Rapporto di Ricerca
SI/RR - 95/06
Aprile/1995

Reasoning About VHDL Using Operational and Observational Semantics

K. G. W. Goossens

Dipartimento di Scienze dell'Informazione
Università degli Studi di Roma *La Sapienza*
via Salaria, 113 - I - 00198 - Roma

Reasoning About VHDL Using Operational and Observational Semantics*

K. G. W. Goossens†
Dipartimento di Scienze dell'Informazione
Università di Roma "La Sapienza"
Roma, Italy
kgg@dsi.uniroma1.it

April 1995

Abstract

We define a Plotkin-style structural operational semantics for a subset of VHDL that includes delta time, zero-delay scheduling and waits, arbitrary wait statements, and (commutative) resolution functions. While most of these features have been dealt with in separation, their combination is intricate. We follow closely the "careful prose" definition of VHDL as given in [9].

We prove a (conditional) monogenicity result for the operational semantics showing that the parallelism present in VHDL is benign. A classification of program behaviours is also given.

While the semantics is of interest, of greater importance is the interpretation of the mature process algebra theory to our particular setting. An adaptation of bisimulation may be constructed but the concept of an *observer*, a process which inspects or acts as a test harness, turns out to be more useful. It leads naturally to a notion of *observational equality* that is a congruence with respect to parallel composition. This important result enables substitution of behaviourally equivalent subprograms without affecting the overall program behaviour. The capability to pass (incapability to fail) a test gives rise to a the *may* (*must*) preorder on processes. These preorders are shown to coincide.

*A shorter version of this report appears as [8].

†This work is supported by the EUROFORM network sponsored by the Human Capital and Mobility programme of the European Community.

Contents

1 Introduction

VHDL is one of the most widely used hardware description languages. The language definition [9, 10] is given in English prose, opening the way to a multitude of formal semantics (see, for example, the recent collection of papers [6] and [1, 2, 7, 14, 15, 16, 17, 18]). As VHDL is a large language most of this research limits itself to subsets of VHDL, often ignoring essential features of the VHDL model of hardware, such as delta time, signal resolution, and the structural hierarchy. In this work also we deal with a fraction of the language, but our intention is to deal with all fundamental behavioural constructs present within one entity. The VHDL subset therefore contains local variables, (possibly resolved) signal assignments (including zero delay signal scheduling), all versions of wait statements, if and while statements, and parallel composition of sequential programs. In Section 3 we present a Plotkin-style structural operational semantics [13] that closely follows the informal VHDL definition. We believe that an operational framework is most natural for VHDL because it is generally understood in the context of a simulation kernel processing a hardware description.

We prove several results of the operational semantics, including a classification of program behaviours (Theorem 5.11). Also in Section 5 we show a (limited) monogenicity result for the operational semantics: despite the presence of parallelism executions are essentially deterministic (Theorem 5.1). At this point we must point out that the semantics for our VHDL subset is based on VHDL87 [9] instead of VHDL93 [10]. The latter includes shared variables that fundamentally change the semantic model of VHDL. In fact, one of the important properties of VHDL87, monogenicity, no longer holds, making system verification more complex, both in theory and practice. (Consult Section 3.1 for further details.)

Building on our formalisation of VHDL we define in Section 8 an observational semantics using the testing theory of [4]. In this framework two programs are considered equal if they pass the same set of tests. An observer is nothing more than the formalisation of a test harness, analysing the output generated in response to particular inputs. The theory thus blends comfortably with the tradition of hardware testing. Alternatively, VHDL models reactive systems that respond to a active environment that can be interpreted as a program or circuit because, essentially, it is defined by changes it generates on input signals. Both views support the identification of observer and observee, leading to a pleasing symmetry and simplicity. The use of an established theory such as testing allows us to import its concepts such as *may* and *must* preorders and the *observation semantics*. We prove

that due to the limited nondeterminism in VHDL may and must preorders coincide. Also, observational equivalence is a congruence with respect to parallel composition (Theorem 8.2). This theorem is important because it allows us to substitute observationally equivalent system components without affecting the overall system behaviour.

2 Definition of the VHDL Subset

Our VHDL subset contains local variables, variable assignment, (possibly resolved) signals, signal assignments (including zero delay signal scheduling), full-blown wait statements, if and while statements, and parallel composition of sequential programs. The abstract syntax is defined as follows:

$$
\begin{array}{lll}
\mathit{pgm} & ::= & \|_{i \in I}\ \mathit{ss} \\
\mathit{ss} & ::= & x\ \texttt{:=}e \mid x \Leftarrow e\ \texttt{after}\ e \mid \texttt{wait on}\ S\ \texttt{for}\ e\ \texttt{until}\ e \mid \texttt{null} \mid \\
 & & \mathit{ss}\texttt{;}\mathit{ss} \mid \texttt{while}\ e\ \texttt{do}\ \mathit{ss} \mid \texttt{if}\ e\ \texttt{then}\ \mathit{ss}\ \texttt{else}\ \mathit{ss} \\
e & ::= & v \mid x \mid e\ \mathit{binop}\ e \mid \mathit{unop}\ e \mid s\texttt{'delayed(}e\texttt{)} \mid \texttt{null}
\end{array}
$$

Binary operators *binop* include $\wedge$, $\vee$, $-$, and $+$; unary operators include $\neg$ and $-$. I is an arbitrary, finite index set for processes and S a finite, possibly empty set of signal names. x is a variable or signal name ($x \in \mathit{Var} \cup \mathit{Sig}$) and v is a value from a given value domain ($v \in \mathit{Val}$); see the following section for *Var*, *Val*, and *Sig*.

Abbreviations

In the following we expect VHDL processes of the form `process` [(S)] `begin` ss `end process` to have been transformed to `while true do` (ss [;`wait on` S]) prior to evaluation in our semantics. Processes are then a special case of sequential programs and we use the terms process and sequential program interchangeably.

In the context of the wait statement, when `on` S, `for` e, or `until` e is omitted, `on` T (T is equal to the set of all signals appearing in the until clause), `for` ∞, or `until true` are to be inserted respectively. [9, Section 8.1]. Similarly, $s \Leftarrow e$ is shorthand for $s \Leftarrow e$ `after 0`. We adopt a single time scale and thus omit suffixes such as `ns`.

3 A Structural Operational Semantics

Regarding the static semantics of our VHDL subset we expect expressions and programs to be well-typed, following the usual VHDL rules.

Before introducing our semantic entities it is helpful to show the relations and semantic functions that constitute our structural operational semantics:

$$\begin{array}{l}\mathcal{E} : e \times \mathit{Store} \to \mathit{Val}_\perp \\ \to_{ss}\text{: } (\mathit{Store} \times ss) \times (\mathit{Store} \times ss) \\ \to_{pgm}\text{: } (\mathcal{P}(\mathit{Store}) \times pgm) \times (\mathcal{P}(\mathit{Store}) \times pgm)\end{array}$$

$\mathcal{E}$ is a semantics for evaluating expressions in a store, $\to_{ss}$ defines how statements evolve together with a store, and $\to_{pgm}$ relates programs and their state (a set of stores) to new programs and state. The latter are relations instead of functions because computations can be nondeterministic. A program and a set of stores is equivalent to a set of sequential programs each with their own store; to emphasise this fact we often regard $\to_{pgm}$ as having type $\mathcal{P}(\mathit{Store} \times ss) \times \mathcal{P}(\mathit{Store} \times ss)$ and write $\|_I <\sigma_i, ss_i>$ for $<\Sigma_I, \Pi_I>$ where $\Sigma_I \equiv \{\sigma_i | i \in I\}$ (a set of stores) and $\Pi_I \equiv \|_I\ ss_i$ (a set of processes).

3.1 Semantic Entities

We take as given a value domain *Val*, which must include natural numbers and booleans, together with appropriate operators $+$, $-$, $\wedge$, $\vee$, and $\neg$. Assuming also a domain *Var* of variables and a domain *Sig* of signals we define:

$$\mathit{Store} = (\mathit{Var} \mapsto \mathit{Val}) \times (\mathit{Sig} \mapsto \mathcal{P}(\mathbb{Z} \times \mathit{Val}_\perp))$$

Signals are mapped onto sets f, which we will frequently interpret as partial functions $f : \mathbb{Z} \to \mathit{Val}_\perp$ with the following intuition: for $n < 0$, $f(n)$ is the value of the signal of n time steps ago; $f(0)$ is the current value of signal s; for $n \geq 0$, $f(n+1)$ is the projected value for s for n time steps into the future. Thus `s` $\Leftarrow$ `e after` n affects $\sigma(s)(n+1)$, and $\sigma(s)(1)$ contains the value scheduled for the next delta cycle. $\sigma(s)$ contains at least $<-\infty, i>$ and $<0, v>$ for initial value i and current value v of signal s. Note that only for $n > 0$ is $<n, \perp>$ a valid pair in $\sigma(s)$; it then encodes a null transaction for time n.

The types which are used together with the, possibly subscripted and

primed, canonical elements are:

$$\begin{aligned} v &\in \mathit{Val} \\ s &\in \mathit{Sig} \\ \sigma &\in \mathit{Store} \\ \Sigma, \Sigma_I &\in \mathcal{P}(\mathit{Store}) \\ \Pi, \Pi_I, \|_I\ ss_i &\in \mathcal{P}(ss), pgm \\ c, <\sigma, ss> &\in \mathit{Store} \times ss \\ C, C_I, \|_I <\sigma_i, ss_i> &\in \mathcal{P}(\mathit{Store} \times ss) \\ C, C_I, <\Sigma_I, \Pi_I> &\in \mathcal{P}(\mathit{Store}) \times pgm \end{aligned}$$

As discussed previously, there is an obvious correspondence between $\mathcal{P}(ss)$ and *pgm*, and $\mathcal{P}(\mathit{Store} \times ss)$ and $\mathcal{P}(\mathit{Store}) \times pgm$. Whether we use $\|_I <\sigma_i, ss_i>$ or $<\Sigma_I, \Pi_I>$ depends on the emphasis we wish to place on the interpretation of the element. We call c a sequential program configuration and C a (program) configuration. We omit the index set I of configurations where it is not relevant.

x is either a variable or a signal. Variable and signal names can be distinguished if necessary, for example by making *Var* and *Sig* disjoint. This allows us to unambiguously write $\sigma(x)$ to obtain a value for either $x \in \mathit{Var}$ or $x \in \mathit{Sig}$ (also implicitly inserting appropriate projections on the ith component π_i). We define $\mathit{dom}_{\mathit{Var}} = \mathit{dom} \circ \pi_1$ and $\mathit{dom}_{\mathit{Sig}} = \mathit{dom} \circ \pi_2$, and extend them to sets of stores: $\mathit{dom}_{\mathit{Sig}}(\Sigma_I) = \bigcup_{i \in I} \mathit{dom}_{\mathit{Sig}}(\sigma_i)$ and $\mathit{dom}_{\mathit{Var}}(\Sigma_I) = \bigcup_{i \in I} \mathit{dom}_{\mathit{Var}}(\sigma_i)$. We write `null` for the bottom element of $\mathit{Val}_\perp$ and leave implicit projections and injections converting between *Val* and $\mathit{Val}_\perp$.

