1

A FUNCTIONAL SEMANTICS FOR DELTA-DELAY VHDL BASED ON FOCUS

Max Fuchs*, Michael Mendler**

**Institut für Informatik*
Technische Universität München

***Department of Computer Science*
Technical University of Denmark

'It's long,' said the Knight,
'but very, VERY beautiful.
Everybody that hears me sing it –
either it brings the TEARS into their eyes, or else – '
— Lewis Carroll, Alice Through the Looking Glass

ABSTRACT

This tutorial paper gives a functional semantics for delta-delay VHDL, *i.e.* VHDL *restricted to zero-delay signal assignments. In combination with the sequential statements zero-delay signal assignment is sufficient to generate the full algorithmic expressibility of* VHDL. *The restriction is useful for a formal semantics of* VHDL *aimed at higher levels of abstraction where real, absolute, and precise timing often is painful if not impossible to prescribe.*

The approach employs the functional specification methodology Focus *which is based on the concept of streams and stream-processing functions. It advocates a three-level semantics reflecting* VHDL*'s three syntactic levels of expressions, statements, and processes.*

1 INTRODUCTION

There is a widespread consensus which interprets the reference semantics of VHDL [66] to define the behaviour of processes as proceeding in two dimensions of time, corresponding to the two levels of delta and physical delay execution. Although this informal two-dimensionality of time is well recognized, when it comes to formal functional semantics the delta delay generally is ignored, with few exceptions such as [98, 100]. Many approaches focus on the physical time dimension while excluding delta delay execution [21, 106, 107, 108, 121]. The semantics of [119] does model zero delay signal assignment but leaves out sequential statements. Our objective is to complement these approaches by taking the orthogonal route, namely to make the delta delay the object of our formal investigation with physical time being of secondary importance. In fact, delta delay and sequential statements already provide all the computational power and thus make up VHDL as a programming language. On top of this semantical platform various notions of 'time' can be encoded depending on the particular level of abstraction chosen. The usual low-level notion of physical time, for instance, may be obtained simply as a side effect on a global 'time' signal, which is maintained and incremented every now and then by dedicated processes [100]. On higher levels of abstraction, such as system or architecture level, one might want to abstract away from physical time completely, or use more abstract models of time [54]. Focusing on delta delay reflects the fact that the particular model of time can be left to the application level and need not be built into a formal semantics for VHDL.

We give a functional semantics for delta-delay VHDL (δVHDL) which is essentially VHDL restricted to zero-delay signal assignment. The formal setting in which our semantics is developed is Focus [28], a general functional framework for the formal specification and development of distributed systems. It allows us to give a precise, consistent, and unambiguous definition of the intended dynamic behavior of VHDL constructs directly in terms of ordinary mathematical objects such as sets, relations, functions, lists, streams, and so on. This contrasts with other functional approaches that proceed indirectly via a translation or encoding into another formal language. Read and Edwards [98] use a translation of VHDL into the lisp-like language of the Boyer-Moore theorem prover, Breuer *et al.* [21] translate into the lazy functional language Gofer; Sánchez *et al.* [107, 108] propose a two-step translation of VHDL, first into a special dialect VHDL* and from there into the stream language STREAM; Also, van Tassel [119] presents what is essentially a translation semantics, namely into the language of HOL (Higher-Order-Logic); Reetz and Kropf [100] translate into flow graphs which in turn are encoded in HOL; Wilsey [121] puts forward a par-

ticular formalism of Interval Temporal Logic. In all these cases the problem of providing a semantics for VHDL is 'solved' by reducing it to the semantics of another language, which does not necessarily make it easier to get a clear understanding. The advantage of our direct mathematical approach is that we are not stuck with a particular and restricted formalism but are free to choose the concepts most suited to explain a given semantical feature of VHDL. We will exploit this by using different semantical styles at the levels of expressions, statements, and processes.

Our semantics aims at VHDL as a specification and programming rather than description language, to be used mainly at higher levels of abstraction. An abstract system description gets assigned a Focus semantics in terms of streams and stream-processing functions, which is amenable to formal analysis with a variety of reasoning styles and proof techniques supported by Focus. Focus also offers powerful refinement calculi for distributed systems to be developed in a step-wise manner [28].

2 A MOTIVATING EXAMPLE

It will be useful to have our formal development be accompanied by a working example just complex enough to convey the basic ideas. Here the counter proposed in the Introduction to this volume is chosen. It is well understood in both its synchronous and asynchronous variants and thus is ideally suited to illustrate our formal semantics of δVHDL.

One of the first steps to be taken in modelling a piece of hardware is to decide for a suitable level of abstraction for both the data manipulated and the notion of time. The counter, for instance, may be seen to count either over integers or over finite bitvectors, and in the case of a synchronous counter we may decide to model its dynamic behaviour *wrt.* synchronous time given by the succession of clock ticks, or *wrt.* to real physical time measured in nano seconds. Abstraction is a very successful engineering principle not least because it helps us to master the design of complex systems. The functional behaviour of a microprocessor, for instance, could not possibly be described solely at the level of bit-vectors and real physical time. On the other hand, abstraction by its very nature always involves some loss of low-level information which may not be always safe to ignore. For instance it may be necessary, as a subcomponent of a larger system, to model a counter over finite bit-vectors and physical time, in order to keep track of arithmetical overflow and timing violations.

In this paper we will focus on the timing aspect and show that the notion of a δ-delay is powerful enough to encompass both synchronous and asynchronous timing descriptions. The idea is that we are free to take a δ-step to model the tick of a global system clock, or as the passage of a fictive physical unit of time. Note that the potential for data abstraction will be clear from the fact that δVHDL is parameterized in a collection of primitive data types which we are free to choose in an arbitrary way.

The left-hand part of Figure 1 contains the description, in δVHDL, of an asynchronous 1-bit counter stage with input signal i and output signal o. The process cnt_1 has two internal variables. Variable z stores the output state of the counter stage and x contains the last input read. Whenever the input switches from 1 to 0, the condition **if** i = '0' **and** x = '1' becomes true and the output state z is inverted. The new output is written to the output signal with o <= z. Since this update of the output does not become effective in the same but in the next δ step, we get a 1δ delay from the triggering input edge to the resulting change of the output. This models the propagation delay of an asynchronous counter stage, where 1δ may corresponds to some arbitrary but fixed physical delay unit, say $3.1412ns$. Notice, although the process of Figure 1 has an empty sensitivity list there is no explicit wait statement in the body. In δVHDL there is always an implicit '**wait for** 0 ns' at the end of a process which, since it is built in automatically, need not be written into the code.

```
cnt1 (i, o) :
process
  variable x, z : BIT := '0';
begin
  if i = '0' and x = '1'
    then z := not z else null;
  x := i; o <= z;
end process;
```

```
cnt3 (i, o0, o1, o2) :
design
begin
  cnt1 (i, o0)
  cnt1 (o0, o1)
  cnt1 (o1, o2)
end design;
```

Figure 1 A Simple 3-Bit Asynchronous Counter

Three such 1-bit counters can be composed to form a 3-bit asynchronous counter cnt_3 with input i and outputs o_0, o_1, o_2, as shown on the right-hand side of Figure 1. One expects that the resulting counter has a total propaga-

tion delay of 3δ. In fact, a VHDL simulation would produce a typical timing diagram like the one shown in Figure 2.

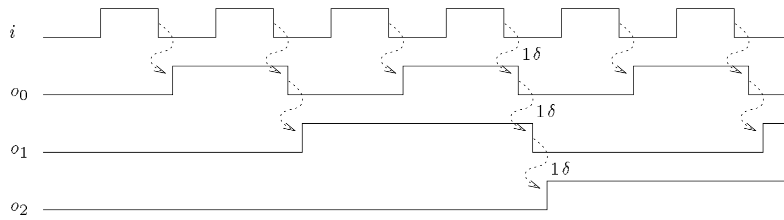

Figure 2 Timing of the Asynchronous Counter

It suggests that whenever the input switches from 1 to 0 the counter value is increased and the new output becomes stable after 3δ steps. We will see later how this behaviour can be derived formally from our semantics.

Exercise *It is not difficult to modify the example so as to obtain an asynchronous counter stage with a delay of $n\,\delta$ steps. Formulate both inertial behaviour, i.e. any input change arriving while the stage is unstable is ignored, and transport behaviour, i.e. every negative input change results in an output change after $n\,\delta$ steps. Hint: Use additional internal variables to count time and, if necessary, interfering input changes. Observe that the local counting of time depends on an empty sensitivity list, i.e. the fact that the process actively participates in every δ step.*

Another variant of the counter is the synchronous one where we take the δ steps to represent the synchronous computation determined by a global clock signal. Here, the clock signal is implicit in the activation of the processes; every time the implicit master clock ticks to increment synchronous time the next computation step is set off by activating all processes. A naive description of a synchronous counter is given in Figure 3 and a timing diagram shown in Figure 4.

