalence problem for Turing-complete languages is undecidable [21], the portability cannot be tested algorithmically. □

In order to detect lack of portability of a VHDL-93 description at run-time, the simulator can be augmented with additional information specified in the DFA described in Sections 4.1.1 and 4.1.2. One can view this information as a new implementation of the abstract data type *shared variable.*

So far we have discussed an approach that can demonstrate non- portability of a VHDL-93 description. Now we focus on developing a sufficient condition for determining portability of a VHDL-93 description, without simulating it for all possible sequentializations of

read/write accesses to the shared variables. In effect, we exploit the ability of the DFA to implicitly construct counterexamples to portability.

Theorem 5.8. *A VHDL-*93 *description is* portable *if, for all values of the* free *variables*, free *shared variables*, *and* free *signals*, *and for all* $k \in \mathcal{N}$, *the computation sequence*

$$\|_{i\in I}\ \langle \sigma_i, \psi, \xi, \texttt{while true do } ss_i\rangle \ \rightarrow^{k}_{pgm}\ \|_{i\in I}\ \langle \sigma_i^k, \psi^k, \xi^k, ss_i^k\rangle)$$

implies $(\forall i \in I : \sigma_i^k \in \mathbf{F})$.

Proof: We consider computation sequences for all possible values for all *free* identifiers to ensure that all fragments of the VHDL-93 description are exercised. (In other words, it generates all possible straight-line programs from the conditionals and the loops.) From the assumption of this theorem and Lemma 4.5, we know that all interleavings of read/write actions lead to an accepting state in **F**. Thus, according to the Definition 5.1, the VHDL-93 description is portable (and from Theorem 5.4, it admits a unique meaning). □

6. Conclusions

The designers of VHDL-93 extended VHDL-87 by introducing shared variables and postponed processes into the language. Here, we developed 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. VHDL-93 LRM stipulates that a VHDL-93 description that generates different behaviors on different simulators is erroneous. In this paper we explored causes of ambiguity through examples and later proposed sufficient conditions for a VHDL-93 behavioral description with shared variables and postponed processes to be portable, that is, to have a 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.

The above results further support the appropriateness of *protected types* to deal with shared variables as discussed in [20]. Willis [20] uses Hoare's *monitors* as the basis for implementing shared variable mutual exclusion semantics, thereby enabling expression of *algorithmic nondeterminism*.

References

1. M. Belhadj, R. McConnell, and P. Le Guernic, "A framework for macro- and micro-time to model VHDL attributes," *European Design Automation Conference with EURO-VHDL*, Sept. 1993, pp. 520–525.
2. J. Bhasker, *A VHDL Primer*, 2nd edn., Prentice Hall, Inc., Englewood Cliffs, NJ, 1995.
3. P. Breuer, L. Sanchez, and C.D. Kloos, "A simple denotational semantics, proof theory and validation condition generator for unit delay VHDL," *Formal Methods in System Design*, Vol. 7, No. 1/2, 1995, pp. 27–51.
4. W. Damm, B. Josko, and R. Schlör, "A net-based semantics for VHDL," *European Design Automation Conference with EURO-VHDL*, Sept. 1993, pp. 520–525.

5. D. Deharbe and D. Borrione, "Semantics of a verification-oriented subset of VHDL," In *Correct Hardware Design and Verification Methods*, CHARME, LNCS, Vol. 987, Springer Verlag, Oct. 1995, pp. 293–310.
6. E. Encrenaz, "A symbolic relation for a subset of VHDL-87 descriptions and its application to symbolic model checking," In *Correct Hardware Design and Verification Methods*, CHARME, LNCS, Vol. 987, Springer Verlag, Oct. 1995, pp. 328–342.
7. C.D. Kloos and P. Breuer (Eds.), *Formal Semantics of VHDL*, Vol. 307, Kluwer Academic Publishers, Boston, March 1995.
8. K.G.W. Goossens, "Reasoning about VHDL using operational and observational semantics," In *Correct Hardware Design and Verification Methods*, CHARME, LNCS, Vol. 987, Springer Verlag, Oct. 1995, pp. 311–327.
9. M. Hennessy, *The Semantics of Programming Languages: An Elementary Introduction using Structural Operational Semantics*, John Wiley & Sons, New York, 1990.
10. J. Hopcroft and J. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley Co., Reading, MA 1979.
11. Institute of Electrical and Electronics Engineers, 345 East 47th Street, New York, USA. *IEEE Standard VHDL Language Reference Manual, Std 1076-1993*, 1993.
12. N. Madrid, P. Breuer, and C.D. Kloos, "A semantic model or VHDL-AMS," In *Correct Hardware Design and Verification Methods*, CHARME, Sept. 1997, pp. 106–123.
13. S. Olcoz and J. Colom, "Toward a formal semantics of IEEE Std. VHDL 1076," in *European Design Automation Conference with EURO-VHDL*, Sept. 1993, pp. 526–531.
14. H. Sasaki, K. Mizushima, and T. Sasaki, "Semantic validation of VHDL-AMS by an abstract state machine," in *IEEE/VIUF International Workshop on Behavioral Modeling and Simulation*, Oct. 1997, pp. 61–68.
15. T. Sasaki, H. Sasaki, and K. Mizushima, "Semantic analysis of VHDL-AMS by attribute grammar," in *Forum on Design Languages*, Sept. 1998.
16. S. Shankar and J. Slagle, "A polymodal semantics for VHDL," In *Correct Hardware Design and Verification Methods*, CHARME, Sept. 1997, pp. 88–105.
17. K. Thirunarayan and R. Ewing, "Characterizing a portable subset of behavioral VHDL-93," In *Computer Hardware Description Languages and their Applications*, Chapman and Hall, London, April 1997, pp. 97–113.
18. J.P. van Tassel, " Femto-VHDL: The semantics of a subset of VHDL and its embedding in the HOL proof assistant," Ph.D. Dissertation, University of Cambridge, 1993.
19. P.A. Wilsey, "Developing a formal semantic definition of VHDL," in J. Mermet (Ed.), *VHDL for Simulation, Synthesis and Formal Proofs of Hardware*, Kluwer Academic Publishers, Boston, 1992, pp. 243–256.
20. J. Willis, Steve Bailey and Chuck Swart, "Shared variable language change specification," `http://www.vhdl.org/vi/svwg/lcs/lcs.htm`, 1996.
21. G. Winskel, *The Formal Semantics of Programming Languages: An Introduction*, The MIT Press, Cambridge, MA 1993.