Semantic Functions

Two functions, $\mathcal{T}$ and $\mathcal{U}$, are defined to handle the advance of time (rule 17) and delta time (rule 18). They are best read in conjunction with these rules.

The advance of time $\mathcal{T}$ on a store is defined as follows: variables are unchanged: $\mathcal{T}(\sigma)(x) = \sigma(x)$. For signals we advance time, *i.e.*

$$\mathcal{T}(\sigma)(s) = \{<n-1, v> | <n, v> \in \sigma(s)\} \cup \{<0, \sigma(s)(1)\ \mathit{else}\ \sigma(s)(0)>\}$$

Here x *else* y means "if x is defined then x else y."

A signal s is *active* if $\exists \sigma \in \Sigma_I, v \in \mathit{Val}_\perp . <1, v> \in \sigma(s)$. A set of stores is then updated as follows: $\mathcal{U}(\{\sigma_i | i \in I\}) = \{\sigma'_i | i \in I\}$ where variable entries and quiet (non-active) signals are unchanged: $\sigma'_i(x) = \sigma_i(x)$. For active signals

s the current value[1] is replaced by the value r_s, obtained through the signal resolution function f_s (equal to the identity function for unresolved signals):

$$r_s = f_s\{\!\{v_i \mid \exists i \in I. <1, v_i> \in \sigma_i(s) \wedge v_i \neq \texttt{null}\}\!\}$$
$$\sigma_i'(s) = (\sigma_i(s) \setminus \{<0, \sigma_i(0)>, <1, \sigma_i(1)>\}) \ \cup \ \{<0, r_s>\}$$

The multiset $\{\!\{\cdot\}\!\}$ contains all the values scheduled for signal s for the next delta time, excluding `null` signal assignments. The use of a multiset ensures that identical values are not coalesced ($\{\!\{1,1\}\!\} \neq \{1,1\}$) and that f_s can not depend on any order of its elements. We do not, therefore, model resolution functions that depend on the order of values in their input array, even though the VHDL definition does not forbid these [9, Section 2.4]. Because the simulation model of VHDL was clearly designed to ensure monogenicity (Theorem 5.1) we feel justified restricting resolution functions to be commutative – as they almost always are. Thiedefinition makes use of the partial function interpretation of sets and also undefined value components in defined entries. Thus, both undefined transactions and `null` signal assignments do not contribute to the resolved value. Moreover, resolution functions must deliver a result in Val (not $Val_\perp$).

3.2 Expressions

The expression fragment of the language may be given any suitable semantics. We have used a denotational style.

$$\mathcal{E}[\![v]\!]\sigma = v \tag{1}$$

$$\mathcal{E}[\![\texttt{null}]\!]\sigma = \texttt{null} \tag{2}$$

$$\mathcal{E}[\![unop\ e]\!]\sigma = unop(\mathcal{E}[\![e]\!]\sigma) \tag{3}$$

Unary operators *unop* include $\neg$ and $-$.

$$\mathcal{E}[\![e\ binop\ e']\!]\sigma = (\mathcal{E}[\![e]\!]\sigma)\ binop\ (\mathcal{E}[\![e']\!]\sigma) \tag{4}$$

Binary operators *binop* include $+$, $\vee$ and $\wedge$.

$$\mathcal{E}[\![x]\!]\sigma = \sigma(x) \qquad \text{if } x \in dom_{Var}(\sigma) \tag{5}$$

$$\mathcal{E}[\![x]\!]\sigma = \sigma(x)(0) \qquad \text{if } x \in dom_{Sig}(\sigma) \tag{6}$$

$$\mathcal{E}[\![s\texttt{'delayed(}e\texttt{)}]\!]\sigma = \sigma(s)(n) \qquad \text{largest } n \in dom(\sigma(s)).\, n \leq -\mathcal{E}[\![e]\!]\sigma \tag{7}$$

We disallow `s'delayed(0)` which the last rule incorrectly equates to s. This is easily remedied by the addition of a component in the Var to $Val_\perp$ map to store the penultimate value of s.

[1] The driving and effective values coincide for all our signals.

3.3 Statements

The semantic rules for statements are standard, except for signal assignment and the rule for `null`. These are explained below.

$$\frac{\mathcal{E}[\![e]\!]\sigma = v}{<\sigma, x := e> \rightarrow_{ss} <\sigma[v/x], \texttt{null}>} \tag{8}$$

Read this as "assuming that the expression e evaluates to v in store σ, the store-program pair $<\sigma, ss_1>$ evaluates to the the `null` program in a new memory $\sigma[v/x]$ that is equal to σ except that $\sigma(x)$ now contains v.

To change the value of a signal at a certain time the time-value pair $<t+1, v>$ is added to the event set $\sigma(x)$, deleting all elements scheduled for a later time.

$$\frac{\mathcal{E}[\![e]\!]\sigma = v \qquad \mathcal{E}[\![et]\!]\sigma = t}{<\sigma, x \Leftarrow e \texttt{ after } et> \rightarrow_{ss} <update(\sigma, x, v, t), \texttt{null}>} \tag{9}$$

where $update(\sigma, x, v, t)$ is equal to $(\sigma(x) \backslash \{<n, \sigma(n)> | n > t\}) \cup \{<t+1, v>\}$.

There is no rule for null alone: it is handled in conjunction with the sequencing operator:

$$\frac{}{<\sigma, \texttt{null}; ss> \rightarrow_{ss} <\sigma, ss>} \tag{10}$$

This poses no problems as there is always a next statement because every sequential program is wrapped in a non-terminating while loop. Similarly wait statements are treated together with the parallel composition operator (rules 17 and 18).

$$\frac{<\sigma, ss_1> \rightarrow_{ss} <\sigma', ss'>}{<\sigma, ss_1; ss_2> \rightarrow_{ss} <\sigma', ss'; ss_2>} \tag{11}$$

$$\frac{\mathcal{E}[\![e]\!]\sigma = true}{<\sigma, \texttt{while } e \texttt{ do } ss> \rightarrow_{ss} <\sigma', ss; \texttt{while } e \texttt{ do } ss>} \tag{12}$$

$$\frac{\mathcal{E}[\![e]\!]\sigma = false}{<\sigma, \texttt{while } e \texttt{ do } ss> \rightarrow_{ss} <\sigma, \texttt{null}>} \tag{13}$$

$$\frac{\mathcal{E}[\![e]\!]\sigma = true}{<\sigma, \texttt{if } e \texttt{ then } ss_1 \texttt{ else } ss_2> \rightarrow_{ss} <\sigma, ss_1>} \tag{14}$$

$$\frac{\mathcal{E}[\![e]\!]\sigma = false}{<\sigma, \texttt{if } e \texttt{ then } ss_1 \texttt{ else } ss_2> \rightarrow_{ss} <\sigma, ss_2>} \tag{15}$$

3.4 Programs

So far the semantics has been straightforward. Its complexity lies with the advancement of time. VHDL's timing model is unusual, and process synchronisation and communication is rather convoluted. A VHDL program consists of a set of communicating sequential processes which execute independently of one another (this aspect is handled by rule 16). Global synchronisation occurs when all processes have encountered a wait statement and at this point communication via shared signals is effected. If no signal has changed (the circuit described by the program has settled into a steady state) time is advanced (rule 17). If, on the other hand, some signal remains active relevant processes are reactivated without any change to time (rule 18) – this zero-time increment is also known as delta time. Process resumption involves removing the leading wait statement due to either (i) a change on a signal on which is being waited and the boolean condition holds, or (ii) a time-out specified in the `until` clause . If neither condition is satisfied the process remains suspended.

The following rule allows the processes that make up a program to evolve independently.

$$\frac{<\sigma_j, ss_j> \rightarrow_{ss} <\sigma'_j, ss'_j>}{\|_{I\cup\{j\}} <\sigma_i, ss_i> \rightarrow_{pgm} \|_{I\cup\{j\}} <\sigma'_i, ss'_i>} \tag{16}$$

$\sigma'_i = \sigma_i$ and $ss'_i = ss_i$ for all $i \neq j$, and $\sigma'_i = \sigma'_j$ and $ss'_i = ss'_j$ for $i = j$.

The time increment rule advances time by (i) updating all the stores ($\mathcal{T}(\Sigma_I)$, effectively by subtracting one from all signal function indexes), and (ii) decreasing by one all time-out clauses in wait statements ($\mathcal{E}[\![te_i]\!]\sigma_i - 1$).

$$\frac{\neg resume(<\Sigma_I, \|_I\ ss_i(te_i)>)}{<\Sigma_I, \|_I\ ss_i(te_i)> \rightarrow_{pgm} <\mathcal{T}(\Sigma_I), \|_I\ ss_i(\mathcal{E}[\![te_i]\!]\sigma_i - 1)>} \tag{17}$$

For each $i \in I$ $ss_i(t_i)$ is equal to `(wait on` S_i `for` t_i `until` be_i`;` r_i`)`. *resume* determines if there is a process that must be resumed because it timed out (*timeout*) or contains an active signal (*active*). (*event* is used in rule 18.)

$$\begin{aligned}
resume(<\Sigma_I, \|_I\ ss_i(te_i)>) &\equiv \exists i \in I.\ active(\sigma_i) \vee timeout(\sigma_i, te_i) \\
active(\sigma) &\equiv \exists s \in dom_{Sig}(\sigma), v \in Val_\perp.\ <1, v> \in \sigma(s) \\
timeout(\sigma, te) &\equiv \mathcal{E}[\![te]\!]\sigma = 0 \\
event(\sigma, \sigma', s) &\equiv \sigma(s)(0) \neq \sigma'(s)(0)
\end{aligned}$$

The delta-time advance rule is more involved: it applies when all processes are blocked in a wait and there exists an active signal (we have not

yet reached a stable state) or a `wait for 0`. First resolution functions are applied to compute new current signal values ($\mathcal{U}(\Sigma_I)$), and then a process is activated (*i.e.* the process's leading wait statement is removed – † below) if it timed out or a signal on which it waited was active.

$$\frac{resume(<\Sigma_I, \|_I\ ss_i>)}{<\Sigma_I, \|_I\ ss_i> \rightarrow_{pgm} <\mathcal{U}(\Sigma_I), \|_I\ ss_i'>} \tag{18}$$

If $\mathcal{U}(\Sigma_I) = \{\sigma_i' | i \in I\}$ and if ss_i is equal to `(wait on` S_i `for` te_i `until` be_i`;` r_i`)`, for each $i \in I$, then we define

$$ss_i' = \begin{cases} r_i & \text{if } timeout(\sigma_i, te_i) \vee \\ & \quad \exists s \in S_i.\ event(\sigma_i, \sigma_i', s)\ \wedge\ \mathcal{E}[\![be_i]\!]\sigma_i' \\ \texttt{wait on } S_i \texttt{ for } t_i \texttt{ until } be_i;\ r_i & \text{otherwise, where } t_i = \mathcal{E}[\![te_i]\!]\sigma_i \end{cases} \tag{†}$$

In the following we will use the labels $\mathbf{A}$, $\mathbf{T}$, and δ with the $\rightarrow_{pgm}$ relation to indicate that rule 16, 17, or 18 has been used respectively. We will frequently omit the *pgm* subscript from $\rightarrow_{pgm}$. Let Act be $\{\delta, \mathbf{T}, \mathbf{A}\}$, and let α range over elements from Act.