The process $cnt_k\,(i_0, i_1, \ldots, i_k, o)$ describes the counter stage k with $k+1$ inputs $i_0, i_1, \ldots, i_k$ and output o. The inputs are connected to the outputs of the previous stages to detect the count overflow: if all earlier stages are at '1' the output must switch. Again, there is an implicit '**wait for** 0 *ns*' at the end of the processes.

```
cnt_k (i_0, i_1, ..., i_k, o) :
process
  variable z : BIT := '0';
begin
  if i_0 = '1' and ··· and i_k = '1'
    then z := not z else null;
  o <= z ;
end process;
```

```
cnt_3 (i, o_0, o_1, o_2) :
design
begin
  cnt_0 (i, o_0)
  cnt_1 (i, o_0, o_1)
  cnt_2 (i, o_0, o_1, o_2)
end design;
```

Figure 3 A Simple Synchronous Counter

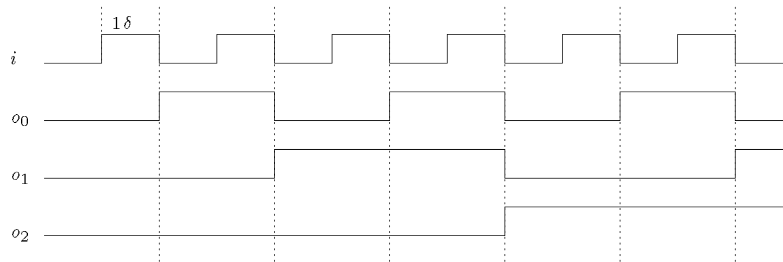

Figure 4 Timing of the Synchronous Counter

3 ASSUMPTIONS

The semantics of VHDL is specified by the language reference manual [66] in terms of an event-driven simulator. The principal aim is to provide a well-defined reference semantics for consistent and efficient simulator implementations. A natural approach towards formalizing the standard, therefore, would attempt the formal description of a VHDL simulator. Several attempts have been made in this direction. An elegant approach based on concurrent evolving algebras is that of Börger *et al.*, which can be found in Chapter 4 of this volume. Another example is the rather comprehensive formalization of the VHDL simulation engine in terms of coloured Petri nets by Olcoz in Chapter 5.

Clearly, the direct formalization of the standard supports the implementation of VHDL simulators and development of the VHDL language itself. However, from the users' point of view the semantics of a VHDL design appears as the semantics of processes and their communication rather than that of the simulator. Therefore, being interested in VHDL as a specification language, we will be a bit more abstract and formalize the *result* of a simulation not the simulation itself.

We develop a formal VHDL semantics for VHDL users

VHDL processes will be modelled formally as streams and stream-processing functions, which are the basic concepts in Focus. The starting point for our semantics is a sea-of-processes plus connectivity information, which is obtained during the elaboration phase [66]. Each process, which stands for a concurrent statement or a collection of sequential statements, is modelled by a stream-processing function transforming an input stream into an output stream. The connections between processes are reflected by channels. Signals correspond to streams of messages sent along the channels. Contrary to VHDL's simulator model, where only events are sent around, a message in our model is the signal state observable at a particular delta time. Consequently, suspension and resumption of processes becomes unnecessary – a process must be alive all the time. Also, wait statements are assumed to be busy and built into the process. 'Built in' means that the wait statement does not appear explicitly in our semantic functions and 'busy' means that the process is responsible for checking the input signals for being activation events. This allows us to abstract from the event-driven simulation cycle. Eventually our system model for VHDL semantics looks as follows:

System model:

- process → stream-processing function
- connections → channels
- signals → streams of messages
- signal data → abstract data type

The system model above has a decisive influence on the statements we give a formal semantics for. Obviously we need formal semantics for processes. The

concurrent VHDL statements are transformed into processes during elaboration, hence we can skip the explicit handling of these statements as they are already covered by processes. However for the sequential statements, which are the ingredients of processes we need semantic functions. To concentrate on the principal idea behind our semantic functions and to keep the paper concise we restrict ourselves to a subset of the sequential statements. We exclude assertions, procedures and procedure calls, the case statement, as well as next, exit and return. We also ignore arbitrary wait statements but introduce an implicit '**wait for** $0\ ns$' at the end of each process.

Formal semantics is given to:

- processes based on busy waiting
- an adequate subset of sequential statements

One powerful feature of VHDL are the timing concepts. Nevertheless we claim that delta-delay timing is sufficient to understand the VHDL statements. Of course, inertial and transport delay have influence on the transmission of signals, but the semantics of the statements itself is independent from timing. As a by-product of delta-delay handling we need not take into consideration preemptiveness, and waveforms as well as after clauses in signal assignments.

Delta-delay is sufficient to give semantics to VHDL statements

Since a signal in VHDL is accessible to all the processes, it may be driven by many different sources. VHDL requires each such signal to be resolved by a resolution function. In our modelling the resolution functions are just another type of process in the sea-of-processes. On the input side they are connected to the different driving sources and on the output-side the resulting effective values are accessible.

Resolution functions are modelled by processes

Our semantics evades a number of syntactic features of VHDL. Some omissions have not been mentioned like signal attributes and null waveform elements. For a concise definition of exactly what has been covered the reader is referred to the appendix which contains the abstract syntax definition of δVHDL. We wish

to stress, however, that all restrictions — except for omitting wait statements and the after clauses — are not essential to our approach in the sense that they have been made only for convenience and simplicity rather than intrinsic technical reasons.

3.1 The Specification Methodology Focus

The formal setting in which our semantics is developed is Focus [28], a general framework for the formal specification and development of distributed systems. The suitability of Focus in the area of system and software design has been demonstrated by many case studies [27]. Focus has already been used successfully to give formal semantics to existing languages like SDL [26, 64] and Esterel [75]. Recently, some efforts have been made in applying Focus in the area of hardware design [53, 52, 44].

A system in Focus is modelled by a network of components working concurrently, and communicating via FIFO channels. A number of reasoning styles and techniques are available. Focus also provides mathematical formalisms which support highly abstract, not necessarily executable, specifications with a clear semantics and offers powerful refinement calculi to develop distributed systems in a stepwise manner. Focus is modular in the sense that design decisions can be checked at the point where they are taken, component specifications can be developed in isolation, and in the sense that already completed developments can be re-used in new program developments. The mathematical foundation of Focus is based on streams, stream-processing functions and functional specifications by means of predicates.

Streams. A *stream* is a finite or infinite sequence of data modelling the history of a communication channel in terms of the sequence of messages sent along the channel. The application of streams as a general model for distributed computing has been introduced in [76] and for digital circuits in [43]. In this paper streams are used in the specific context of VHDL: channels represent VHDL signals and messages represent signal values in a given δ-step. The flow of time inherent in this notion of a stream, then, represents the succession of δ-steps effected by the simulation cycle. In our model, which abstracts from the finiteness of VHDL's simulation time (`TIME'High`), we assume that simulation runs on forever. Thus, the streams we are actually interested in are the infinite streams. A finite stream is an approximation corresponding to an incomplete simulation which has determined the history of a system only up to a certain number of δ steps. Its continuation is unknown and may in fact

never be attained since the computation for the next δ-state may be trapped in an infinite loop.

The formal setting is as follows. Given a set of actions D, D^* denotes the set of all finite streams generated from D; D^∞ denotes the set of all infinite streams generated from D, and D^ω, called the set of *streams* over D, is the union $D^* \cup D^\infty$. The few fundamental operations on streams sufficient for our purposes are shown in Figure 5. Many more useful operations can be defined to obtain a rich algebra of streams [28, 43]. When we write down

$\langle\,\rangle$	empty stream
$d \,\&\, y$	stream resulting from prefixing stream y with element $d \in D$
$zip(y_1, \ldots, y_n)$	stream obtained by merging n streams y_i into a single stream of n-tuples
$unzip^n_k(x)$	stream obtained by extracting the k-th component from stream x of n-tuples ($n \geq k$)
$\#x$	length of stream x (∞ if x is infinite)

Figure 5 Basic Operations on Streams.

streams we use angled brackets and let time flow from left to right, *e.g.* the stream $\langle 7, 8, 9, 10, \ldots \rangle$ describes the behaviour of a counter starting with 7 at the beginning of the simulation. The prefixing operation $d \,\&\, y$ adds an element $d \in D$ at the start of stream y, *e.g.* $5 \,\&\, \langle 0, 1 \rangle = \langle 5, 0, 1 \rangle$. The *zip* function synchronizes n streams into a single stream of n tuples, *e.g.*

$$\begin{aligned} zip(\langle 7, 8, 9, 10, \ldots \rangle, \langle \text{“}ready\text{”}, \text{“}steady\text{”}, \text{“}go\text{”} \rangle) &= \\ \langle (7, \text{“}ready\text{”}), (8, \text{“}steady\text{”}), (9, \text{“}go\text{”}) \rangle. \end{aligned}$$

Observe that the output of *zip* stops as soon as one of its argument streams does. This models the strong synchronization of VHDL's simulation kernel: If one process fails to terminate, *i.e.* a statement runs into an infinite loop, the whole simulation is blocked. With *unzip* we can break up a stream of tuples into slices, *e.g.*

$$unzip^2_1(\langle (7, \text{“}ready\text{”}), (8, \text{“}steady\text{”}), (9, \text{“}go\text{”}) \rangle) = \langle 7, 8, 9 \rangle.$$

Observe that *unzip* is not the inverse of *zip* since the synchronization performed by *zip* throws away information that cannot be recovered. Finally, the length of a stream is the number of actions contained in it, for instance $\#\langle 3, 8, 1003, 0 \rangle = 4$, $\#\langle 0, 1, 2, \ldots \rangle = \infty$. The notion of length of a stream will be generalized to

tuples of streams: If r is a tuple of streams, then $\#r$ denotes the length of the shortest stream in r. For example, $\#(\langle 2,5,8\rangle, \langle\,\rangle, \langle 1\rangle) = 0$.

On D^ω we define a partial prefix ordering $s \sqsubseteq r$, which denotes that s is a prefix of t. Intuitively, $s \sqsubseteq t$ says that simulation history t is a continuation of history s. Alternatively, $\sqsubseteq$ can also be understood as an information ordering so that $s \sqsubseteq t$ means s is an approximation of history t which gives less information about the whole simulation run than t. An extreme case is the empty stream $\langle\,\rangle$ which carries no information at all. While not being of much practical relevance the empty stream is an important theoretical concept. It plays a double rôle of modelling the beginning of a simulation, *i.e.* the empty history, as well as a simulation stopped by a nonterminating statement: Whenever the result of executing a statement is undefined, which we will denote as $\perp$, the simulation stream is empty. Technically, this is achieved by taking & to be strict in its first argument and non-strict in its second, *i.e.* $d \,\&\, y$ is empty iff d is undefined.

