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

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.

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, 14–18]). 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) signals, signal assignments (including zero delay signal scheduling), all versions of wait statements, if and

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

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 2). Also in Section 4 we show a (limited) monogenicity result for the operational semantics: despite the presence of parallelism executions are essentially deterministic (Theorem 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 6 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 4). 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, signals (possibly resolved), 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 below 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:

$$\begin{array}{l}\mathcal{E} : e \times \mathit{Store} \rightarrow \mathit{Val}_\perp \\ \rightarrow_{ss}\colon (\mathit{Store} \times ss) \times (\mathit{Store} \times ss) \\ \rightarrow_{pgm}\colon (\mathcal{P}(\mathit{Store}) \times \mathit{pgm}) \times (\mathcal{P}(\mathit{Store}) \times \mathit{pgm})\end{array}$$

$\mathcal{E}$ is a semantics for evaluating expressions in a store, $\rightarrow_{ss}$ defines how statements evolve together with a store, and $\rightarrow_{pgm}$ 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 $\rightarrow_{pgm}$ 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:

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

Signals are mapped onto sets f, which we will frequently interpret as partial functions $f : \mathbf{Z} \rightarrow \mathit{Val}_\perp$ 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 $<n, \bot>$ 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: $v \in \mathit{Val}$; $s \in \mathit{Sig}$; $\sigma \in \mathit{Store}$; $\Sigma \in \mathcal{P}(\mathit{Store})$; $\Pi, \|_I\ ss_i \in \mathcal{P}(ss), \mathit{pgm}$; $c, <\sigma, ss> \in \mathit{Store} \times ss$; $C, <\Sigma_I, \Pi_I>, \|_I <\sigma_i, ss_i> \in \mathcal{P}(\mathit{Store} \times ss)$. As discussed previously, there is an obvious correspondence between $\mathcal{P}(ss)$ and *pgm*. Whether we use $<\Sigma_I, \Pi_I>$ or $\|_I\ <\sigma_i, ss_i>$ 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 $\mathit{dom}_{\mathit{Sig}} = \mathit{dom} \circ \pi_2$. We write `null` for the bottom element of $\mathit{Val}_\bot$ and leave implicit projections and injections converting between *Val* and $\mathit{Val}_\bot$.

Semantic Functions

Two functions, $\mathcal{T}$ and $\mathcal{U}$, are defined to handle the advance of time (rule 7) and delta time (rule 8). They are best read in conjunction with these rules.

The advance of time $\mathcal{T}$ on a store is defined as follows: 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." Variables are unchanged: $\mathcal{T}(\sigma)(x) = \sigma(x)$.

A signal s is *active* if $\exists \sigma \in \Sigma_I, v \in \mathit{Val}_\bot . <1, v> \in \sigma(s)$. 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):

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

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 1) we feel justified restricting resolution functions to be commutative.

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

3.2 Expressions

The expression fragment of the language may be given any suitable semantics. We show two of the seven rules in a denotational style:[2]

$$\mathcal{E}[\![x]\!]\sigma = \sigma(x)(0) \;\text{ if } x \in \mathit{dom}_{\mathit{Sig}}(\sigma) \tag{1}$$

$$\mathcal{E}[\![s\texttt{'delayed(}e\texttt{)}]\!]\sigma = \sigma(s)(n) \;\text{ largest } n \in \mathit{dom}(\sigma(s)).\, n \leq -\mathcal{E}[\![e]\!]\sigma \tag{2}$$

3.3 Statements

$$\frac{<\sigma, ss_1> \rightarrow_{ss} <\sigma', ss'>}{<\sigma, ss_1;ss_2> \rightarrow_{ss} <\sigma', ss';ss_2>} \tag{3}$$

Read this as "assuming that the store-program pair $<\sigma, ss_1>$ evaluates to store σ' with program ss_1', the pair $<\sigma, ss_1;ss_2>$ evaluates to the program $ss';ss_2$ in store σ'."

$$\frac{\mathcal{E}[\![e]\!]\sigma = v \qquad\qquad \mathcal{E}[\![et]\!]\sigma = t}{<\sigma, x \Leftarrow e \;\texttt{after}\; et> \rightarrow_{ss} <\sigma[f/x], \texttt{null}>} \tag{4}$$

where $f = (\sigma(x) \setminus \{<n, \sigma(n)>|n > t\}) \;\cup\; \{<t+1, v>\}$.
There is no rule for null alone: it is handled in conjunction with the sequencing operator:

$$<\sigma, \texttt{null};\, ss> \rightarrow_{ss} <\sigma, ss> \tag{5}$$

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 7 and 8 of Section 3.4). The remaining five rules, for the assignment, while, and if statements, are standard.