Some Remarks

- While rule 16 is convenient theoretically, it is rather cumbersome and in practice we will use instead the following formulation:

$$\frac{\forall i \in I. <\sigma_i, ss_i> \Rightarrow_{ss} <\sigma_i', ss_i'>}{\|_I <\sigma_i, ss_i> \rightarrow_{pgm} \|_I <\sigma_i', ss_i'>} \tag{19}$$

  $\Rightarrow_{ss}$ is the transitive reflexive closure of $\rightarrow_{ss}$: $<\sigma_0, ss_0> \Rightarrow_{ss} <\sigma_n, ss_n>$ iff

$$\begin{array}{l} (\sigma_0 = \sigma_n \wedge ss_0 = ss_n) \vee \\ \exists <\sigma_1, ss_1>, \ldots, <\sigma_{n-1}, ss_{n-1}>.\ <\sigma_0, ss_0> \rightarrow_{ss} <\sigma_1, ss_1> \rightarrow_{ss} \ldots \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \rightarrow_{ss} <\sigma_{n-1}, ss_{n-1}> \rightarrow_{ss} <\sigma_n, ss_n> \end{array}$$

  One application of this rule may be replaced by zero or more applications of rule 16 and vice versa (Lemma 5.5).

- Rules 17 and 18 are not precise as they stand because the leading statement must be a wait statement. Almost always, however, the leading statement will be a sequential statement, the first statement

of which is a wait statement. Strictly speaking all ss_i must be *waiting*: $(\cdots(\texttt{wait on } S_i \texttt{ for } t_i \texttt{ until } be_i\texttt{; } r_i^1)\cdots\ r_i^{n_i})$, as defined by Definition 5.1. Alternatively add the rule

$$\frac{}{<\sigma,(ss_i;ss_2);ss_3> \rightarrow_{ss} <\sigma, ss_i;(ss_2;ss_3)>}$$

A third solution would be to syntactically identify the programs $(ss_i;ss_2);ss_3$ and $ss_1;(ss_2;ss_3)$. Similarly, `null`;ss and ss could then be identified.

- In the definition for ss_i' in both rule 17 and 18, we evaluate the time-out clause every time. Although this seems contrary to the language definition [9, Section 8.1], it functions correctly in our setting for the following reason: te_i's first evaluation corresponds to the only evaluation in the definition. Subsequent evaluations of the same wait statement are vacuous in the sense that te_i has been replaced by its denotation $\mathcal{E}[\![te_i]\!]\sigma_i$. (Equivalently, a closure of the form $<\sigma_i, te_i>$ could be used.) After the wait statement is deleted (activation of process) the next encounter of the same wait statement will contain the expression te_i afresh as a consequence of the while loop surrounding every process body (*cf.* Section 2).

  The boolean expression be_i in wait statements has the opposite characteristics of the time-out clause: it must be evaluated anew every time the wait statement is encountered. The time-out clause is evaluated at the time of suspension (*i.e.* with stores Σ_I) whereas be_i is used at time of reactivation (with $\mathcal{U}(\Sigma_I)$) [9, Section 8.1].

- The VHDL reference manual defines the simulation process as follows [9, Section 12.6]:

  1. If no driver is active, then simulation time advances to the next time at which a driver becomes active or a process resumes [due to time-out. ...]
  2. Each active signal in the model is updated. [...]
  3. For each process P, if P is currently sensitive to a signal S, and an event has occurred on S in this simulation cycle, then P resumes.
  4. Each process that has just resumed is executed until it suspends.

  We have elided parts that are irrelevant for our VHDL subset. Our semantics follows closely this description: apply rule 17 as many times as possible (1); rule 18 involves first the use of $\mathcal{U}$ corresponding to (2)

and second process resumption through wait statement deletion (3); (4) is dealt with by (one or more applications of) rule 16.

- We draw attention to the fact that an application of rule 16, 17, or 18 may be directly followed by an application of any one of these rules, except that (17) cannot precede (16).

Rule	Followed by Rule	Example
16	16	`null; null`
16	17	`null; wait for 1`
16	18	`null; wait for 0`
17	16	not possible
17	17	`wait for 2`
17	18	`wait for 1`
18	16	`wait for 0; null`
18	17	`wait for 0; wait for 1`
18	18	`wait for 0; wait for 0`

- There is no upper limit on time imposed in our semantics whereas VHDL stipulates a finite upper bound `time'high`. Concepts such as termination are more intricate to define in our case, but we believe it is more natural not to impose an arbitrary end of time.

3.5 A Generalised Treatment of Time

We can reformulate the rules dealing with time as follows.

$$\begin{aligned} \mathit{active}(\Delta, \sigma_i) &\equiv \exists s \in \mathit{dom}_{Sig}(\sigma_i), v \in \mathit{Val}. <\Delta + 1, v> \in \sigma_i(s) \\ \mathit{timeout}(\Delta, \sigma_i, te_i) &\equiv \mathcal{E}[\![te_i]\!]\sigma_i = \Delta \\ \mathit{resume}(\Delta, \Sigma_I) &\equiv \forall \delta < \Delta. \neg \mathit{resume}(\delta, \Sigma_I) \wedge \exists i \in I. \mathit{active}(\Delta, \sigma_i) \vee \mathit{timeout}(\Delta, \sigma_i, te_i) \end{aligned}$$

We change the precondition for the delta time rule (18) to be $\mathit{resume}(0, \Sigma_I)$, which is equivalent to its previous precondition $\mathit{resume}(\delta, \Sigma_I)$. We replace the single rule 17 with a family of rules corresponding to $\Delta > 0$; every rule $\mathit{resume}(\Delta, \Sigma_I)$ is more general than its successor $\mathit{resume}(\Delta + 1, \Sigma_I)$.

$$\frac{\mathit{resume}(\Delta, \Sigma_I)}{<\Sigma_I, \|_I\ ss_i(te_i)> \rightarrow_{pgm} <\mathcal{T}_\Delta(\Sigma_I), \|_I\ ss_i(\mathcal{E}[\![te_i]\!]\sigma_i - \Delta)>} \qquad (20)$$

For each $i \in I$ $ss_i(t_i)$ is equal to `(wait on` S_i `for` t_i `until` be_i`;` r_i`)`. In fact, the VHDL reference manual can be read to lead naturally to this interpretation: it is preferable to advance Δ clock ticks using one rule (with

precondition $resume(\Delta, \Sigma_I)$) rather than use the single time step (rule 17) Δ times.

It is also possible to use

$$resume(\Delta, \Sigma_I) \equiv \neg\exists i \in I, \delta < \Delta.\ active(\delta, \sigma_i) \vee timeout(\delta, \sigma_i, te_i)$$

but then $resume(\delta, \Sigma_I)$ must be defined as $\neg resume(1, \Sigma_I)$ rather than $resume(0, \Sigma_I)$, which is vacuously true.

4 A Very Small Example in Great Detail

In this section we treat a very simple example (a NOT gate with one change of input) in great detail. It is worth following the example through to understand the interaction of the delta time rule (18) and the time increment rule (17). It also shows that the initialisation of the VHDL simulation requires no special treatment in our semantics.

Consider the following two processes:

$\| \{< \sigma_0$,`while true do (o` $\Leftarrow$ `¬i after 0; wait on {i})`$>$,
$< \tau_0$,`while true do (i` $\Leftarrow$ `true after 1; wait for` ∞`)`$>\}$

The first is a simple inverter while the second provides one change of input, one time step into the future. We assume that all signals are initially false:

$$\sigma_0 = \{<\texttt{o}, \{<-\infty, \texttt{false}>, <0, \texttt{false}>\}, <\texttt{i}, \{<-\infty, \texttt{false}>, <0, \texttt{false}>>\}>\}$$
$$\tau_0 = \{<\texttt{i}, \{<-\infty, \texttt{false}>, <0, \texttt{false}>\}>\}$$

We omit all $<-\infty, v>$ pairs from the stores in the following because they persist throughout the simulation. We use rule 12 (lifted to the $\rightarrow_{pgm}$ level by rule 16) and get:

$\| \{< \sigma_0$,`while true do (o` $\Leftarrow$ `¬i after 0; wait on {i})`$>$,
$< \tau_0$,`(i` $\Leftarrow$ `true after 1; wait for` ∞`); while true do` $\cdots >\}$

Then rule 11 gives:

$\| \{< \sigma_0$,`while true do (o` $\Leftarrow$ `¬i after 0; wait on {i})`$>$,
$< \tau_1$,`(null; wait for` ∞`); while true do` $\cdots >\}$

with $\tau_1 = \{<\texttt{i}, \{<0, \texttt{false}>, <2, \texttt{true}>\}>\}$ and then rule 10 removes the leading `null`:

$\| \{< \sigma_0$,`while true do (o` $\Leftarrow$ `¬i after 0; wait on {i})`$>$,
$< \tau_1$,`wait for` ∞`; while true do` $\cdots >\}$

Now we can work only on the first process (rule 12):

$$\| \{< \sigma_0, \texttt{(o} \Leftarrow \neg \texttt{i after 0; wait on \{i\}); while true do} \cdots >, \\ < \tau_1, \texttt{wait for} \infty \texttt{; while true do} \cdots >\}$$

which, using rule 11 produces:

$$\| \{< \sigma_1, \texttt{(null; wait on \{i\}); while true do} \cdots >, \\ < \tau_1, \texttt{wait for} \infty \texttt{; while true do} \cdots >\}$$

with $\sigma_1 = \{<\texttt{o}, \{<0, \texttt{false}>, <1, \texttt{true}>\}, <\texttt{i}, \{<0, \texttt{false}>>\}>\}$. Rule 10 then results in:

$$\| \{< \sigma_1, \texttt{wait on \{i\}; while true do} \cdots >, \\ < \tau_1, \texttt{wait for} \infty \texttt{; while true do} \cdots >\}$$

At this point no more actions can be performed at the sequential program level ($\rightarrow_{ss}$ lifted to $\rightarrow_{pgm}$ by use of rule 16) because both processes are waiting. We applied the rules in the order 12, 11, 10, 12, 11, 10, which corresponded to evaluating the second process before the first. However, they could have been interleaved, for example 12, 12, 11, 11, 10, 10 with the same outcome. Now we apply the delta time rule (18) because $R_\delta(\{\sigma_1, \tau_1\})$ holds (the signal o is active in σ_1). We obtain the following stores after $\mathcal{U}$ has done its job:

$$\begin{aligned} \sigma_2 &= \{<\texttt{o}, \{<0, \texttt{true}>\}>, <\texttt{i}, \{<0, \texttt{false}>\}>\} \\ \tau_2 &= \{<\texttt{i}, \{<0, \texttt{false}>, <2, \texttt{true}>\}>\} \end{aligned}$$