Remark *The set of streams ordered by $\sqsubseteq$ is a complete partial order (cpo) with the empty stream $\langle\,\rangle$ as least element. This algebraic structure can be used for solving recursive type equations [122]. For instance, one can show that D^ω is a canonical solution of the cpo-isomorphism*

$$D^\omega \;\cong\; (D \times D^\omega)_\perp,$$

where $X_\perp$ denotes X extended with a fresh bottom (=least) element $\perp \notin X$, i.e. $X_\perp = X \cup \{\perp\}$, and D is a set with discrete ordering. Being a 'canonical' solution means that we can define recursive functions over D^ω by induction on the structure of its elements. For instance, we know that a recursive definition like

$$f((d_1, d_2, d_3) \,\&\, rest) = d_2 \,\&\, f(rest)$$

determines a unique function $f \in (D \times D \times D)^\omega \to D^\omega$, namely the function $unzip^3_2$. For the mathematical theory of cpos and their application as semantical data types the reader is referred to [110] or the more recent textbooks [57, 122].

Stream Processing Functions. VHDL processes will be modelled by stream-processing functions. A stream-processing function is a continuous mapping from tuples of input streams to tuples of output streams [28]. In concrete terms this means that the function's behaviour for infinite inputs is completely determined by its behavior for finite inputs. This is a reasonable requirement: every output a VHDL process produced in a full simulation run must have occurred after a finite number of inputs already. All our basic operations on streams $d\&$, *zip*, *unzip* are continuous.

Remark *Continuous functions f over complete partial orders (cpo) with a least element have unique minimal fixpoints which can be completely characterized by the*

iterations of f. This property is the basis for describing the behaviour of a system of VHDL *input/output processes with feedback functionally by fixpoint theory, in perfect concordance with the iterative simulation model. For the mathematical theory of continuous functions and their rôle in solving recursive equations the reader is referred to [110, 57, 122]. The special case of streams and stream-processing functions, as well as their application to system design is elaborated on in [25, 76, 43].*

A special feature which our stream model for VHDL is distinguished by is the synchronous communication paradigm. The simulation of δ-steps in VHDL is a completely synchronous execution process in which the individual processes and the kernel communicate synchronously. This can be captured formally: A function $f \in D_1^\omega \times \cdots \times D_n^\omega \to D^\omega$ is called *weakly synchronous* if it has the property

$$\forall i, j \in D_1^\omega \times \cdots \times D_n^\omega : \; i \sqsubseteq j \wedge \#f(i) < \#i \;\Rightarrow\; f(i) = f(j).$$

Intuitively, the condition states that whenever f has taken more input elements than it has produced output elements, then f has broken down and will never produce any more output. This is a characteristics of synchronous systems — viewed as functions on streams — as opposed to asynchronous ones, which may first need to consume a certain number of input values before producing the next output. More specifically, in a synchronous system the output is ahead of the input by a constant amount of values reflecting the constant delay through the system's registers. Since our condition above also permits that this internal delay varies over time and even becomes infinite we call it the *weak* synchrony property.

Adopting weak synchrony has the advantage that communication channels behave as ordinary wires rather than unbounded FIFO buffers. Notice that all our basic operations on streams *d&*, *zip*, *unzip* are weakly synchronous, and that this property is preserved under functional composition.

Specification. In general a process may be modelled by a (non-empty) set of weakly synchronous stream-processing functions. Each function from this set corresponds to one particular (deterministic) behaviour. Let S be a component with input streams $D_1^\omega \times \cdots \times D_n^\omega$ and output stream D^ω. Then the behaviour of S, or its semantical denotation, is a logical predicate

$$[\![S]\!](f \in D_1^\omega \times \cdots \times D_n^\omega \to D^\omega)$$

describing a set of weakly synchronous stream-processing functions f that represent the behavior of the specified component. If the set is empty, the specification is called *inconsistent*. If the set consists exactly of one function, the

specification is called *determined*. If the set contains more than one function, the specification is an *under-specification* since more than one implementation is possible. In our description of VHDL all specifications will be determined.

To give an example of a specification in Focus, let us specify a zero-delay level-triggered d-latch DL, taking $\mathbb{B} = \{0, 1\}$ to denote the set of bits:

$$
\begin{array}{ll}
[\![DL]\!](f \in \mathbb{B}^\omega \times \mathbb{B}^\omega \rightarrow \mathbb{B}^\omega) \; \equiv & \\
\quad \exists g \in \mathbb{B}^\omega \times \mathbb{B}^\omega \times \mathbb{B} \rightarrow \mathbb{B}^\omega : \exists z \in \mathbb{B} : & (0) \\
\quad\quad \forall i_1, i_2 \in \mathbb{B}^\omega : f(i_1, i_2) = g(i_1, i_2, z) \wedge & (1) \\
\quad\quad \forall ds, cls \in \mathbb{B}^\omega : \forall d, z \in \mathbb{B} : & \\
\quad\quad\quad g(d \,\&\, ds, 1 \,\&\, cls, z) = d \,\&\, g(ds, cls, d) & (2) \\
\quad\quad\quad g(d \,\&\, ds, 0 \,\&\, cls, z) = z \,\&\, g(ds, cls, z) & (3)
\end{array}
$$

The latch has a clock and a data input, and a data output, so the function f to be specified is of type $\mathbb{B}^\omega \times \mathbb{B}^\omega \rightarrow \mathbb{B}^\omega$. As seen in (1) we specify f in terms of an auxiliary function $g \in \mathbb{B}^\omega \times \mathbb{B}^\omega \times \mathbb{B} \rightarrow \mathbb{B}^\omega$, which has a parameter list extended by an extra 'state parameter' z of type $\mathbb{B}$. The auxiliary function is introduced in the specification by the existential quantifier $\exists g$ in (0).

The basic behaviour of the latch is to transfer the input to the output as long as the clock is high. If the clock goes low the last transferred input data is held at the output. The initial data $z \in \mathbb{B}$ for the output is chosen nondeterministically, either low or high, by the $\exists z$-quantifier in (0) – the corresponding data only appears at the output if the first clock element is low. In (2) the first element at the data input is transferred to the output because the first clock element is high. In (3) the first clock element is low and therefore the old output data, stored in the state variable z, is output again. In both cases the auxiliary function invokes itself with reduced inputs and an accordingly manipulated state variable.

In Focus we usually avoid notational ballast and simplify our specifications by dropping the quantifiers whenever these can be worked out systematically from the structure of the equational specification: the rule is that, by reading the equations from top to bottom, all variables appearing on the left-hand side of an equation are universally quantified while those only occurring on the right-hand side are introduced by existential quantification. To structure the specification a 'where' statement is sometimes introduced to indicate a subspecification. Adopting this scheme the specification of the latch simply becomes

$$\begin{aligned} [\![DL]\!](f \in \mathbb{B}^\omega \times \mathbb{B}^\omega \to \mathbb{B}^\omega) \;\equiv\; & f(i_1, i_2) = g(i_1, i_2, z) \\ \text{where } & g(d \,\&\, ds, 1 \,\&\, cls, z) = d \,\&\, g(ds, cls, d) \\ & g(d \,\&\, ds, 0 \,\&\, cls, z) = z \,\&\, g(ds, cls, z). \end{aligned}$$

For a more detailed introduction into Focus see [28].

4 FORMAL SEMANTICS FOR δ-VHDL

The discussion of our formal semantics for δVHDL proceeds bottom up. Beginning with the identifiers we work our way up via expressions and sequential statements to concurrent processes. These are the basic levels of computation each one with its own characteristic notion of execution, time, and composition.

VHDL as almost any other programming language rests on the notion of identifiers as the most primitive syntactic entity. One might, at first, expect that there is not much to be said here, but in fact the structure pertaining to VHDL identifiers is fairly rich already and deserves separate mention.

First, VHDL is a typed language which means that identifiers are distinguished by their types. Formally, we have a set of *identifiers* $\mathcal{N} = \{x, y, z, \ldots\}$, a set of *type names* $\mathcal{T} = \{\alpha, \beta, \gamma, \ldots\}$ together with a map that associates a (unique) type name $type(x) \in \mathcal{T}$ with every identifier $x \in \mathcal{N}$. To indicate the typing an identifier x can be decorated with its associated type name α to produce a *type indication* $x : \alpha$. Second, identifiers are structured further in VHDL by distinguishing, among other things, between *variables* and *signals*. Signals in turn have an associated *mode* to distinguish, for instance, between *input* and *output* signals. There is much more structure to signals and identifiers (the 'names') in VHDL which we shall not treat in this paper. For our purposes it is sufficient to assume a fixed subdivision of $\mathcal{N}$ into variables $\mathcal{V}$ and signals $\mathcal{S}$, *i.e.* $\mathcal{N} = \mathcal{V} \cup \mathcal{S}$, $\mathcal{V} \cap \mathcal{S} = \emptyset$, and further distinguished subsets $\mathcal{I} \subset \mathcal{N}$ and $\mathcal{O} \subset \mathcal{N}$ of input and output signals. We deliberately permit $\mathcal{I}$ and $\mathcal{O}$ to overlap providing us with signals of mode inout. The fact that we fix these sets once and for all is not a restriction since they can be chosen large enough (*e.g.* infinite) for a practically inexhaustible supply of fresh identifiers.