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 6). 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 7). If, on the other hand, some signal remains active relevant processes are reactivated without any change to time (rule 8) – this zero-time increment is also known as delta time. Process resumption involves removing the leading wait statement due to

[2] 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 $\mathit{Val}_\perp$ map to store the penultimate value of s.

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.

$$\frac{<\sigma_j, ss_j> \rightarrow_{ss} <\sigma'_j, ss'_j>}{\|_{I\cup\{j\}} <\sigma_i, ss_i> \rightarrow_{pgm} \|_{I\cup\{j\}} <\sigma'_i, ss'_i>} \tag{6}$$

$\sigma'_i = \sigma_i$ 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 ($\mathcal{E}[\![te_i]\!]\sigma_i - 1$).

$$\frac{\neg \mathit{resume}(<\Sigma_I, \|_I\ ss_i(te_i)>)}{<\Sigma_I, \|_I\ ss_i(te_i)> \rightarrow_{pgm} <\mathcal{T}(\Sigma_I), \|_I\ ss_i(\mathcal{E}[\![te_i]\!]\sigma_i - 1)>} \tag{7}$$

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 8.)

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

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.

$$\frac{\mathit{resume}(<\Sigma_I, \|_I\ ss_i>)}{<\Sigma_I, \|_I\ ss_i> \rightarrow_{pgm} <\mathcal{U}(\Sigma_I), \|_I\ ss'_i>} \tag{8}$$

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

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

Strictly speaking, waiting programs in rules 7 and 8 must be defined as follows:

Definition 1. A sequential program is *waiting* if it is of the form:
`(`$\cdots$`(wait on` S `for` e `while` te`;` ss_1`);` $\cdots$`);` 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.

In the definition for ss'_i in both rule 7 and 8, 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 $\mathcal{E}[\![te_i]\!]\sigma_i$. After the wait statement is deleted (activation of process – † above) 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. Also, 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].

In the following we will use the labels **A**, **T**, and δ with the $\rightarrow_{pgm}$ relation to indicate that rule 6, 7, or 8 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*.

4 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.

4.1 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 1 ((Monogenicity of $\rightarrow_{pgm}$)). For all C, if $C \overset{\alpha}{\rightarrow} C'$ then C' is unique, with the proviso that if α is equal to **A** then C' must be a waiting configuration.

The proof is straightforward and relies on the monogenicity of the semantics for expressions and sequential statements.[3] The relevance of this theorem is elucidated by the following result.

[3] Proofs of all theorems have been omitted from this paper; they may be found in [8].

Corollary 1. 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 $C' =_{Visible} C''$.

$C =_{Visible} C'$ may be informally stated as: all past and current values of all common signals of C and C' are equal.

4.2 Program Behaviours

Programs of sequential languages either terminate or diverge. Parallel languages have the further possibility of deadlock. VHDL, of course, is different. Let us first consider individual sequential programs. A sequential program configuration c *diverges* if c can do an infinite number of $\rightarrow_{ss}$ steps. Then every sequential program configuration either diverges or evaluates to a waiting configuration (Definition 1). Divergence in a sequential program is always due to a while statement. Because δ and $\mathbf{T}$ actions exhaust all other possibilities, at the level of programs it then follows that:

Lemma 1 ((Liveness)). It is not possible for a program configuration to deadlock, that is, to reach a state in which no transactions are possible.

Using the program configuration $\Omega_n \equiv$ `while true do wait for` n we can now define various behaviours of program configurations C:

- 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 $\Delta \equiv$ `while true do null`. (We can define strong and weak bisimulation as usual with $Act = \{\mathbf{A}, \delta, \mathbf{T}\}$ and regarding $\mathbf{A}$ as the silent action.)
- 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.
- 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.
- 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 $\mathbf{0} = \Omega_\infty$. A program configuration C then has a *finite behaviour* if C can evolve to a terminated configuration in a finite number of steps.

That this list is exhaustive may easily be proved:

Theorem 2. Every program configuration C exhibits exactly one of finite behaviour, infinite behaviour, delta-divergence, or sequential program divergence.

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 ($\mathbf{A}$ actions); computation of a fixed point within every time step (δ actions); and finally time steps modelled by $\mathbf{T}$ actions. (We refer to [7] for a similar hierarchy.) Delta delays model internal computation steps and should be invisible to the user. 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.

4.3 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 6 are more useful when discussing processes.