No wait statements are deleted because no event occurs on i and `wait for` ∞ never wakes up. At this point all processes are suspended and the initialisation phase of the simulation has been completed. Note that no special rules where necessary. The simulation proper now begins by advancing time with rule 17 which uses $\mathcal{T}$ to produce:

$$\begin{aligned} \sigma_3 &= \{<\texttt{o}, \{<-1, \texttt{true}>, <0, \texttt{true}>\}>, <\texttt{i}, \{<-1, \texttt{false}>, <0, \texttt{false}>\}>\} \\ \tau_3 &= \{<\texttt{i}, \{<-1, \texttt{false}>, <0, \texttt{true}>, <1, \texttt{true}>\}>\} \end{aligned}$$

Strictly speaking we replace `wait for` ∞ by `wait for` $\infty - 1$ and `wait for` ∞ `on` {i} by `wait for` $\infty - 1$ `on` {i}. Of course, this does not lead to a real change. Now i is active in τ_3 so that we apply the delta rule. Its rôle is to compute the new value for i using its resolution function $f_{\texttt{i}}$ (the identity function in this case) and propagate this value to all processes

which use i, in this case both σ_3 and τ_3.

$$\begin{aligned}\sigma_4 &= \{<\mathtt{o},\{<-1,\mathtt{true}>,<0,\mathtt{true}>\}>,<\mathtt{i},\{<-1,\mathtt{false}>,<0,\mathtt{true}>\}>\}\\ \tau_4 &= \{<\mathtt{i},\{<-1,\mathtt{false}>,<0,\mathtt{true}>\}>\}\end{aligned}$$

At this point the wait statement of first process is deleted because there has been an event on i ($\sigma_3(\mathtt{i})(0) \neq \sigma_4(\mathtt{i})(0)$):

$\|\ \{<\sigma_4$,`while true do (o` $\Leftarrow$ `¬i after 0; wait on {i})`$>$,
$\quad<\tau_4$,`wait for` ∞`; while true do` $\cdots>\}$

Using rules 12, 11, 10 as before we obtain

$\|\ \{<\sigma_5$,`wait on {i}; while true do` $\cdots>$,
$\quad<\tau_5$,`wait for` ∞`; while true do` $\cdots>\}$

with stores

$$\begin{aligned}\sigma_5 &= \{<\mathtt{o},\{<-1,\mathtt{true}>,<0,\mathtt{true}>,<1,\mathtt{false}>\}>,\\ &\quad\ <\mathtt{i},\{<-1,\mathtt{false}>,<0,\mathtt{true}>\}>\}\\ \tau_5 &= \{<\mathtt{i},\{<-1,\mathtt{false}>,<0,\mathtt{true}>\}>\}\end{aligned}$$

Again, only the delta rule applies:

$$\begin{aligned}\sigma_6 &= \{<\mathtt{o},\{<-1,\mathtt{true}>,<0,\mathtt{false}>\}>,<\mathtt{i},\{<-1,\mathtt{false}>,<0,\mathtt{true}>\}>\\ \tau_6 &= \{<\mathtt{i},\{<-1,\mathtt{false}>,<0,\mathtt{true}>\}>\}\end{aligned}$$

Now the program configuration has terminated (see Section 5.2.2) because the only rule that applies advances time:

$$\begin{aligned}\sigma_7 &= \{<\mathtt{o},\{<-2,\mathtt{true}>,<-1,\mathtt{false}>,<0,\mathtt{false}>\}>,\\ &\quad\ <\mathtt{i},\{<-2,\mathtt{false}>,<-1,\mathtt{true}>,<0,\mathtt{true}>\}>\}\\ \tau_7 &= \{<\mathtt{i},\{<-2,\mathtt{false}>,<-1,\mathtt{true}>,<0,\mathtt{true}>\}>\}\end{aligned}$$

and so on. No transactions are scheduled for i so the first process will never be resumed.

An amusing process is the following:

```
while true do
      (s ⇐ s + s'delayed(s+1);
       wait for 0;
       wait for s)
```

This circuit produces the Fibonacci series with the delays between successive results equal to the value output. That is, it outputs $\mathit{fib}\, n$ at time $\Sigma_{1..n}\mathit{fib}\, n$. s must have been initialised with the value 1.

5 Properties of the Operational Semantics

In this section we first show that VHDL is essentially deterministic. Then we give a classification of program behaviours that is more refined than usual.

5.1 Parallelism and Nondeterminism

Languages that contain parallelism normally are nondeterministic, complicating both the design and verification of programs. Even though VHDL includes the parallel execution of processes its somewhat peculiar simulation model, in particular the use of delayed signal updates, ensures that the resulting nondeterminism is benign. In VHDL nondeterminism only arises through **A** actions, *i.e.* arbitrary interleaving of processes. But at every δ or **T** action all execution paths converge so that the visible behaviour is perfectly deterministic. By the visible behaviour we intend all current and past signal values, as opposed to the whole system configuration that also includes projected signal values and variables. This analysis leads naturally to the following theorem:

Theorem 5.1 (Monogenicity of $\rightarrow_{pgm}$) For all C, if $C \stackrel{\alpha}{\rightarrow} C'$ then C' is unique, with the proviso that if α is equal to **A** then C' must be a waiting configuration.

The relevance of this theorem is elucidated by the following result.

Corollary 5.2 At any point in a computation the visible system state is unique:
for all C, if $C \rightarrow^*_{pgm} C'$ and $C \rightarrow^*_{pgm} C''$ are two computations of equal length then $C' =_{Visible} C''$.

$C =_{Visible} C'$ is given below (Definition 5.2), but can be informally stated as: the current values of all common signals of C and C' are equal.

We now prove Theorem 5.1 and Lemma 5.2 through a series of definitions and lemmas.

Theorem 5.3 $\mathcal{E}$ is monogenic, that is, for all σ and e, if $\mathcal{E}[\![e]\!]\sigma = v \wedge \mathcal{E}[\![e]\!]\sigma = v'$ then $v = v'$.

Proof. A simple induction on the structure of terms. The exact assumptions, mentioned in passing in previous sections, are: (1) all monadic and binary operators on *Val* are monogenic and total; (2) $dom_{Var}(\sigma)$ and $dom_{Sig}(\sigma)$ are disjoint; (3) $\sigma(x)(-\infty)$ and $\sigma(x)(0)$ are defined for every signal x used in the expression e; (4) expressions are well-typed. □

Definition 5.1 A sequential program is *waiting* if it has the following form: $(\cdots(\texttt{wait on } S \texttt{ for } e \texttt{ while } te\texttt{; } ss_1)\texttt{; } \cdots)\texttt{; } ss_n)$ for some S, e, te, and ss_1 to ss_n. A program is *waiting* if all its constituent sequential programs are waiting.

Theorem 5.4 $\rightarrow_{ss}$ is monogenic for all well-typed sequential programs ss of the form `while true do` ss. $\rightarrow_{ss}$ is also total for non-waiting programs.

Proof. All expressions in well-typed sequential program are well-typed. The proof proceeds by structural induction on the ss. The case $ss = \texttt{null}$ relies on the presence of the enclosing `while` loop (see rule 10). `wait` statements are trivially monogenic because there is no rule for them. When they are excluded the $\rightarrow_{ss}$ becomes total, in the sense that all sequential programs can do at least two $\rightarrow_{ss}$ transitions. (To be exact, `while true do` $ss \rightarrow_{ss}$ ss`; while true do` $ss \rightarrow_{ss} ss'$`; while true do` ss or $\rightarrow_{ss}$ `while true do` ss.) □

Lemma 5.5 The following are properties of rule 16:

1. Let $C \xrightarrow{\mathbf{A}} C'$. Then exactly one sequential program configuration is modified in C'. If it is c_i with $i \in I$ (I being the index set of C) we write $C \xrightarrow{\mathbf{A}}_i C'$ and C' is unique ($C \xrightarrow{\mathbf{A}}_i C''$ implies $C' = C''$).

2. For all $i, j \in I$, if $C \xrightarrow{\mathbf{A}}_i C_1 \xrightarrow{\mathbf{A}}_j C_2$ and $C \xrightarrow{\mathbf{A}}_j C_3 \xrightarrow{\mathbf{A}}_i C_4$ then $C_2 = C_4$.

3. All applications of rule 19 may be replaced by a series of applications of rule 16, and vice versa.

4. For all C and waiting configurations C', if $C \xrightarrow{\mathbf{A}}{}^{*} C'$ then C' is unique.

Proof. (1) follows from Lemma 5.4 and the formulation of rule 16. (2) is trivial. For (3) consider that any sequence of **A** actions may be reordered using (2) so that all **A** actions pertaining to sequential program configuration c_i are consecutive. This corresponds to rule 19. (4) follows in the same way. C' must be waiting to ensure that no more **A** actions can be done because uniqueness only holds when all sequential programs have executed the same number of steps in alternative computations. (*i.e.* $C \xrightarrow{\mathbf{A}}_i C'$, $C \xrightarrow{\mathbf{A}}_j C''$ implies $C' = C''$ iff $i = j$ or $c_i = c_j$.) □

We define $=_{Visible}$ which compares the stores of two configurations:

Definition 5.2 $<\Sigma_I, \Pi_I> =_{Visible} <\Sigma'_J, \Pi'_J>$ is defined by:
$\forall s \in dom_{Sig}(\Sigma_I) \cap dom_{Sig}(\Sigma'_J). \forall i \in I, j \in J.\, s \in dom_{Sig}(\sigma_i) \cap dom_{Sig}(\sigma_j) \Rightarrow \sigma_i(s)(0) = \sigma_j(s)(0)$.

It then follows that **A** actions have no effect upon the visible state:

Lemma 5.6 For all C, $C \xrightarrow{\mathbf{A}} C'$ implies $C =_{Visible} C'$.

Proof. Lemma 5.5(1) states that only one sequential program configuration (call it c_i) is modified in C. By a simple structural induction on $c_i \equiv <\sigma_i, ss_i>$ show that an **A** action does not modify $\sigma_i(s)(0)$ for any s. (Only rules 8 and 9 modify the store.) □

Lemma 5.7 $\mathcal{U}$ is monogenic and total assuming that all signal resolution functions f_s are monogenic and total.

Proof. Simple case analysis. □

Lemma 5.8 $\mathcal{T}_\Delta$ is monogenic and total for all $\Delta > 0$.

Proof. Relies on the monogenicity and totality of set operators. □

We are now in a position to prove Theorem 5.1:

Theorem 5.1 For all C, if $C \xrightarrow{\alpha} C'$ then C' is unique, with the proviso that if α is equal to **A** C' must be a waiting configuration.

Proof. There are three cases to consider:

α=δ This follows from the monogenicity of rule 18 which in turn follows from Lemma 5.7.

$\alpha = \mathbf{T}$ Lemma 5.8 entails the monogenicity of rule 17 which leads to the result.

$\alpha = \mathbf{A}$ Use Lemma 5.5(4).

□

The proof of Corollary 5.2 is as follows:

Corollary 5.2 At any point in a computation the visible system state is unique:
for all C, if $C \rightarrow^*_{pgm} C'$ and $C \rightarrow^*_{pgm} C''$ are two computations of equal length then $C' =_{Visible} C''$.