Just as the syntax is generated by identifiers the semantics will build up from a semantics for the identifiers. We assume that there is associated with every type name $\alpha \in \mathcal{T}$ a concrete data type $[\![\alpha]\!]$ containing all the values we will

want to denote by identifiers of type α. For instance, among the type names we would have INTEGER and BIT such that $[\![\text{INTEGER}]\!]$ and $[\![\text{BIT}]\!]$ are the set of integers $\{\ldots-2,-1,0,1,2,\ldots\}$ and bits $\mathbb{B}=\{0,1\}$, respectively. Let $Id=\{x_1:\alpha_1,\ldots,x_n:\alpha_n\}$ be a set of identifiers with type indications. Then, the semantics of Id is the set of records r with labels $x_1,\ldots,x_n$, such that component $r.x_i$, $i=1,\ldots,n$, is an element in the set $[\![\alpha_i]\!]$, formally

$$[\![Id]\!] \quad = \quad \{x_1 \in [\![\alpha_1]\!],\ldots,x_n \in [\![\alpha_n]\!]\}.$$

Given a record $r \in [\![Id]\!]$ we write $r\{x \leftarrow a\}$ for the record obtained from r by updating entry x with a new value a:

$$r\{x \leftarrow a\}.z \quad = \quad \begin{cases} a & \text{if } z = x \\ r.z & \text{otherwise.} \end{cases}$$

For example, if $Id=\{i:\text{BIT},o:\text{INTEGER}\}$, then $r=\{i=0,o=1993\}$ is a record in $[\![Id]\!]$ with $r.i=0$, $r.o=1993$; $r\{o \leftarrow 1994\}$ is the record $\{i=0,o=1994\}$. The record interpretation is very convenient as it allows us to use the identifiers of δVHDL directly to reference semantical objects. When the names of variables, and hence those of the record fields, are understood we may identify $[\![Id]\!]$ with the product set $[\![\alpha_1]\!] \times \cdots \times [\![\alpha_n]\!]$.

4.1 Expressions

The first semantic level to be treated corresponds to the syntactic class of expressions. In formalizing this level we will be fairly generic in the sense that we do not stick too much to the concrete syntax of VHDL but rather focus on the essential concepts involved. We describe the abstract syntax of expressions so that we can ignore syntactic mechanisms such as type overloading, type conversion, and operator precedences. Also, to keep matters simple we shall not treat type constructions such as subtypes, enumeration types, *etc.*

Expressions in VHDL as in many other programming languages are built from a set of primitive operations which are thrown in and implemented directly by the run-time system. Examples are the subtraction operation or the logical operation *xor*. Since VHDL is a typed language operations can be applied only to values of a certain data type and produce results only of a certain type. For instance, subtraction is not allowed on booleans while *xor* is not admissible on integers.

Formally, the syntax of expressions may be specified by two bits of data: a set of *operator names* $\mathcal{O}=\{f,g,h,\ldots\}$ and a map associating with every

operator f a finite nonempty sequence of type names $\alpha_1\alpha_2\cdots\alpha_n\beta$, called the *arity* of f. For instance, there will be the operators subtraction and exclusive-or with arities INTEGER INTEGER INTEGER and BIT BIT BIT, respectively. Operators whose arity is a single type name are called *constants*. For example, '0', '1' are constants of arity BIT. The arity of an operator specifies in which contexts it can be applied: if f has arity $\alpha_1\alpha_2\cdots\alpha_n\beta$, then f can be applied to n arguments $g_1, g_2, \ldots, g_n$ of types $\alpha_1, \alpha_2, \ldots, \alpha_n$ to obtain a *well-formed* term $f(g_1, g_2, \ldots, g_n)$ of type β. By this inductive process the well-formed expressions are built up from the operators. The primitive elements are the identifiers and constants: an identifier $x : \alpha$ is a well-formed expression of type α. A constant of arity α is a well-formed expression of type α. The identifiers occurring in an expression e are called the *free* identifiers of e.

The semantics of expressions is defined by associating with every operator $f \in \mathcal{O}$ of arity $\alpha_1\alpha_2\cdots\alpha_n\beta$ a concrete operation $[\![f]\!] \in [\![\alpha_1]\!]\times[\![\alpha_2]\!]\times\cdots\times[\![\alpha_n]\!] \to [\![\beta]\!]$, where $[\![\alpha_i]\!]$ and $[\![\beta]\!]$ are the concrete data types associated with the type names α_i and β, $i = 1, \ldots, n$. For instance, $[\![xor]\!] \in \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ might be the concrete exclusive-or operation on bits implemented in the run-time system of the VHDL simulator. The semantics of a well-formed expression e of type α with free identifiers contained in $I = \{x_1 : \alpha_1, \ldots, x_n : \alpha_n\}$ is the function

$$[\![e]\!] \in [\![I]\!] \to [\![\alpha]\!]$$

inductively defined as follows:

$$\begin{aligned} [\![x_i]\!](\{x_1 = a_1, \ldots, x_n = a_n\}) &= a_i \qquad (i = 1, \ldots, n) \\ [\![f(g_1, \ldots, g_m)]\!](r) &= [\![f]\!]([\![g_1]\!](r), \ldots, [\![g_m]\!](r)). \end{aligned}$$

Note that both variables and signals are identifiers and thus are covered by the above definition. For the semantics of expressions no distinction between variables and signals is made.

The formalization of the above concepts at the level of expressions is well known from semantic algebras [110]. Clearly, there is a trade-off of just how much computational power one bothers to put into this algebraic level. This is not just a matter of what primitive data types and primitive operations there should be available but also how much extra algebraic structure one decides to put in. For instance, in VHDL there are derived data-types such as subtypes and enumeration types, and general recursive functions. One might go even further to allow inductive datatypes and function types as in high-level functional programming languages. Though still maintaining the essential algebraic nature (leaving aside the problem of nontermination), this would get us well beyond the scope of a traditional hardware description language.

4.2 Statements

The second semantical level is that of the sequential statements, which make up the algorithmic kernel of VHDL. At this level VHDL appears much like an ordinary imperative programming language such as Pascal or C, with statements such as variable assignment, if-then-else, and loop. Therefore, in a first approximation towards a formal semantics for VHDL, which is concerned with the VHDL-specific aspects of signals and processes, this level can be ignored. This approach is taken, for instance, in the formal semantics presented in Chapters 3 and 2 of this volume. Yet, if one is to capture VHDL as an imperative programming language the sequential part, which provides all the actual algorithmic power, is essential. It is the meat in the soup.

As in ordinary procedural languages the execution of a statement amounts to a sequence of changes of program state, *i.e.* in the values of variables declared in the current local context. Accordingly, the semantics can be based on the notion of state transitions obtained from the sequential execution of the program statements. Among these are the loop statements, which sometimes may not terminate resulting in an infinite state sequence. Whenever it does terminate a statement may be viewed as a transformation of the initial start state to the final end state. Formally, the semantics of a statement with variables V in first approximation would be a partial function f with domain $[\![V]\!]$ and range $[\![V]\!]$,

$$f \in [\![V]\!] \to [\![V]\!]_{\perp},$$

where $\perp$ indicates that f may not terminate for some inputs.

The salient new concept in VHDL is the signal assignment, which links up with a new semantical level unknown in procedural languages, namely the level of signals and processes. We must refine the traditional semantics of statements given above to capture this extra structure. First we need to fix our notion of signal. Formally, if $s : \alpha$ is a signal identifier for a signal of type α, we need to define a mathematical object $Sig[\![s : \alpha]\!]$ representing the semantics of signal s. There are many sensible choices and the more information we decide to put into $Sig[\![s : \alpha]\!]$ the more powerful our notion of signal and the more expressive our semantics of statements will be. For instance, as in [106] we might take a signal to be a list of values of type α modelling a history of values over physical time (measured in femtoseconds, say) and put $Sig[\![s : \alpha]\!] = [\![\alpha]\!]^*$. This allows us to capture VHDL signal attributes such as '*delayed* or '*stable* as operations on lists [106], but not δ-delays. If we wanted to model signal attributes also in the presence of δ delays this is not enough. In this case we might enrich the structure to a list of time-stamped values, *i.e.* $Sig[\![s : \alpha]\!] = ([\![\alpha]\!] \times \mathbb{N})^*$, using the natural numbers $\mathbb{N}$ to represent physical time. The list ordering now

represents δ time while physical time is part of the signal value. Clearly, there is no limitation as to how much information we can put into the notion of signal. In this paper, where we do not consider signal attributes and physical time we can get away with the simplest possible semantics, *i.e.* $Sig[\![s : \alpha]\!] = [\![\alpha]\!]$, or more generally for a set $S = \{s_1 : \alpha_1, \ldots, s_n : \alpha_n\}$ of signal identifiers

$$Sig[\![S]\!] \quad = \quad [\![S]\!] \; = \; \{s_1 \in [\![\alpha_1]\!], \ldots, s_n \in [\![\alpha_n]\!]\}.$$

Thus, as far as the execution of sequential statements is concerned, the semantics of a set of signals is given by a record of single values, representing the signal state at the present δ step, *i.e.* the one in which the statement is executed.

Given the notion of signal we can proceed to the signal assignment. A signal assignment unlike the variable assignment does not change the state of variables but has a side-effect of scheduling a transaction on a particular signal. In the course of executing a statement or a sequence of statements a number of such transactions for a number of different signals can be accumulated. These lists of accumulated transactions may be called *transaction traces* or *traces* for short. Let $O = \{s_1 : \beta_1, \ldots, s_m : \beta_m\}$ be a set of output signal identifiers. Then we will need a semantical object $Tr[\![O]\!]$ to represent the traces over O. Again, there are many possibilities of such structures and each particular choice will model the semantics of signal assignments at different granularity and detail. In a very detailed semantics with full physical time signal assignments of the inertial and transport type a transaction for an output signal $s : \beta$ would need to record at least three bits of information: the value assigned, the value of the delay time, and whether it is an inertial or transport delay. For the total statement execution a list of such transactions would have to be recorded in the order in which they occur. So, in the case of a single signal $s : \beta$ the notion of trace might become $Tr[\![s : \beta]\!] = ([\![\beta]\!] \times \mathbb{N} \times \{inertial, transport\})^*$. On the other end of the scale, in our simple case where we only consider δ-delay signal assignments, all we need to record as a transaction for a signal is the value assigned, leading to the following definition:

$$Tr[\![O]\!] \quad = \quad \{\; s_1 \in [\![\beta_1]\!]^* \;, \ldots, \; s_m \in [\![\beta_m]\!]^* \;\}.$$

Thus, a transaction trace tr accumulates for each output signal s_i, $i = 1, \ldots, m$, a list $tr.s_i$ of transaction values, each of which is a value in $[\![\beta_i]\!]$. When no assignment to s_i occurs, $tr.s_i$ is the empty list, filled in as a 'padding' element. For the semantics of statements it is irrelevant how these traces are eventually translated into projected waveforms and effective signal values. This is determined by drivers and resolution functions which are introduced at the next semantical level of processes and systems of processes.