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:

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

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.

5 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:

$\mathit{not} \equiv$ `while true do (wait on {i}; o ⇐ ¬i)`

$\mathit{or} \equiv$ `while true do (wait on {i1,i2}; o ⇐ i1 ∨ i2)`

$\mathit{and} \equiv$ `while true do (wait on {i1,i2}; o ⇐ i1 ∧ i2)`

$p_1 \equiv \mathit{and}[x/o] \parallel \mathit{not}[x/i]$

$p_2 \equiv \mathit{not}[i1/i, x1/o] \parallel \mathit{not}[i2/i, x2/o] \parallel \mathit{or}[x1/i1, x2/i2]$

$[a/b]$ indicates that signal b has been renamed a. Programs p_1 and p_2 have the same input-output behaviour but a different structure. A more abstract notion of equality at the level of statements ($=_{ss}$) can be defined 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.

5.1 Bisimulation

We briefly referred to bisimulation when defining various program behaviours in Section 4.2. Define the set of actions *Act* by $\{\mathbf{A}, \mathbf{T}, \delta\}$ where the labels $\mathbf{A}$, $\mathbf{T}$, and δ represent applications of semantic rules 6, 7, and 8 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 $\mathbf{A}$ 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 $\mathbf{A}$ as a silent action (like τ in CCS). This leads to the *weak bisimulation* in which $\mathbf{A}$ 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 $\mathbf{T}$ 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 $\mathbf{T}$ actions to observe, corresponding to the ability to observe the passing of time, and no more. This is a rather minimal notion of observation.

A more serious problem is that bisimulation ignores the functional behaviour, that is, the values of signals, so that `s ⇐ true` and `s ⇐ 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 $\mathbf{T}$ 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 2. A binary relation $\mathcal{S}$ over program configurations is a *weak signal bisimulation* if $<C_I, C_J> \in \mathcal{S}$ implies, for all $\alpha \in Act$, $\forall C'_I.\ C_I \overset{\alpha}{\rightarrow} C'_I$ implies $\exists C'_J.\ C_J \overset{\alpha}{\Rightarrow}\ C'_J \wedge C'_I =_{Visible} C'_J \wedge$ (if $\alpha = \delta$ or $\mathbf{T}$ 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.)

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.

5.2 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 8), but the lack of input data (more precisely, a quiet signal) does not inhibit execution of the program (*cf.* Theorem 1). 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.[4] While asynchrony combines uneasily with bisimulation, an observer naturally emits or omits input data at specific points in time.

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.

5.3 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. Both methods are explored in [8], but in the following we restrict ourselves to the testing theory. 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, to show inequality we need find only one observer. Thus, even though bisimulation is not further elaborated upon it remains an alternative worth investigating.

6 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 C_O and process C_P a non-empty set of computations $Comp(C_O, C_P)$.

$\mathcal{Q}$ 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.[5] 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

[4] In Definition 2 "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*.

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

through. Finally, a program passes a test if the observer has assigned true to the reserved signal SUCCESS.

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

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

Definition 4. For a given set of observers $\mathcal{O}$ we define:

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

6.1 Results of the Observational Semantics

Due to the limited nondeterminism of VHDL (Theorem 1) the may and must preorders coincide, as is shown by the next theorem.

Theorem 3. $\sqsubseteq_{may} = \sqsubseteq_{must} = \sqsubseteq_{test}$

The result can easily be shown: observers can only inspect the visible system state which by Corollary 1 has a unique computation path. 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 previous section, 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 4. $\simeq$ 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. 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.

6.2 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 ⇐ ¬i)`

$p_4 \equiv$ `while true do (wait on {i}; wait for 0; o ⇐ ¬i)`

$p_5 \equiv$ `while true do (wait on {i}; o ⇐ ¬i; wait for 0)`

$o_1 \equiv$ `i ⇐ ¬i; wait for 0; i ⇐ ¬i; wait for 0;` SUCCESS `⇐ (o = i)`

$o_2 \equiv$ `i ⇐ ¬i; wait for 0; i ⇐ ¬i; wait for 0;`
`wait for 0;` SUCCESS `⇐ (o ≠ 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` seems not to 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 4). 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 2 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.)

6.3 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: $ss_1 \simeq ss_2 \not\Rightarrow \texttt{wait for } n;ss_1 \simeq \texttt{wait for } n;ss_2$.

7 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 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 this paper, and the referees 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. Rapporto di Ricerca SI/RR 95/06, Dipartimento di Scienze dell'Informazione, Università di Roma “La Sapienza”, April 1995.
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.