Proof. First consider that all **A** sequences embedded in the computation

end in a waiting state that is unique by Lemma 5.5(4). (Otherwise δ or **T** cannot apply immediately afterwards.) All δ and **T** actions are monogenic (*cf.* the proof of Theorem 5.1). Finally, trailing **A** actions do not alter the visible state (Lemma 5.6) after the last δ or **T** action so that the last visible state in the computation is unique. □

5.2 Program Behaviours

Programs of sequential languages either terminate or diverge; in parallel languages there is the further possibility of deadlock. VHDL, of course, is different. As we will see below, programs never complete (Lemma 5.10) and hence the notions of termination and divergence need adjustment. Having done this, a complete classification of VHDL program behaviours is then given in Theorem 5.11.

5.2.1 Sequential Programs

Definition 5.3 A sequential program configuration c *diverges* if c can do an infinite number of $\rightarrow_{ss}$ steps.

Divergence in a sequential program is always due to a while statement.

Lemma 5.9 Every sequential program configuration c diverges or evaluates to a waiting configuration.

Proof. A sequential program ss can either do an infinite number of steps (divergence) or a finite number of steps. In the latter case, consider that all configurations can evolve to a new configuration, except when the leading statement is a wait statement (Lemma 5.4). □

5.2.2 Programs

First we prove that programs do not terminate. This will then allow us to classify all program behaviours.

Lemma 5.10 (Liveness) It is not possible for a program configuration to dead-lock, that is, to reach a state in which no transactions are possible.

Proof. Proof by contradiction: suppose that no action is possible. Then **A** is not possible, *i.e.* all sequential programs are waiting, by Lemma 5.9. Because δ cannot be applied its precondition $resume(\Sigma_I)$ must be false. Then $\neg resume(\Sigma_I)$ is true and **T** is possible, which is a contradiction. □

We can define strong and weak bisimulation as usual with $Act = \{\mathbf{A}, \delta, \mathbf{T}\}$ and regarding $\mathbf{A}$ as the silent action. (See Section 7 for the details.) Using the program

$$\Omega_n \equiv <\{\}, \|\{\texttt{while true do wait for } n\}>$$

and $\mathbf{0} = \Omega_\infty$ we now define a number of program behaviours.

Definition 5.4 A program configuration C has *terminated* if C can do an infinite number of $\mathbf{T}$ actions, without other intervening actions.

Thus, all terminated configurations are strongly bisimilar to $\mathbf{0}$.

Definition 5.5 A program configuration C has a *finite behaviour* if C can evolve to a terminated configuration in a finite number of steps.

Definition 5.6 A program configuration C has an *infinite behaviour* if C can do an infinite number of $\mathbf{T}$ actions and infinitely many δ or $\mathbf{A}$ actions.

The configuration Ω_n causes an infinite behaviour for any finite $n > 0$. There is no single configuration to which all programs with an infinite behaviour are bisimilar.

Definition 5.7 A program configuration C *diverges sequentially* if C can do an infinite number of $\mathbf{A}$ actions without intervening $\mathbf{T}$ or δ actions.

Any diverging sequential program causes the enclosing program configuration to diverge sequentially. All sequentially diverging configurations are strongly bisimilar to Δ, defined by

$$\Delta \equiv <\{\}, \|\{\texttt{while true do null}\}>$$

Definition 5.8 A program configuration C *delta-diverges* if C can do an infinite number of δ actions without intervening $\mathbf{T}$ actions.

All delta-diverging configurations are weakly bisimilar to Ω_0. One (sequentially or delta-) diverging subprogram inhibits the progress of the entire system. From the examples ($\mathbf{0}$, Δ, Ω_n) it will be clear that the for clause of the `wait` statement is very expressive.

Theorem 5.11 All program configurations C exhibit exactly one of infinite behaviour, finite behaviour, delta-divergence, or sequential program divergence.

Proof. By Lemma 5.10 we need only consider traces of $\rightarrow_{pgm}$ of infinite length. There are three labels of which at least one must therefore appear infinitely often. The proof considers four cases:

1. Only an infinite number of **T** actions are done. After all **A** and δ actions have been used only **T** actions are possible. The configuration has then terminated (finite behaviour).

2. Only an infinite number of **A** actions are possible. By a similar argument, the configuration is diverging sequentially.

3. An infinite number of δ actions are possible and only a finite number of **T** actions can be performed. The configuration therefore delta-diverges after all **T** actions have been used.

4. For the remaining traces consider that an infinite number of **T** actions must be done (otherwise we have case 2 or 3). Furthermore there must be an infinite number of **A** actions or an infinite number of δ actions (otherwise we have case 1). By definition, this is a configuration with an infinite behaviour.

□

This result reflects to some extent the stratified nature of VHDL's simulation model. In order of increasing granularity we encounter: evaluation of expressions ($\mathcal{E}$); evaluation of statements ($\rightarrow_{ss}$) or equivalently asynchronous process execution (**A** actions); computation of a fixed point within every time step (δ actions); and finally time steps modelled by **T** actions. (We refer to [7] for a similar hierarchy.) Delta delays model internal computation steps and should be invisible to the user. Thus, program divergence may take place at the sequential program level (within one process) or may be due to the failure to reach a fixed point when several processes may be involved. In both cases progress of the system as a whole is inhibited, even though the causes are very different.

5.3 Program Transformations

Operational semantics are always rather cumbersome to work with. We prefer therefore to work with derived properties. At the level of statements ($\rightarrow_{ss}$) these include program transformations. Behavioural notions to be introduced in Section 8 are more useful when discussing processes.

We wish to say little about program transformations because it is already a well-developed area. VHDL processes consists of two separate data spaces,

for variables and for signals, that are to a large extent independent. Some examples of transformations that are valid in VHDL, in addition to usual sequential program laws are:

- $s_1 \Leftarrow e_1$ `after` n_1`;` $s_2 \Leftarrow e_2$ `after` n_2 $=_{ss}$
  $s_2 \Leftarrow e_2$ `after` n_2`;` $s_1 \Leftarrow e_1$ `after` n_1 if $s_1 \neq s_2$
- $s \Leftarrow e_1$ `after` n_1`;` $s \Leftarrow e_2$ `after` n_2 $=_{ss}$ $s \Leftarrow e_2$ `after` n_2 if $n_2 \leq n_1$
- $s_1 \Leftarrow e_1$ `after` n`;` x `:=` e_2 $=_{ss}$ x `:=` e_2`;` $s_1 \Leftarrow e_1$ `after` n if $x \notin FV(e_1)$
  $FV(e_1)$ denotes the free variables of e_1.
- `if` e `then (`ss_1`;` $s \Leftarrow e_1$ `after` n_1`) else (`ss_2`;` $s \Leftarrow e_2$ `after` n_2`)`
  $=_{ss}$
  `if` e `then (`ss_1`;` x_{new} `:=` e_1`;` x'_{new} `:=` n_1`) else`
  `(`ss_2`;` x `:=` e_2`;` x'_{new} `:=` n_2`);` $s \Leftarrow x_{new}$ `after` x'_{new}

x_{new} and x'_{new} must be fresh variables. $C_1 =_{ss} C_2$ (sequential program equivalence) iff $\forall C'_1, C'_2.\, (C_1 \rightarrow_{ss} C'_1 \not\rightarrow_{ss}) \wedge (C_2 \rightarrow_{ss} C'_2 \not\rightarrow_{ss}) \Rightarrow C'_1 =_{Common} C'_2$. $=_{Common}$ encodes equality of the stores on the common domain and by $C \not\rightarrow_{ss}$ we mean that C cannot do another $\rightarrow_{ss}$ transition. Using these rules most "real computation" can be moved to immediately follow the wait statements and signal assignments can immediately precede wait statements.

6 Towards a More Abstract Semantics

The operational semantics presented in the preceding sections is useful because it allows formal reasoning about VHDL programs. It is, however, not abstract enough because it distinguishes programs that intuitively behave in the same way. Consider, for example, the two NAND gates p_1 and p_2:

$$
\begin{array}{lcl}
not & \equiv & \texttt{while true do (wait on \{i\}; o} \Leftarrow \neg\texttt{i)} \\
or & \equiv & \texttt{while true do (wait on \{i1,i2\}; o} \Leftarrow \texttt{i1} \vee \texttt{i2)} \\
and & \equiv & \texttt{while true do (wait on \{i1,i2\}; o} \Leftarrow \texttt{i1} \wedge \texttt{i2)} \\
p_1 & \equiv & and[x/o] \parallel not[x/i] \\
p_2 & \equiv & not[i1/i, x1/o] \parallel not[i2/i, x2/o] \parallel or[x1/i1, x2/i2]
\end{array}
$$

$[a/b]$ indicates that signal b has been renamed a. Programs p_1 and p_2 have the same input-output behaviour but a different structure. We made a small step towards a more abstract notion of equality with the definition of $=_{ss}$

in Section 5.3 but it is not always easy to see if one program can be transformed into another, even if a complete set of transformations were to exist. Moreover, this approach works only for sequential program fragments without wait statements within single processes because process interaction is outside the scope of $=_{ss}$. The following observation aggravates this limitation. Hardware systems are composed of many concurrently operating components that can be modelled by VHDL processes. Often these processes are relatively simple (in structural descriptions they could correspond to individual gates) and it is their interaction that requires the focus of attention. A sufficiently abstract method to compare complete processes or programs is therefore required. For any labelled transition system two candidates exist: bisimulation [12] and the testing theory of [4]. We discuss both in turn.

6.1 Bisimulation

We briefly referred to bisimulation when defining various program behaviours in Section 5.2. Define the set of actions Act by $\{\mathbf{A}, \mathbf{T}, \delta\}$ where the labels **A**, **T**, and δ represent applications of semantic rules 16, 17, and 18 respectively. Intuitively two configurations are then *strongly bisimilar* if each is capable of matching all the actions of the other [11]. This is a very close correspondence because **A** actions represent computation steps internal to sequential processes and even `null; null` and `null` are not strongly bisimilar. Since our interest lies at the level of processes we regard **A** as a silent action (like τ in CCS). This leads to the *weak bisimulation* in which **A** actions are essentially ignored. As an example, recall that a program delta-diverges if it can do an infinite number of δ actions without intervening **T** actions, *i.e.* is weakly bisimilar to `while true do wait for 0`. In fact, even delta time is really a computational artifact and does not correspond to any hardware behaviour that VHDL is intended to model. This leaves us only **T** actions to observe, corresponding to the ability to observe the passing of time, and no more. This is a rather minimal notion of observation (let us call it *very weak bisimulation*).

A more serious problem is that bisimulation ignores the functional behaviour, that is, the values of signals, so that `s` $\Leftarrow$ `true` and `s` $\Leftarrow$ `false` are bisimilar. In essence process algebras such as CCS take a highly abstract view of programs: a process is defined by the actions it can perform, and the state is contained within the process term. However, in our formalisation of VHDL information is not given by the labels of $\rightarrow_{pgm}$ but by the values on input and output signals of a program. The use of relations $=_{Visible}$ and $=_{Common}$ of previous sections are manifestations of this fact. In fact, it is

not difficult to extend bisimulation to take into account signal values (at δ and **T** actions all current values of common signals must coincide).