It is worth pointing out that other approaches to a formal semantics for VHDL, such as, for instance, [98, 106] or Van Tassel's semantics in Chapter 3, do not use an independent notion of transaction trace. The drivers are implicitly built into the execution of signal assignments. Here we take the stand that drivers are not intrinsic to the semantics of statements and should be kept as a separate concept.

A bit of auxiliary notation regarding traces will prove helpful. The empty transaction trace over O is denoted by Λ_O, *i.e.* $\Lambda_O = \{s_1 = [], \ldots, s_m = []\} \in Tr[\![O]\!]$. If tr_1 and tr_2 are traces over O we define their point-wise concatenation $tr_2 \mathbin{@} tr_1$ as follows: For all $i = 1, \ldots, m$,

$$(tr_2 \mathbin{@} tr_1).s_i \quad = \quad tr_2.s_i \circ tr_1.s_i,$$

where $\circ$ on the right hand side is the Focus notation for the concatenation of lists.

Suppose *stmt* is a well-formed statement in the context of variable identifiers V, input signal identifiers I, and output signal identifiers O. The semantics of *stmt*, as seen so far, will be a partial function

$$f \;\in\; [\![V]\!] \times [\![I]\!] \;\rightarrow\; ([\![V]\!] \times Tr[\![O]\!])_\perp,$$

noting that in our case $Sig[\![I]\!] = [\![I]\!]$. Compare this with the original type $[\![V]\!] \rightarrow [\![V]\!]_\perp$: our statement semantics now distinguishes between variables and input signals in the function argument and between a variable state and an output trace in the function result.

We can now list the semantic clauses for all sequential statements of δVHDL, where as in the case of expressions the definition proceeds by recursion on the syntactic structure. The definition can be given in many ways. Here we adopt the style of Focus [28] in presenting, for each type of statement *stmt*, a predicate $[\![stmt]\!]\,(f \in [\![V]\!] \times [\![I]\!] \;\rightarrow\; ([\![V]\!] \times Tr[\![O]\!])_\perp)$ that specifies the partial function f representing the behaviour of *stmt*. The following general pattern is adopted:

$$[\![stmt]\!]\,(f \in [\![V]\!] \times [\![I]\!] \;\rightarrow\; ([\![V]\!] \times Tr[\![O]\!])_\perp) \quad \equiv \quad Q(f),$$

where Q is a logical predicate with a single free variable f. This semantical style could be called property-oriented since we are constraining the intended semantical functions by a logical predicate rather than constructing the functions themselves. In our case the difference is inessential and one can verify that the definitions presented below in fact define a unique partial function for every statement of δVHDL. The mathematical theory is well documented and can be found in many textbooks, such as, for instance [122].

If-Statement. The result (var', tr) of executing **if** *cond* **then** $stmt_1$ **else** $stmt_2$ **end if** starting from the state (var, sig) is obtained by first evaluating the condition *cond* in the context (var, sig); if *cond* is `true` then $stmt_1$ is executed, if *cond* is `false` then $stmt_2$ is executed.

$$[\![\textbf{if } cond \textbf{ then } stmt_1 \textbf{ else } stmt_2 \textbf{ end if}]\!]\,(f) \;\equiv$$
$$f(var, sig) \;=\; if\,[\![cond]\!](var, sig)\ then\ f_1(var, sig)\ else\ f_2(var, sig)$$
$$\text{where } [\![stmt_1]\!]\,(f_1) \;\wedge\; [\![stmt_2]\!]\,(f_2).$$

Recall our conventions for reading the equational specification: variables var, sig are to be universally quantified while f_1, f_2 are to be existentially quantified.

Signal Assignment Statement. A signal assignment s `<=` *expr* started in state (var, sig) first evaluates the expression *expr* and then the resulting value $[\![expr]\!](var, sig)$ is recorded in the transaction trace for signal s. No other transaction is produced, so the resulting trace is $\Lambda_O\{s \leftarrow [\ [\![expr]\!](var, sig)\ ]\}$. The variables are left unchanged. Recall that $r\{x \leftarrow a\}$ is our notation for updating entry x of record r with value a.

$$[\![s \texttt{ <= } expr]\!](f) \quad\equiv\quad f(var, sig) \;=\; (var\ ,\ \Lambda_O\{s \leftarrow [\ [\![expr]\!](var, sig)\ ]\}).$$

Variable Assignment Statement. The variable assignment v `:=` *expr* assigns the value $[\![expr]\!](var, sig)$ to variable v and leaves all other variables unchanged. No transaction is produced, whence the resulting trace is the empty trace Λ_O.

$$[\![v \texttt{ := } expr]\!](f) \quad\equiv\quad f(var, sig) \;=\; (var\{v \leftarrow [\![expr]\!](var, sig)\}\ ,\ \Lambda_O).$$

Null Statement. The **null** statement has trivial effect. It doesn't change the variable state and does not schedule any transaction.

$$[\![\textbf{null}]\!](f) \quad\equiv\quad f(var, sig) \;=\; (var\ ,\ \Lambda_O).$$

Sequential Composition. A sequential composition $stmt_1; stmt_2$ is executed by first executing $stmt_1$, say in the start state (var, sig). The result is a pair (var', tr_1) with the new variable state var' and a transaction trace tr_1. Then, $stmt_2$ is executed in state (var', sig) producing a variable state var'' and a second trace tr_2. The final variable state, then, of $stmt_1; stmt_2$ is var'' and the total trace is the point-wise concatenation $tr_2@tr_1$. This process is nothing but a special form of functional composition: If f_1 and f_2 are two functions

of type $[\![V]\!] \times [\![I]\!] \rightarrow [\![V]\!] \times Tr[\![O]\!]$ we define their "sequential" composition $f_1 \; ; \; f_2 \in [\![V]\!] \times [\![I]\!] \rightarrow [\![V]\!] \times Tr[\![O]\!]$ as follows:

$$(f_1 \; ; \; f_2)(var, sig) \;=\; (var'', tr)$$
$$\text{where } (var', tr_1) = f_1(var, sig) \;\wedge\; (var'', tr_2) = f_2(var', sig) \;\wedge\; tr = tr_2 @ tr_1.$$

The semantics of $stmt_1; stmt_2$ then simply becomes:

$$[\![stmt_1; stmt_2]\!](f) \;\equiv\; f \;=\; f_1 \; ; \; f_2 \text{ where } [\![stmt_1]\!](f_1) \;\wedge\; [\![stmt_2]\!](f_2).$$

Notice, the only way sequential statements can exchange information is by the variables. The signals *sig* are only input, they are never changed, and the traces tr_1, tr_2 are only output, they are never read during the execution of a sequential statement.

Loop Statement. Finally, let *loop_stmt* be the statement **while** *cond* **loop** *body* **end loop**. We follow the usual pattern of explaining the behaviour of *loop_stmt* in terms of **if**, **null**, and sequential composition, namely by stipulating that $[\![loop_stmt]\!]$ is the least partial function satisfying the semantic equation

$$loop_stmt \;\cong\; \textbf{if } cond \textbf{ then } body \; ; \; loop_stmt \textbf{ else null}.$$

Formally,

$$\begin{aligned} &[\![loop_stmt]\!](f) \;\equiv \\ &\quad f(var, sig) \;=\; if \; [\![cond]\!](var, sig) \\ &\qquad\qquad then \; (b \, ; \, f)(var, sig) \;\; else \; n(var, sig) \\ &\quad \text{where } [\![\textbf{null}]\!](n) \;\wedge\; [\![body]\!](b), \end{aligned}$$

where we understand that f be the least partial function satisfying the condition.

Remark *The proviso to take the least partial function satisfying the defining property of the loop statement is necessary to ensure that* $[\![loop_stmt]\!](f)$ *actually defines the unique function that represents the operational execution of the statement. For instance, if loop_stmt is the trivial statement* **while** *TRUE* **loop null end loop** *the specification* $[\![loop_stmt]\!](f)$ *is equivalent to the condition* $f(var, sig) \;=\; f(var, sig)$ *which is satisfied by all partial functions. But only the least such solution, viz. the empty function, represents the actual execution of loop_stmt, viz. the nonterminating loop.*

Let us stop here our treatment of sequential statements. Of course, a number of VHDL features have not been covered. Standard imperative constructs such as procedures, case, exit, next statements, and for-loops present no real difficulty. The omission of transport and inertial delay assignments as well as the after clause are a consequence of our restriction to zero-delay. A more serious restriction is the omission of arbitrary wait statements which would require a nontrivial extension of our semantics. The reason is that upon entering a wait statement the program does not simply terminate but rather suspends execution. This means that the result of executing a statement is not just a final state of variables but also a continuation program. For standard operational semantics there are no problems (see *e.g.* [122]) but in our functional approach this is not entirely straightforward. The interested reader is referred to Chapter 2 of this volume where Breuer *et al.* present a functional treatment of arbitrary wait statements based on streams.

Remark *In order to treat arbitrary wait statements in a functional setting, basically we must employ a solution of the recursive type equation*

$$Stmt \;\cong\; [\![V]\!] \times [\![I]\!] \;\rightarrow\; ([\![V]\!] \times Tr[\![O]\!] \times Stmt)_{\perp}.$$

There are standard methods of solving such equations using domain theory but this is beyond the scope of this work.

The absence of wait statements does not effect any loss of expressibility. It is known that the wait statement can always be eliminated by adding extra state variables and if-then-else checks. A syntactic translation was given in [107, 108]. In view of this other functional approaches, like ours, simply omit the wait [98, 119].

In order to work through an example applying our semantical clauses let us consider the sequential statement of the asynchronous counter stage shown in Figure 1. The example involves all sequential statements of δVHDL except the loop statement. The internal variables here are $V = \{x : \text{BIT}, z : \text{BIT}\}$, the input signals $I = \{i : \text{BIT}\}$ and output $O = \{o : \text{BIT}\}$. We wish to evaluate the predicate $[\![cnt_1]\!](f \in [\![V]\!] \times [\![I]\!] \rightarrow [\![V]\!] \times Tr[\![O]\!])$. Unrolling the semantical definitions systematically one by one we can derive that

$$[\![\textbf{if } i = \text{`0'} \textbf{ and } x = \text{`1'} \textbf{ then } z \texttt{ := } \textbf{not } z \textbf{ else null} \,;\, x \texttt{ := } i \,;\, o \texttt{ <= } z \,;]\!]\,(f)$$

is equivalent to the following condition:

$$\begin{aligned} & f(var, sig) = \mathit{if}\ sig.i = 0 \wedge var.x = 1 \\ & \quad \mathit{then}\ (var\{z \leftarrow not(var.z)\}\{x \leftarrow sig.i\}\ ,\ \Lambda_O\{o \leftarrow [not(var.z)]\}) \\ & \quad \mathit{else}\ (var\{x \leftarrow sig.i\}\ ,\ \Lambda_O\{o \leftarrow [var.z]\}). \end{aligned}$$

If we identify $[\![I]\!], [\![O]\!]$ with $[\![\mathrm{BIT}]\!] = \mathbb{B}$, $[\![V]\!]$ with $\mathbb{B} \times \mathbb{B}$, and $Tr[\![O]\!]$ with $\mathbb{B}^*$ we can remove the explicit naming of signals and variables and simplify the condition to become

$$f(x,z,i) \quad = \quad \mathit{if}\ i = 0 \wedge x = 1\ \mathit{then}\ (i, not(z), [not(z)])\ \mathit{else}\ (i, z, [z]),$$

which is precisely what we expected.

Exercise *The fundamental difference between variables and signals is that in the execution of a statement variables store information while signals do not. This difference, which also may be expressed by saying that signals are static while variables are dynamic, can be turned into a precise mathematical statement: the order of signal assignments is immaterial: Let* e_1, e_2 *be arbitrary expressions and* $s_1 \neq s_2$ *distinct signals whose types are those of* e_1 *and* e_2 *respectively. Then,*

$$[\![s_1 \texttt{<=} e_1\ ;\ s_2 \texttt{<=} e_2\ ;]\!] \quad = \quad [\![s_2 \texttt{<=} e_2\ ;\ s_1 \texttt{<=} e_1\ ;]\!],$$

where $[\![A]\!] = [\![B]\!]$ *means* $\forall f.\ [\![A]\!](f) \Leftrightarrow [\![B]\!](f)$*. One can easily concoct a counter example showing that this property does not hold for variables, in general.*

Remark *Notice that in all cases the semantics of a composite statement is given in terms of the semantics of its immediate subprograms. Therefore, it is straightforward to verify that our semantics is compositional, i.e. if* $stmt_1$ *and* $stmt_2$ *are statements such that* $[\![stmt_1]\!] = [\![stmt_2]\!]$*, then* $[\![C[stmt_1]]\!] = [\![C[stmt_2]]\!]$ *where* $C[\cdot]$ *is any well-formed context. The proof is by induction on the structure of* $C[\cdot]$*.*

4.3 Processes

The semantics of processes as declared by the standard [66] involves a two-phase simulation cycle, starting from some initial state of variables and signals. In the first phase the active processes' sequential statements are executed until all terminate yielding new states for all (local) variables and transaction traces for the signals. In the second phase the transaction traces are translated into a new global signal state through the use of drivers and resolution functions. This cyclic process, which is effected by a global simulation kernel, defines a linear succession of strictly synchronous steps, the so-called δ steps. Thus, each δ-step corresponds to one pass through the simulation cycle. This admits a simple functional model based on streams and stream-processing functions. The general picture is as seen in Figure 6 and motivated in this section. The statements are translated into stream-processing functions from a stream of input signals to a stream of transaction traces on output signals. These output traces are transformed into a stream of driving values by drivers, and finally into a stream of effective signal values through a resolution process. Thus,

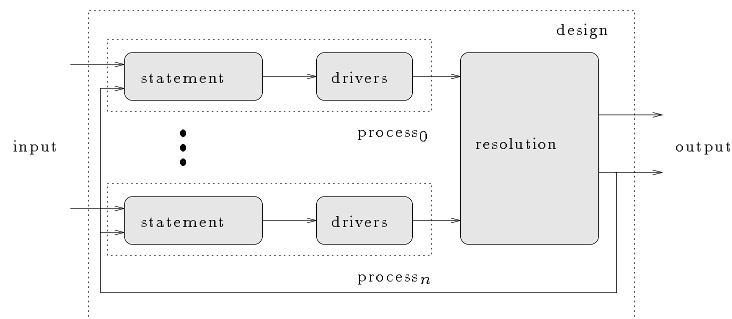

Figure 6 A Functional Model of VHDL

contrasting with the standard where drivers and resolution functions are parts of the global simulation kernel we model both as separate stream-processing functions; the drivers' special status owes to the fact that they are introduced by the semantics automatically rather than being specified by the programmer. Drivers are introduced with the **process** construct and resolution processes with the **design** construct, a new syntactic construct used in this work as a simple distillate of VHDL's bulky architecture and entity concept. Statement and drivers make up a process (this is indicated by dashed lines in Figure 6), a (number of) process(es) together with the resolution make(s) up a design. Loosely speaking, the overall system of Figure 6 embodies a synchronous system executing the simulation cycle. Statement and resolution play the rôle of the combinational part and the drivers represent the state-holding registers which separate one δ step from the next.

Our first goal is to give semantic meaning to a process *proc* such as

proc : **process**
 variable $v_1 : \gamma_1$:= e_1 ;
 ...
 variable $v_k : \gamma_k$:= e_k ;
begin
 stmt
end process ;

which has *stmt* as its sequential statement and assigns the initial value e_i to the state variable v_i, $i = 1, \ldots, k$.

In order to obtain the process' semantics it is clear that its syntactic constituents, the statement *stmt* and the variable declarations '$v_i : \gamma_i$:= e_i', cannot be arbitrary; only certain well-formed combinations will be sensible. To begin with all expressions, $e_1, \ldots, e_k$, must be of the right types, $\gamma_1, \ldots, \gamma_k$, and moreover they must be closed, *i.e.* they must not depend on variables or signals. This makes sure that we get well-defined initial values ($init_V \in [\![V]\!]$) for the internal variables of the process, *viz.*

$$init_V \quad = \quad \{v_1 = [\![e_1]\!], \ldots, v_k = [\![e_k]\!]\}.$$

Further, well-formedness requires that the sequential statement *stmt* by itself is well-formed, and that it has only the free variables $V = \{v_1 : \gamma_1, \ldots, v_k : \gamma_k\}$. All other identifiers occurring in *stmt*, then, are signals which will form the external interface of the process.

Statements. Let us suppose that I are all the input signals and O the output signals occurring in *stmt*. Then, the semantics of *stmt* as defined in the previous section specifies a partial function

$$[\![stmt]\!](f \in [\![V]\!] \times [\![I]\!] \rightarrow ([\![V]\!] \times Tr[\![O]\!])_\perp).$$

We can now lift $[\![stmt]\!]$ to a stream-processing function $g \in [\![I]\!]^\omega \rightarrow Tr[\![O]\!]^\omega$ using the initial state of variables $init_V$:

$$\begin{array}{l} [\![stmt]\!]\,(g \in [\![I]\!]^\omega \rightarrow Tr[\![O]\!]^\omega) \\ \qquad \equiv \quad g(in) \;=\; h(init_V)(in) \\ \qquad\qquad \text{where } h(v)(i \,\&\, in) = tr \,\&\, h(v')(in) \\ \qquad\qquad\qquad \text{where } (v', tr) = f(v, i) \;\wedge\; [\![stmt]\!](f). \end{array}$$

There is a subtle point here regarding nonterminating statements. Suppose *stmt* has a semantics f, $[\![stmt]\!](f)$, that does not terminate for some signal input i, in initial state $init_V$. Then $f(init_V, i)$ is not defined, and as a consequence $h(init_V)(i \,\&\, in)$ is undefined too. (Remember that & is strict in its first argument.) But being a stream this means it is the empty stream, whence finally $f^\omega(i \,\&\, in) = \langle\rangle$. Extending this argument one finds that if an input stream forces *stmt* into an infinite loop at some point, then the output stream stops forever. This is a rather natural way of treating nontermination which falls out automatically from our definition.

Remark *The predicate* $[\![stmt]\!](g)$ *specifies a unique function* $g \in [\![I]\!]^\omega \rightarrow Tr[\![O]\!]^\omega$. *One can show that this function is continuous and weakly synchronous. Moreover, if stmt is always terminating then one has for all input streams in,* $\#g(in) = \#in$.

This is the mathematical property reflecting VHDL*'s fundamental abstraction from sequential computations: a statement taken as a process does not advance time, its execution is completely performed within a single δ step.*

Let us take the counter stage $cnt_1\,(i, o)$ of Figure 1 as an example. Its body *c-stmt* is the statement

$$\textbf{if}\ i = \text{`}0' \ \textbf{and}\ x = \text{`}1'\ \textbf{then}\ z\ \texttt{:=}\ \textbf{not}\ z\ \textbf{else null}\,;\ x\ \texttt{:=}\ i\,;\ o\ \texttt{<=}\ z\,;$$

which specifies a stream-processing function $[\![c\text{-}stmt]\!](g \in [\![I]\!]^\omega \rightarrow Tr[\![O]\!]^\omega)$, where $I = O = \{i : \text{BIT}\}$. The initial state is $init_V = \{x = 0, z = 0\}$. We identify $[\![I]\!]$ and $[\![O]\!]$ with $\mathbb{B}$, $Tr[\![O]\!]$ with $\mathbb{B}^*$, $[\![V]\!]$ with $\mathbb{B} \times \mathbb{B}$, and obtain the set of equations

$$\begin{aligned} &[\![c\text{-}stmt]\!](g \in [\![I]\!]^\omega \rightarrow Tr[\![O]\!]^\omega) \\ &\qquad \equiv \quad g(in) \ = \ h(0,0)(in) \\ &\qquad\qquad h(x,z)(i \,\&\, in) = tr \,\&\, h(x',z')(in) \\ &\qquad\qquad (x', z', tr) \ = \ f(x,z,i) \\ &\qquad\qquad f(x,z,i) \ = \ \mathit{if}\ i = 0 \wedge x = 1 \\ &\qquad\qquad\qquad\qquad \mathit{then}\ (i, not(z), [not(z)])\ \mathit{else}\ (i, z, [z]), \end{aligned}$$

in which the last equation is the expanded predicate $[\![c\text{-}stmt]\!](f)$ as evaluated in the previous section.