In common with value-passing process algebras VHDL's signals further complicate bisimulation by requiring that when a value is read *all possible* values be taken into account. This is normally handled by a quantification over the value domain of relevant action but the asynchronous nature of VHDL (discussed in more detail below) necessitates the additional possibility of "no input." A rough sketch of the extended definition of bisimimulation would be:

Definition 6.1 A binary relation $\mathcal{S}$ over program configurations is a *weak signal bisimulation* if $<C_I, C_J> \in \mathcal{S}$ implies, for all $\alpha \in Act$, $\forall C'_I.\ C_I \xrightarrow{\alpha} C'_I$ implies $\exists C'_J.\ C_J \stackrel{\alpha}{\Rightarrow}\ C'_J \wedge C'_I =_{Visible} C'_J \wedge$ (if $\alpha = \delta$ or **T** then for all common signals s, for all $v_s \in Val_\perp$, either set the projected value of s to v_s in C'_I and C'_J (giving C''_I and C''_J) or leave it unchanged) $\wedge <C''_I, C''_J> \in \mathcal{S}$. (And vice versa.)

In summary, the notion of bisimulation is well adapted to abstract process algebras but turns out to be intricate to state and cumbersome to work with in our more concrete operational semantics. This is unfortunate because the proof method of finding bisimulations to prove program equivalence is efficient and elegant.

6.2 Testing

The testing framework of [4] is a method for comparing programs; two processes are considered equal if they pass the same set of tests. An *observer* is a process or program that emulates the environment of a circuit, in other words, it supplies the *observee* with inputs and analyses the results. A test is successful if the observer indicates success, for example by raising a distinguished flag. If two circuits pass the same set of tests they are indistinguishable by all environments and may hence be considered equal. This is the basis of *observational equality*.

Testing, like bisimulation, has traditionally been applied to process algebras but, unlike bisimulation, works well for operational semantics (see also [5]).

Recall that bisimulation is maladapted for the presence of an explicit state that must be partly ignored. An observer is a normal program (collection of processes) and as such can access only its local variables, and current and past values of signals. Thus there is no need to explicitly restrict the

scope of visibility. Per definition an observer can only access the *visible* environment as defined by $=_{Visible}$.

The primary data observed during bisimulation are the labels of the semantic relation but in our testing framework signals take first place. Value passing considerably complicates bisimulation (requiring quantification over all possible input values) but it comes naturally to observers, which are, after all, just programs. Moreover, VHDL may be said to be *asynchronous*. That is, a program can continue to evolve internally (modify its state, diverge) or externally (produce results) without or despite the intervention of the observer. Also, inputs are *non-blocking*: if an input signal is active the incoming value will be consumed at the first opportunity (rule 18), but the lack of input data (more precisely, a quiet signal) does not inhibit execution of the program (*cf.* Theorem 5.10). This is fundamentally different from synchronous languages such as CCS on which bisimulation and testing are based. There only signal activity counts, whereas VHDL also includes events, null transactions, and values.[2] While asynchrony combines uneasily with bisimulation, an observer naturally emits or omits input data at specific points in time. In a language with an explicit model of time such as VHDL *when* a certain event occurs on an input can be as important as the value that arrives.

All in all, observers cope well with programs exactly because they themselves are programs. Apart from a single distinguished flag SUCCESS no new machinery needs to be introduced.

6.3 Conclusion

The two notions of bisimulation and observation are pivotal to the theory of process algebras. From the previous sections we may conclude that the observational framework can be more easily adapted to our structural operational semantics. Section 7 describes the bisimulation approach in more detail, while the testing is elaborated upon in Section 8. On a more philosophical level, the use of bisimulations is a *positive* method in the sense that it allows us to show that processes are equal, in contrast to testing which is *negative* making it easy to prove that two processes are different. To prove equality we need to find only one bisimulation but need to run an infinite number of tests (because the number of observers is infinite). Conversely,

[2] In Definition 7.7 "no input" corresponds to a quiet signal, $v_s = \sigma(s)(0)$ to an active signal without an event, and $v_s \neq \sigma(s)(0)$ to an active signal with an event. In the case of $v_s = \texttt{null}$, s is active but may or may not have an event, depending on resolution. See Section 3.4 for *active* and *event*.

to show inequality we need find only one observer.

7 Bisimulation

This section may be safely omitted by the reader familiar with standard process-algebraic techniques; it serves solely to explain in more detail the discourse of Section 6.1.

In the following sections we omit the *pgm* subscript of the $\rightarrow_{pgm}$ relation. We use the labels δ, $\mathbf{T}$, and $\mathbf{A}$ with the $\rightarrow_{pgm}$ relation to indicate that rule 18, 17 or 16 has been used respectively. Let Act be $\{\delta, \mathbf{T}, \mathbf{A}\}$, and let α range over elements from Act, β over $\{\delta, \mathbf{T}\}$, and γ over $\{\delta, \mathbf{A}\}$.

Following standard process algebra techniques [11] we introduce the concepts of strong and weak bisimulation.

Definition 7.1 A binary relation $\mathcal{S}$ over program configurations is a *strong bisimulation* if $<C, D> \in \mathcal{S}$ implies, for all $\alpha \in Act$

- $\forall C'.\, C \xrightarrow{\alpha} C'$ implies $\exists D'.\, D \xrightarrow{\alpha} D' \wedge <C', D'> \in \mathcal{S}$
- $\forall D'.\, D \xrightarrow{\alpha} D'$ implies $\exists C'.\, C \xrightarrow{\alpha} C' \wedge <C', D'> \in \mathcal{S}$

Two program configurations C and D are strongly bisimilar, written $C \sim D$ if there exists a strong bisimulation $\mathcal{S}$ such that $<C, D> \in \mathcal{S}$.

Strong bisimulation is useful for defining program behaviours such as program termination, but other than that it is too strong: even `null;null` $\not\sim$ `null`.

Definition 7.2 $C \stackrel{\mathbf{A}}{\Rightarrow} C'$ holds if and only if $C = C'$ or $\exists C_1, \ldots, C_n.\, C \xrightarrow{\mathbf{A}} C_1 \xrightarrow{\mathbf{A}} \ldots \xrightarrow{\mathbf{A}} C_n \xrightarrow{\mathbf{A}} C'$. $\exists D, D'.\, C \stackrel{\mathbf{A}}{\Rightarrow} D \xrightarrow{\mathbf{T}} D' \stackrel{\mathbf{A}}{\Rightarrow} C'$ is abbreviated by $C \stackrel{\mathbf{T}}{\Rightarrow} C'$. Similarly, let $C \stackrel{\delta}{\Rightarrow} C'$ be equal to $\exists D, D'.\, C \stackrel{\mathbf{A}}{\Rightarrow} D \xrightarrow{\delta} D' \stackrel{\mathbf{A}}{\Rightarrow} C'$.

In other words, $\stackrel{\alpha}{\Rightarrow}$ allows spurious $\mathbf{A}$ actions.

Definition 7.3 A binary relation $\mathcal{S}$ over program configurations is a *weak bisimulation* if $<C, D> \in \mathcal{S}$ implies, forall $\alpha \in Act$

- $\forall C'.\, C \xrightarrow{\alpha} C'$ implies $\exists D'.\, D \stackrel{\alpha}{\Rightarrow} D' \wedge <C', D'> \in \mathcal{S}$
- $\forall D'.\, D \xrightarrow{\alpha} D'$ implies $\exists C'.\, C \stackrel{\alpha}{\Rightarrow} C' \wedge <C', D'> \in \mathcal{S}$

Two program configurations C and D are *weakly bisimilar,* written $C \approx D$ if there exists a weak bisimulation $\mathcal{S}$ such that $<C, D> \in \mathcal{S}$.

Now, `null;null` $\approx$ `null`. Weak bisimulation is a useful relation for comparing programs because actions of a smaller granularity than the delta and time increment actions are ignored. We omit the definition of very weak bisimulation referred to in previous section because we do not use in further developments.

Consider the following simple examples.

```
q1 ≡ while true do
          (x := i1 ∧ i2;
           o ⇐ ¬x after 0;
           wait on {i1, i2})
q2 ≡ while true do
          (o ⇐ ¬(i1 ∧ i2) after 0;
           wait on {i1, i2})
q3 ≡ while true do
          (x ⇐ i1 ∧ i2 after 0;
           wait on {x}; o ⇐ ¬x after 0;
           wait on {i1, i2})
```

```
q4 ≡ ‖ {while true do
              (x ⇐ i1 ∧ i2 after 0;
               wait on {i1, i2}),
        while true do
             (o ⇐ ¬x after 0;
              wait on {x})}
```

Assuming all programs are waiting, they have the following traces when an event occurs on one of the inputs: $q_1 : \mathbf{AAAAT}$, $q_2 : \mathbf{AAT}$, $q_3 : \mathbf{AA}\delta\mathbf{AAT}$, $q_4 : \mathbf{AA}\delta\mathbf{AAT}$. We therefore have the following equivalence classes: $\{q_1\} \not\sim \{q_2\} \not\sim \{q_3, q_4\}$, $\{q_1, q_2\} \not\approx \{q_3, q_4\}$, and for very weak bisimulation all programs are equivalent $\{q_1, q_2, q_3, q_4\}$. The illusion that weak bisimulation is a reasonable basis for comparing programs is shattered when we consider that the program q_5 that computes nothing is strongly bisimilar to q_1 above (and appears in the same equivalence class in all three cases).

```
q5 ≡ while true do (null; null; wait on {i1, i2})
```

Thus weak bisimulation has a serious shortcoming: the functionality of pro-

grams is irrelevant — it is only the number of delta actions needed to compute the stable state for the current time step that matters. This inadequacy will now be dealt with.

Strong and weak bisimulation are less useful than expected because only the number of steps in an execution are used. The functional behaviour of a program is ignored. It is not the labels on the transactions that carry the information of interest but signals, which are kept in the stores. We are only interested in the current values of signals — the history we can assume to have already checked and the future is not yet fixed. Furthermore, variables are local to sequential programs and may therefore be ignored. These considerations give rise to *signal bisimulation*: in addition to being a bisimulation it requires correspondence of the current values of common signals.[3]

Definition 7.4 For $\gamma \in \{\delta, \mathbf{A}\}$, $C \stackrel{\gamma}{\Longrightarrow} C'$ holds if and only if $C \stackrel{\delta}{\Rightarrow} C'$ or $\exists C_1, \ldots, C_n . C \stackrel{\delta}{\Rightarrow} C_1 \stackrel{\delta}{\Rightarrow} \ldots \stackrel{\delta}{\Rightarrow} C_n \stackrel{\delta}{\Rightarrow} C'$. Let $C \stackrel{\mathbf{T}}{\Longrightarrow} C'$ be equal to $\exists D, D' . C \stackrel{\delta}{\Longrightarrow} D \stackrel{\mathbf{T}}{\rightarrow} D' \stackrel{\delta}{\Longrightarrow} C'$.

In other words, $\stackrel{\alpha}{\Longrightarrow}$ allows spurious $\mathbf{A}$ and δ actions.