Drivers. In order to translate from the output transactions $Tr[\![O]\!]$ of the statement to driven values $[\![O]\!]$ for the output signals we need to formalize the concept of drivers. In δVHDL a transaction trace is simply a list of projected values for every output signal and of these values only the very first element (*viz.* the last value scheduled) will become the driving value in the next δ-step. Thus, the driver for a process with output signals O is nothing but a stream-processing function dr from $Tr[\![O]\!]^\omega$ to $[\![O]\!]^\omega$ which, in every computation step, updates the output signals by the first element in the corresponding transaction list, and takes the old output if the transaction list is empty. In the first computation step the old value is just the default value specified by an assignment $init_O : [\![O]\!]$ of initial values to output signals. The formal definition of dr is as follows:

$$\begin{aligned} &dr(in) = init_O \,\&\, h(init_O)(in) \\ &\quad \text{where } h(o)(tr \,\&\, in) = o' \,\&\, h(o')(in) \\ &\qquad\quad \text{where } \mathit{if}\ tr.z = [\,]\ \mathit{then}\ o'.z = o.z\ \mathit{else}\ o'.z = hd\,(tr.z). \end{aligned}$$

Note the genericity of our approach: If we had started with a more refined notion of transaction traces, say for physical time signal assignments, we could

plug in here the corresponding notion of driver incorporating preemption, inertial and transport delay. This genericity is due to the notion of transaction trace, $Tr[\![O]\!]$, which decouples the semantics of statements from that of processes.

Remark *The above equations specify a unique stream-processing function* $dr \in Tr[\![O]\!]^\omega \rightarrow [\![O]\!]^\omega$. *Furthermore, one can show that* dr *is continuous and weakly synchronous. In fact, one shows that for all inputs* tr *the output is always exactly one step ahead, i.e.* $\#tr + 1 = \#dr(tr)$, *which means that the drivers are state-holding components with a delay of one* δ *step. They play the rôle of registers in the simulation cycle.*

The driver for our counter stage $cnt_1\,(i, o)$ of Figure 1 is a stream-processing function $dr \in Tr[\![O]\!]^\omega \rightarrow [\![O]\!]^\omega$, where $O = \{o : \mathrm{BIT}\}$. It is convenient again to identify $[\![O]\!]$ with $\mathbb{B}$ and $Tr[\![O]\!]$ with $\mathbb{B}^*$. If we assume the initial value $init_O = 0$ for output o, then the specification of the driver dr comes down to the following set of equations:

$$\begin{aligned} dr(in) &= 0 \,\&\, h(0)(in) \\ h(o)(tr \,\&\, in) &= o' \,\&\, h(o')(in) \\ o' &= \mathit{if}\ tr = [\,]\ \mathit{then}\ o\ \mathit{else}\ hd(tr). \end{aligned}$$

Process. We can now assemble the semantics of our generic process $proc$ from the beginning of this section. Again, the Focus definition is presented in terms of a logical predicate $[\![proc]\!](f)$ that specifies a stream-processing function $f \in [\![I]\!]^\omega \rightarrow [\![O]\!]^\omega$:

$$[\![proc]\!]\,(f \in [\![I]\!]^\omega \rightarrow [\![O]\!]^\omega) \quad \equiv \quad f = g\ ;\ dr \quad \text{where } [\![stmt]\!](g).$$

Here, $g; dr$ denotes the ordinary composition of function g followed by dr. With this definition the full counter stage $cnt_1\,(i, o)$ of Figure 1 results in the following equational specification:

$$
\begin{aligned}
&[\![cnt_1\,(i,o)]\!](f \in \mathbb{B}^\omega \rightarrow \mathbb{B}^\omega) \;\equiv \\
&\qquad f(in) = dr(g(in)) \\
&\qquad dr(in) = 0 \,\&\, h(0)(in) \\
&\qquad h(o)(tr \,\&\, in) = o' \,\&\, h(o')(in) \\
&\qquad o' = \mathit{if}\ tr = [\,]\ \mathit{then}\ o\ \mathit{else}\ hd(tr) \\
&\qquad g(in) \;=\; k(0,0)(in) \\
&\qquad k(x,z)(i \,\&\, in) = tr \,\&\, k(x',z')(in) \\
&\qquad (x',z',tr) \;=\; l(x,z,i) \\
&\qquad l(x,z,i) \;=\; \mathit{if}\ i = 0 \wedge x = 1\ \mathit{then}\ (i, not(z), [not(z)])\ \mathit{else}\ (i, z, [z]).
\end{aligned}
$$

The first equation comes from the specification of a process, equations 2–4 specify the driver, 5–7 the statement as a stream-processing function, and the last equation is the semantics of the statement itself. The system can be reduced immediately by performing the obvious substitutions and by packing together h and k into one function $H(x,z)(in) = h(z)(k(x,z)(in))$ so that we obtain the equivalent system

$$
\begin{aligned}
[\![cnt_1\,(i,o)]\!](f) \;\;\equiv\;\; & f(in) = 0 \,\&\, H(0,0)(in) \\
& H(x,z)(i \,\&\, in) = o \,\&\, H(i,o)(in) \\
& o = \mathit{if}\ i = 0 \wedge x = 1\ \mathit{then}\ not(z)\ \mathit{else}\ z,
\end{aligned}
$$

which is precisely what we expect. The function $H(x,z)$ represents the overall behaviour of the counter stage with local variables x, z. The first variable x stores the last input read and the second z the last output written. A falling edge of the input is detected by $i = 0 \wedge x = 1$. If this condition fails, the next output is the previous one $o = z$, otherwise it is inverted, $o = not(z)$.

Design. In the last step towards a semantical model for δ-VHDL we bring together processes with resolution to obtain the semantics of the **design** construct. To this end let $proc_0, \ldots, proc_n$ be a list of well-formed process statements so that, for $i = 0, \ldots, n$, $proc_i$ has the input signals I_i and the output signals O_i. By the definitions given above we have associated with each process $proc_i$ a predicate $[\![proc_i]\!](f \in [\![I_i]\!]^\omega \rightarrow [\![O_i]\!]^\omega)$ describing a stream processing function f from input stream $[\![I_i]\!]^\omega$ to output stream $[\![O_i]\!]^\omega$. Suppose, now, we wished to execute all the processes concurrently and define the dynamic semantics of the composite design as follows:

$$
\mathit{design} \;\;:\;\; \textbf{design begin}\ proc_0\ proc_1\ \cdots\ proc_n\ \textbf{end design}\,;
$$

Then, in general, two complications arise. Since we do not impose any restriction on the sets of inputs and outputs the *design* may contain feedback loops and signals may be written by more than one process. Both situations, which of course are essential for the expressive power of VHDL, jeopardize a purely functional approach to the description of the system's behaviour.

In VHDL the second complication is dealt with by distinguishing between *driving* and *effective* signal value, so that a signal can have more than one driving but only one effective value. The driving values are supplied by the drivers associated with the processes writing the signal. The designer is required to specify a *resolution* function to resolve the values supplied by multiple drivers to a single effective value. The typical examples for resolution functions are wired-**AND** and wired-**OR** but much more sophisticated applications are possible. In our formalism a resolution function for a signal of type α comes down to a function

$$res \in [\![\alpha]\!] \times \cdots \times [\![\alpha]\!] \rightarrow [\![\alpha]\!]$$

mapping a tuple $(a_0, \ldots, a_k)$ of values of type α to a single value $res(a_1, \ldots, a_k)$ of type α. For the wired-**OR**, for instance, we might have $\alpha = \mathrm{BIT}$ and

$$wired\text{-}or(a_0, \ldots, a_k) \quad = \quad a_0 \vee \cdots \vee a_k,$$

where $\vee$ is the OR function on bits. Returning to our *design* we will assume that every output signal $s \in O = \bigcup_i O_i$ has an associated (family of) resolution function res_s, which can take an arbitrary number of inputs and, of course, (each one) is user-defined. We assume that the resolution of a single signal amounts to the identity function, *i.e.* $res_s(a) = a$. These individual resolution functions can be put together to form a global resolution process for all output streams (this resolution process is indicated by the resolution box in Figure 6):

$$res \quad \in \quad [\![O_0]\!]^\omega \times [\![O_1]\!]^\omega \times \cdots \times [\![O_n]\!]^\omega \ \rightarrow \ [\![O]\!]^\omega$$

in the following way:

$$\begin{array}{l} res(o_0 \,\&\, os_0, \ldots, o_n \,\&\, os_n) \ = \ o \,\&\, res(os_0, \ldots, os_n) \\ \quad \text{where } o.z = res_z\,(o_{z_0}.z, \ldots, o_{z_k}.z) \\ \quad \text{and } \{o_{z_0}, \ldots, o_{z_k}\} = \{o_i \mid i \leq n \wedge z \in O_i\}. \end{array}$$

Note that all res_z are defined individually and in the special case where there is no write conflict, namely where the sets of output signals are disjoint the resolution function trivializes to the *zip* function. For example, suppose $O_0 = \{o_0 : \mathrm{BIT}\}$, $O_1 = \{o_1 : \mathrm{BIT}\}$, and $O_2 = \{o_2 : \mathrm{BIT}\}$ where $o_i \neq o_j, i \neq j$. Then,

we may identify $[\![O_i]\!]$ with $\mathbb{B}$, $[\![O]\!]$ with $\mathbb{B} \times \mathbb{B} \times \mathbb{B}$, and the resolution function is nothing but the zip function

$$zip \in \mathbb{B}^\omega \times \mathbb{B}^\omega \times \mathbb{B}^\omega \rightarrow (\mathbb{B} \times \mathbb{B} \times \mathbb{B})^\omega .$$

All it does is zipping together three streams of 1-element records into a single stream of 3-element records. A subtle point to remember is that this function is strict in the sense that it requires all three input streams to deliver a well-defined record in order to produce one output record. In other words, the resolution function waits for all processes to terminate before it starts the next δ step. This is a property of the VHDL simulation model, though one might also take a slightly more relaxed view, as done in [108], where subprocesses may loop indefinitely without stopping other processes not connected to them.