Definition 7.5 A binary relation $\mathcal{S}$ over program configurations is a *weak signal bisimulation* if $< C_I, C_J > \in \mathcal{S}$ implies, for all $\alpha \in Act$

- $\forall C'_I . C_I \stackrel{\alpha}{\rightarrow} C'_I$ implies $\exists C'_J . C_J \stackrel{\alpha}{\Rightarrow} C'_J \wedge C'_I =_{Visible} C'_J \wedge < C'_I, C'_J > \in \mathcal{S}$.
- $\forall C'_J . C_J \stackrel{\alpha}{\rightarrow} C'_J$ implies $\exists C'_I . C_I \stackrel{\alpha}{\Rightarrow} C'_I \wedge C'_I =_{Visible} C'_J \wedge < C'_I, C'_J > \in \mathcal{S}$

Two program configurations C_I and C_J are *weakly signal bisimilar* if there exists a weak signal bisimulation $\mathcal{S}$ such that $< C_I, C_J > \in \mathcal{S}$.

Definition 7.6 A binary relation $\mathcal{S}$ over program configurations is a *very weak signal bisimulation* if $< C, D > \in \mathcal{S}$ implies, for all $\gamma \in \{\delta, \mathbf{A}\}$

- $\forall C' . C \stackrel{\mathbf{T}}{\rightarrow} C'$ implies $\exists D', D'' . D \stackrel{\delta}{\Longrightarrow} \stackrel{\mathbf{T}}{\rightarrow} D' \stackrel{\delta}{\Longrightarrow} D'' \wedge C' =_{Visible} D' \wedge < C', D'' > \in \mathcal{S}$
- $\forall C' . C \stackrel{\gamma}{\rightarrow} C'$ implies $\exists D' . D \stackrel{\gamma}{\Longrightarrow} D' \wedge < C', D' > \in \mathcal{S}$

[3] This does not allow us to ignore intermediate or temporary signals with the same name in both configurations. This is a consequence of the lack of a scope restriction operator in our VHDL subset. On the other hand, we can always rename signals so that only input and output signals are in common.

- $\forall D'.\, D \xrightarrow{\mathbf{T}} D'$ implies $\exists C', C''.\, C \stackrel{\delta}{\Longrightarrow} \xrightarrow{\mathbf{T}} C' \stackrel{\delta}{\Longrightarrow} C'' \wedge C' =_{Visible} D' \wedge < C'', D' > \in \mathcal{S}$
- $\forall D'.\, D \xrightarrow{\gamma} D'$ implies $\exists C'.\, C \stackrel{\gamma}{\Longrightarrow} C' \wedge < C', D' > \in \mathcal{S}$

Two program configurations C and D are *very weakly signal bisimilar* if there exists a very weak signal bisimulation $\mathcal{S}$ such that $< C, D > \in \mathcal{S}$.

The definitions for weak and very weak signal bisimulation deserve some careful attention. Weak signal bisimulation states that a **T** (δ) action must be matched by a **T** (δ) action possibly preceded and followed by some **A** actions, and that an **A** action may be matched by some **A** actions. Because current signal values are not changed by **A** actions, it is safe to allow spurious **A** actions, and signals are effectly only compared at δ and **T** actions. The definition of very weak bisimulation is more involved because only **T** actions must be matched exactly and δ and **A** actions are invisible. But at δ actions the current values of signals may change, and the obvious definition is therefore incorrect:

$$\forall C'.\, C \xrightarrow{\alpha} C' \text{ implies } \exists D'.\, D \stackrel{\alpha}{\Longrightarrow} D' \wedge C' =_{Visible} D' \wedge < C', D' > \in \mathcal{S}$$

(and vice versa). Consider the traces **TA**δ**AT** and **TA**δ**A**δ**AT**: the δ actions represent intermediate states at which signals do not need to correspond to **T** or δ states in the other trace. Configurations with these traces cannot be weakly signal bisimular but they may be very weakly signal bisimular. Requiring matching δ and **T** actions and identical signal values is not sufficient, however. If input signals do not change the bisimulation is likely to weak in the sense that only a very small part of the state space is explored. Thus bisimulation must be extended to also allow for input changes. All in all this makes bisimulation rather messy.

In common with value-passing process algebras VHDL's signals further complicate bisimulation by requiring that when a value may be read *all possible* values be taken into account. This is normally handled by a quantification over the value domain of relevant action but the asynchronous nature of VHDL (discussed in Section 6.2) necessitates the additional possibility of "no input." A rough sketch of the extended definition of bisimimulation would be:

Definition 7.7 A binary relation $\mathcal{S}$ over program configurations is a *weak* VHDL *bisimulation* if $< C_I, C_J > \in \mathcal{S}$ implies, for all $\alpha \in Act$, $\forall C'_I.\, C_I \xrightarrow{\alpha} C'_I$ implies $\exists C'_J.\, C_J \stackrel{\alpha}{\Rightarrow} C'_J \wedge C'_I =_{Visible} C'_J \wedge$ if $\alpha \in \{\delta, \mathbf{T}\}$ then for all common

signals s, for all $v_s \in Val_\perp$, either set the projected value of s to v_s in C'_I and C'_J (giving C''_I and C''_J) or leave it unchanged $\wedge < C''_I, C''_J > \in \mathcal{S}$. (And vice versa.)

A tentative formalisation of *weak* VHDL *bisimulation* would be:

Definition 7.8 A binary relation $\mathcal{S}$ over program configurations is a *weak* VHDL *bisimulation* if $<C_I, C_J> \in \mathcal{S}$ implies,

- $\forall C'_I.\ C_I \stackrel{\mathbf{A}}{\rightarrow} C'_I$ implies $\exists C'_J.\ C_J \stackrel{\mathbf{A}}{\Rightarrow} C'_J \wedge <C'_I, C'_J> \in \mathcal{S}$.
- $\forall \beta \in \{\delta, \mathbf{T}\}.\ \forall C'_I.\ C_I \stackrel{\beta}{\rightarrow} C'_I$ implies $\exists C'_J.\ C_J \stackrel{\beta}{\Rightarrow} C'_J \wedge\ C'_I =_{Visible} C'_J \wedge <C''_I, C''_J> \in \mathcal{S} \wedge \forall s \in Sig.\ \forall v_s \in Val_\perp.\ \forall i \in I, j \in J.\ s \in dom_{Sig}(\sigma'_i) \cap dom_{Sig}(\sigma'_j).\ (\sigma''_i(s) = update(\sigma'_i, s, v_s, 0) \wedge \sigma''_j(s) = update(\sigma'_j, s, v_s, 0)) \vee (\sigma''_i(s) = \sigma'_i(s) \wedge \sigma''_j(s) = \sigma'_j(s))$. The function *update* is defined as for the semantic rule for signal assignment (9) (*i.e.* $s \Leftarrow v_s$ `after 0`).
- And vice versa for the two previous items.

The definition of a *very weak* VHDL *bisimulation* is even more complicated — it is not clear at the moment how to exactly formulate it.

It is clear that bisimulation has been embellished to the point where it is no longer intuitive. More work is needed to asses the practical use of these definitions.

8 An Observational Semantics

To interpret our semantics within the testing theory of [4] we need to define the following entities: a set of processes $\mathcal{Q}$, a set of observers $\mathcal{O}$, a set of states *States* and a set of successful states *Success*, and a method of assigning to every observer and process a non-empty set of computations *Comp*.

$\mathcal{Q}$ and $\mathcal{O}$ are both equal to the set of all program configurations – except that observers may use the distinguished signal SUCCESS – because a program is not only defined by its program text but also by the values its variables and signals have.[4] In addition to the store Σ_I of a program a state comprises the program text $\|_I\ ss_i$ because it must be known how far each process has advanced in its execution. The program text is manipulated and is therefore part of the state so that *States* is equal to the set of all program configurations (store and program text). A computation is the set of all states an observer-observee pair passes through. Finally, a program passes

[4] Variables and signals receive their initial value directly in the store. Signal resolution functions are also part of a program configuration.

a test if the observer has assigned true to the reserved signal SUCCESS.

$$\begin{aligned}
\mathcal{Q} &\equiv \mathcal{P}(Store) \times pgm \\
\mathcal{O} &\equiv \mathcal{P}(Store) \times pgm \\
State &\equiv \mathcal{P}(Store) \times pgm \\
Comp &\equiv \mathcal{P}(State) \\
Comp(<\Sigma_O, \Pi_O>, <\Sigma_P, \Pi_P>) &\equiv \{C_i | C_0 = <\Sigma_{O\cup P}, \Pi_{O\cup P}> \wedge C_i \rightarrow_{pgm} C_{i+1}\} \\
Success &\equiv \{<\Sigma_I, \Pi_I> | \exists i \in I.\, \sigma_i(\text{SUCCESS})(0) = true\}
\end{aligned}$$

A state is *successful* if it is an element of *Success* and a computation is *successful* if contains a successful state and is *unsuccessful* otherwise.

Let us recapitulate what we have defined so far. Given a program configuration we add to the system a number of processes that *test* the program. They simulate the environment by providing all input stimuli and analysing outputs. When the observer (or test harness) decides that the program behaves as it should it signals success on the reserved signal SUCCESS. The framework is quite simple: observers are program configurations like the programs they test with the exception that they can access the reserved signal SUCCESS. No special start or stop signals are necessary (see below for a more detailed discussion on this point) and constructing a test entails simply putting the program and observer in parallel.

Having moulded the testing framework to our needs, we immediately obtain the following concepts:

Definition 8.1 A program configuration C_P *may satisfy* the observer C_O, written C_P *may* C_O, iff there exists a successful computation in $Comp(C_O, C_P)$. Similarly, C_P *must satisfy* the observer C_O, written C_p *must* C_O, iff all computations in $Comp(C_O, C_P)$ are successful.

Definition 8.2 For a given set of observers $\mathcal{O}$ we define:

- $C_1 \sqsubseteq_{may} C_2$ iff $\forall C_O \in \mathcal{O}.\ C_1$ *may* $C_O \Rightarrow C_2$ *may* C_O
- $C_1 \sqsubseteq_{must} C_2$ iff $\forall C_O \in \mathcal{O}.\ C_1$ *must* $C_O \Rightarrow C_2$ *must* C_O
- $C_1 \sqsubseteq_{test} C_2$ iff $C_1 \sqsubseteq_{may} C_2 \wedge C_1 \sqsubseteq_{must} C_2$.

The may and must preorders indicate a fitness for purpose: $\sqsubseteq_{may}$ can be read as the *capacity to pass a test* so that $C \sqsubseteq_{may} C'$ means that C' can pass at least all the tests C can pass. The preorder $\sqsubseteq_{must}$ indicates the

incapacity to fail a test and $C \sqsubseteq_{must} C'$ states that all tests that C always passes are also always successful for C'. This then induces a notion of implementation: C_P *implements a specification* C_S ($C_P \sqsubseteq_{impl} C_S$) iff $C_P \sqsubseteq_{may} C_S \wedge C_S \sqsubseteq_{must} C_P$. An implementation must satisfy all tests that the specification always satisfies; moreover, the implementation may not pass tests that the specification does not pass. The former clause defines the minimum behaviour an implementation must exhibit, the latter indicates the limit of possible behaviours of an implementation.

8.1 Results of the Observational Semantics

Due to the limited nondeterminism of VHDL (Theorem 5.1) the may and must preorders coincide, as is shown by the next theorem.

Theorem 8.1 $\sqsubseteq_{may}=\sqsubseteq_{must}=\sqsubseteq_{test}$

The result can easily be shown:
Proof. Without loss of generalisation, pick a sequential program c_j ($j \in O$) of observer C_O. By Corollary 5.2 all signals that c_j uses are uniquely defined at each point in the computation. Moreover, the value of variables of c_j can be modified only by c_j so that no non-determinism arises due to these either. Interleaving of **A** actions of other processes therefore has no effect upon the behaviour of c_j. Thus, if c_j signals success in one computation, it will do so in all other computations too. May and must preorders therefore coincide. (Basically the only indeterminism is due to interleaving of the **A** actions of different processes.) □
This result allows us to omit the *may*, *must*, and *test* subscripts without ambiguity. We write $\simeq$ for the equivalence relation induced by $\sqsubseteq$ ($\simeq = \sqsubseteq \cap (\sqsubseteq)^{-1}$). It is equal to previously defined implementation preorder $\sqsubseteq_{impl}$.

Suppose two programs have the same input-output behaviour but possibly a different structure (p_1 and p_2 of the Section 6, for example). Within a larger system could some or all occurrences of p_1 be replaced by p_2 without changing the behaviour of the system as a whole? The answer is yes, if we use observational equivalence:

Theorem 8.2 $\simeq$ is a congruence with respect to parallel composition. That is,
$C_{J_1} \simeq C_{J_2} \Leftrightarrow \forall C_I . C_I \parallel C_{J_1} \simeq C_I \parallel C_{J_2}$. (Taking care that variables occurring in only one of C_{J_1} and C_{J_2} are not captured by C_I.)

The proof relies on there being no restriction on observers C_O so that any context C_I can be interpreted as an observer.

Proof. The implication to the left follows by setting I to the empty set. The implication to the right is proved as follows. $C_{J_1} \simeq C_{J_2}$ is defined as $\forall C_O.C_{J_1}$ *must* $C_O \Leftrightarrow C_{J_2}$ *must* C_O. That is, all computations of $C_{J_1 \cup O}$ are successful iff all computations of $C_{J_2 \cup O}$ are successful. If we now set $C_O = C_{I \cup O'}$ we obtain: all computations of $C_{J_1 \cup I \cup O'}$ are successful iff all computations of $C_{J_2 \cup I \cup O'}$ are successful. This is equal to $C_{I \cup J_1}$ *must* $C'_O \Leftrightarrow$ $C_{I \cup J_2}$ *must* C'_O. In other words, $\forall C'_O.\, C_{I \cup J_1}$ *must* $C'_O \Leftrightarrow C_{I \cup J_2}$ *must* C'_O, which is the definition of $C_{I \cup J_1} \simeq C_{I \cup J_2}$. □

This theorem is important because it allows us to substitute observationally equivalent system components without affecting the overall system behaviour. A refinement-based design methodology can therefore be safely adopted to construct circuits.

8.2 The Power of Observation

Our notion of behavioural equivalence is very strong because observers can inspect and modify signals at every delta step. Behaviourally equivalent processes must exhibit the same behaviour at every delta step. Consider the following processes:

$p_3 \equiv$ `while true do (wait on {i}; o` $\Leftarrow \neg$`i)`
$p_4 \equiv$ `while true do (wait on {i}; wait for 0; o` $\Leftarrow \neg$`i)`
$p_5 \equiv$ `while true do (wait on {i}; o` $\Leftarrow \neg$`i; wait for 0)`
$o_1 \equiv$ `i` $\Leftarrow \neg$`i; wait for 0; i` $\Leftarrow \neg$`i; wait for 0;` SUCCESS $\Leftarrow$ `(o = i)`
$o_2 \equiv$ `i` $\Leftarrow \neg$`i; wait for 0; i` $\Leftarrow \neg$`i; wait for 0;`
`wait for 0;` SUCCESS $\Leftarrow$ `(o` $\neq$ `i)`

o_1 distinguishes p_3 and p_4 so that $p_3 \not\simeq p_4$. Perhaps unexpectedly, $p_3 \not\simeq p_5$ although `wait for 0` does not delay any statements that modify the state. It does, however, affect the rate with which it is able to consume its inputs.

These observations might lure us to the mistaken belief that delta actions have a temporal significance. This is not so; delta actions represent internal computation steps of the simulation in the convergence to a fixed point or steady state. Inputs to a circuit remain constant during one time step (from one **T** action to the next) and outputs should only be read when they have stabilised (*i.e.* at **T** actions). Observers may be thought of as emulating the environment and this suggests limiting the expressiveness of observers in this way. But there is no simple solution: if we circumscribe the power of observers we must make a corresponding restriction to programs, assuming we wish observational equivalence to be a congruence (see Theorem 8.2). Any effort to excise delta delays reduces the VHDL subset to

a trivial language. More work is needed to find a coarser and more useful notion of observation.

Although we have not yet proved this formally, it is clear that bisimulation outlined in Definition 7.7 is more discerning than our testing framework because observers cannot always reconstruct when (delta) time advances. (Consider a δ action caused only by a `wait for 0` statement; the state does not change and the application of the delta rule passes unnoticeably. Conversely, all the information that is available to an observer can also be used by bisimulation, so that bisimulation is strictly more powerful.)

8.3 Observing Processes Or Sequential Programs

Parallelism in VHDL is uncomplicated because only complete sequential programs can be executed in parallel. Thus our notion of observation lies at the process level (whole sequential programs, *i.e.* $\rightarrow_{pgm}$). This contrasts with the approach of De Nicola and Pugliese who give an observational semantics for the asynchronous concurrent language LINDA in [5]. In LINDA concurrency can be introduced at the level of individual statements through explicit process creation (eval) so that a more refined notion of observation is necessary and programs are tested at the level of sequential program statements. As a result the composition of observer and observee is more involved: in addition to the distinguished signal SUCCESS special start and stop signals are needed. It is not clear if a similarly detailed notion of observation can easily be introduced for our semantics. We cannot simply regard our observers as (partial) sequential programs because wait statements cause an interaction of statement and program semantics. As defined at present $\simeq$ is not a congruence at the $\rightarrow_{ss}$ level because an observer cannot modify the past behaviour of programs. In particular, the problem is caused by histories of signals: $ss_1 \simeq ss_2 \not\Rightarrow$ `wait for` $n;ss_1 \simeq$ `wait for` $n;ss_2$.

9 Conclusions

Of research into VHDL the operational semantics by van Tassel [6, Chapter 3] is closest to ours; but it omits arbitrary wait statements simplifying the semantic model considerably. Formalisation in terms of petri nets by Olcoz [6, Chapter 5] and Börger *et al.* [6, Chapter 4]) are not compositional so that properties can be proven of whole programs only — this is a severe practical limitation. The functional semantics of Fuchs and Mendler [6, Chapter 1] and Breuer *et al.* [6, Chapter 2], as well as the denotational semantics by Breuer *et al.* [3] and stream-based semantics [16] do not suffer from this

defect but are less intuitive than an operational approach. However, because they are often more abstract than our semantics reasoning with them may well be easier.

We have presented a semantics for VHDL subset that contains the principal features of one-entity VHDL programs, to be precise: delta delays, arbitrary wait statement, zero delay scheduling, parallel processes, and local variables. Resolution functions are also included, but they must be commutative. Of the various methods that have been used to define VHDL formally we believe ours to be one of the simplest and most intuitive. That the semantics correctly reflects the informal understanding of VHDL is supported by the fact that the properties that we proved are "common knowledge." Monogenicity of the semantics is important in theory and practice. Using the testing theory to give an observational semantics for a language such as VHDL has been fruitful. Our notion of equivalence on programs that is a congruence is an essential ingredient of any compositional method, be it a formal theory of correctness or an informal design tool.

Future work includes a cleaner characterisation of bisimulation and its relation to observational equivalence. The operational semantics could be extended by including (function) declarations to bring resolution functions into the language, and allowing multiple entities. Some small examples are presented in the technical report version of this paper; they demonstrate that practical use of our semantics is difficult.

We thank Rosario Pugliese for many useful discussions about the application of process algebraic methods to our VHDL semantics, Flavio Corradini for proof reading the published version of this report, and the referees of that paper for valuable suggestions.

References

[1] Dominique Borrione, Laurence Pierre, and Ashraf Salem. PREVAIL: A proof environment for VHDL descriptions. In P Prinetto and P Camurati, editors, *Advanced Research Workshop on Correct Hardware Design Methodologies*, pages 163–186. ESPRIT CHARME, North Holland, June 1991.

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

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

[4] R De Nicola and M C B Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1994.

[5] Rocco De Nicola and Rosario Pugliese. Testing Linda: Observational semantics for an asynchronous language. Rapporto di Ricerca SI/RR-94/06, Dipartimento di Scienze dell'Informazione, Università di Roma, Italy, November 1994. To appear in Structures in Concurrency Theory (STRICT'95).

[6] Carlos Delgado Kloos and Peter T Breuer, editors. *Formal Semantics for VHDL*, volume 307 of *Kluwer International Series In Engineering And Computer Science*. Kluwer Academic Publishers, March 1995.

[7] Ivan V Fillippenko. VHDL verification in the state delta verification system (SDVS). In *1991 International Workshop on Formal Verification in VLSI Design*, January 1991.

[8] K G W Goossens. Reasoning about VHDL using operational and observational semantics. In *Advanced Research Workshop on Correct Hardware Design Methodologies*. ESPRIT CHARME, Springer Verlag, October 1995. A Longer version is available as Università di Roma "La Sapienza" Rapporto di Ricerca SI/RR 95/06.

[9] The Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY10017 USA. *IEEE Standard VHDL Language Reference Manual*, IEEE std 1076-1987 edition, 1988.

[10] The Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY10017 USA. *IEEE Standard VHDL Language Reference Manual*, IEEE std 1076-1993 edition, 1993.

[11] Robin Milner. *Communication and Concurrency*. Prentice Hall International Series in Computer Science. Prentice Hall, 1989.

[12] D M R Park. *Concurrency and Automata on Infinite Sequences*, volume 104 of *Lecture Notes in Computer Science*. Springer Verlag, 1980.

[13] Gordon Plotkin. A structural approach to operational semantics. Technical Report FN-19, Computer Science Department, Aarhus University (DAIMI), 1981.

[14] Simon Read. *Formal Methods for VLSI Design*. PhD thesis, Department of Computation, University of Manchester, 1994.

[15] Ashraf Salem and Dominique Borrione. Formal semantics of VHDL timing constructs. In *Euro-VHDL Stockholm*, September 1991.

[16] L Sanchez, C Delgado Kloos, and P T Breuer. Stream semantics for VHDL: An example. In *Workshop on Design Methodologies for Microelectronics and Signal Processing*, 1993.

[17] Gabriele Umbreit. Providing a VHDL-interface for proof systems. In *EURO-DAC*, pages 698–703, 10662 Los Vaqueros Circle, PO Box 3014, Los Alamitos, CA 90720-1264, September 1992. IEEE Computer Society Press.

[18] Philip A Wilsey. Developing a formal semantic definition of VHDL. In J Mermet, editor, *VHDL for Simulation, Synthesis and Formal Proofs of Hardware*, pages 243–256. Kluwer Academic Publishers, 1992.