Remark *The resolution process res is uniquely defined by the above equational specification. It is further continuous and weakly synchronous.*

Multiply driven signals are fine through resolution, but what about feedback loops? An important advantage of Focus as our formal framework is that it offers two essentially equivalent ways of accommodating feedback: functionally via the fixpoint operator and logically via existential quantification [28]. Here we choose the latter method since we are more interested in the specification rather than in the execution aspect stressed by other stream-based approaches such as [108] or the semantics of Breuer *et al.* in Chapter 2.

Now, let us give the semantics of the parallel composition *design* of the processes $proc_0, \ldots, proc_n$, where each process $proc_i$ denotes a stream-processing function $f_i \in [\![I_i]\!]^\omega \rightarrow [\![O_i]\!]^\omega$ such that $[\![proc_i]\!](f_i)$, $i = 0, \ldots, n$. Let I_p be the set of primary inputs, *i.e.* those inputs which are not output of any subprocess; formally, $I_p = \{s \mid s \in I \wedge s \notin O\}$, where $I = \bigcup_i I_i$ and $O = \bigcup_i O_i$ are the set of input and output signals, respectively. The functional semantics of *design*, then, is a stream-processing function $g \in [\![I_p]\!]^\omega \rightarrow [\![O]\!]^\omega$ from the primary inputs to all outputs. This function g is specified by a predicate $[\![design]\!](g)$ as follows:

$$\begin{aligned} [\![design]\!](g) \quad \equiv \quad & g(is) = os \ \wedge \ is = sig\,|_{I_p} \ \wedge \ os = sig\,|_O \ \wedge && (1) \\ & f_0(sig\,|_{I_0}) = os_0 \ \wedge \ \cdots \ \wedge \ f_n(sig\,|_{I_n}) = os_n \ \wedge && (2) \\ & os = res(os_0, \ldots, os_n). && (3) \end{aligned}$$

The definition is a system of mutually recursive equations linked together by the variable $sig \in [\![I \cup O]\!]^\omega$ representing the observable stream of effective values on all signals in the parallel execution of all processes. We write $sig\,|_X \in [\![X]\!]^\omega$

to denote the restriction of the global signal stream *sig* to a subset of signals $X \subseteq I \cup O$; it can be obtained by appropriate *zip* and *unzip* operations. Variable $is \in [\![I_p]\!]^\omega$ refers to the stream of effective values on the primary inputs and $os \in [\![O]\!]^\omega$ to the stream of effective values on all outputs. The variable os_i refers to the stream of driving output values of process i.

With this interpretation of the variables in mind the equational specification can now be understood as follows: The first line (1) says that the global signal stream *sig* can be partitioned into the primary inputs *is* and the outputs *os*, and further that *os* is obtained by applying g to *is*. Line (2) is the main part of the specification. It superimposes the input-output behaviour of all processes as local constraints $f_i(sig\,|_{I_i}) = os_i$ on their respective input and output signals. The processes' output signals os_i carry driving values, from which in line (3) the effective output stream *os* is computed via the resolution function.

Note, the connection between the processes' inputs and outputs is achieved implicitly by the identity of signal names. Also note the implicit existential quantification over the internal output streams $os_0, \ldots, os_n$ which essentially achieves the hiding of internal computations via feedback.

Remark *It can be shown that the above system of equations specifies a unique stream processing function* $g \in [\![I_p]\!]^\omega \rightarrow [\![O]\!]^\omega$. *From the fixpoint theory for complete partial orders it follows that it has a unique minimal solution. In our particular case this minimal solution is in fact the only one. This results from the drivers in the feedback loops inserting a unique initial value into the computation and the fact that all functions are weakly synchronous.*

Remark *It is obvious that the definition of a design's semantics is compositional, i.e. whenever we replace one of the subprocesses* $proc_i$ *by an equivalent one* $proc'_i$, $[\![proc_i]\!] = [\![proc'_i]\!]$, *then the semantics of the modified design' is equivalent to the original one,* $[\![design]\!] = [\![design]\!]$.

Remark *For a compositional semantics of process systems it is crucial that we use the 'design' construct rather than a binary parallel operator as e.g. in [119]. If we replaced* **design begin** $proc_0$; $proc_1$; $proc_2$ **end design** *by* $proc_0 \parallel proc_1 \parallel proc_2$ *then not only would the* $proc_i$ *be proper subprograms for which a semantics is declared but also* $proc_0 \parallel proc_1$. *But then we must be entitled to replace this subprogram by a semantically equivalent one, say* $proc_0' \parallel proc_1$ *without changing the overall behaviour. Unfortunately, this is not the case, in general, due to the resolution functions. Of course, when resolution is not considered as in [119, 21, 108, 98] this problem is evaded.*

In order to make our semantics of designs concrete, finally, let us evaluate the 3-bit asynchronous counter cnt_3 of Figure 1. Assume that $f_0, f_1, f_2 \in \mathbb{B}^\omega \to \mathbb{B}^\omega$ are the stream-processing functions assigned to the three counter stages, *i.e.* $[\![cnt_1]\!](f_i)$, $i = 0, 1, 2$. Then the above specification yields

$$
\begin{aligned}
&[\![cnt_3]\!](g \in \mathbb{B}^\omega \to (\mathbb{B} \times \mathbb{B} \times \mathbb{B})^\omega) \equiv \\
&\qquad g(i) = zip(o_0, o_1, o_2) \ \wedge \ f_0(i) = os_0 \ \wedge \ f_1(o_0) = os_1 \ \wedge \ f_2(o_1) = os_2 \ \wedge \\
&\qquad zip(o_0, o_1, o_2) = zip(os_0, os_1, os_2).
\end{aligned}
$$

This is an equational specification of the counter's behaviour over the signals i, o_0, o_1, o_2. The extra signals os_0, os_1, os_2 are internal signals of the simulation cycle representing the driving output signals of the counter stages in each δ step. We know that for every input i the equation system defines a unique solution $g(i)$, which thus can be analyzed by equational reasoning. Knowing that this solution is minimal opens up proof strategies based on mathematical induction. A natural property one can verify for the counter is that if the input signal switches from 1 to 0 and is kept stable for at least 3δ steps the output vector (o_0, o_1, o_2) counts up by one modulo 8. The formal verification of this statement is beyond the scope of this paper. The interested reader is referred to [28, 27] for verification examples and a detailed discussion of some of the available proof techniques.

Exercise *Work out the semantics for the synchronous counter of Figure 3 in Section 2.*

5 CONCLUSION

In this tutorial a functional semantics for δVHDL, which is essentially VHDL restricted to zero-delay signal assignment, has been presented.

We start from a sea of VHDL processes, each containing sequential statements, and define their semantics directly in terms of ordinary mathematical objects such as functions and streams. This allows us to choose the semantical concepts most suited for each of the different syntactical levels of VHDL, namely expressions, statements, and processes. The possibility to choose the most adequate mathematical objects to present the semantics as well as the compositionality of our semantics can be seen as the main features of this work.

Focus is the formal framework on which our semantics is based. It provides for the necessary mathematical setting, a variety of well suited specification

techniques and a number of reasoning styles. A detailed explanation of Focus including the verification techniques available to formally reason about VHDL descriptions is beyond the scope of this tutorial. The interested reader is referred to the published material on Focus cited in the text.

Though we have only covered the delta-delay aspect of VHDL the semantics presented here can also be extended to handle physical timing as it is needed for transport and inertial delay signal assignments. This extension to timed signals affects the transaction traces and the drivers and proceeds along similar lines as outlined by Reetz and Kropf in Chapter 7 of this volume.

In some sense this work complements the approach of Breuer *et al.* presented in Chapter 2, which is also a functional semantics based on streams. They consider physical time execution and arbitrary wait statements, which we ignore, but omit delta delay, variable assignment and resolution, which are treated here.

APPENDIX A

SYNTAX OF δ-VHDL

The syntax of δVHDL is parametric in the syntax for identifiers, type names, and expressions. These syntactic classes can be instantiated along the lines set out in Section 4. A particular choice is given by the VHDL language standard [66].

$$
\begin{aligned}
\mathit{sequential_statement} ::=\ & \mathit{signal_assignment_statement} \\
\mid\ & \mathit{variable_assignment_statement} \\
\mid\ & \mathit{if_statement} \\
\mid\ & \mathit{loop_statement} \\
\mid\ & \mathit{null_statement} \\
\mid\ & \mathit{sequential_statement}\ \mathit{sequential_statement}
\end{aligned}
$$

$$\mathit{signal_assignment_statement} ::= \mathit{identifier} \Leftarrow \mathit{expression;}$$

variable_assignment_statement ::= *identifier := expression;*

if_statement ::= **if** *expression*
then *sequential_statement*
else *sequential_statement*
end if*;*

loop_statement ::= **while** *expression*
loop *sequential_statement*
end loop*;*

null_statement ::= **null***;*

process_statement ::= **process**
{ *variable_declaration* }
begin
sequential_statement
end process*;*

variable_declaration ::= **variable** *identifier_list : type_name := expression;*

design_statement ::= **design**
begin
{ *process_statement* }
end design*;*

identifier_list ::= *identifier* { , *identifier* }