A Native Process Algebra for VHDL

Peter T. Breuer and Natividad Martinez Madrid

DIT, Escuela Técnica Superior de Ingenieros de Telecomunicación,
Universidad Politécnica de Madrid, Ciudad Universitaria, E-28040 Madrid, SPAIN.

Abstract

A "native" process algebra for the behavioural component of VHDL is set out without reference to another language or theory. The aim is to reduce the complexity of process algebra-based approaches to formal verification or synthesis of hardware designs in VHDL. Recent work on the formal semantics of VHDL has allowed translation to a foreign paradigm to be avoided.

The subset of the language covered excludes delta delayed signal assignments and compound signals but includes wait statements, positively delayed signal assignments and all procedural language constructs.

1 Introduction

The goal of this paper is to help reduce the complexity of process algebra-based approaches to formal verification in VHDL, and to suggest a new route to system-level synthesis. Approaches to hardware definition languages via process algebras have, in all but one of the instances of which we know, focused on expressing the language in a general setting, such as Milner's CCS [11, 12]. The intention is to harness the popular symbolic evaluation or theorem-proving workbenches, such as those based on LOTOS [10], to the task of verifying the hardware descriptions against a formal specification. But translation makes the problem computationally more complex because the atomic constructs of the description are necessarily re-expressed as compounds of generic primitives. The result is often a combinatorial explosion. The solution offered here is a process algebra in which the hardware description language primitives are the algebra primitives.

Generally speaking, because process algebras are equational theories, sophisticated rewrite engines can readily be adapted to feed a process algebra-based verification workbench. This strategy will win out given sufficient computing resources, but the complexity of the problem means that the engineer must take an active part in the proving process and direct the machine's attention to appropriate subtasks. The degree and nature of the intervention varies with the workbench. SEVERO [6] verifies process algebra descriptions automatically against temporal logic specifications under a model-checking engine, after translation to a symbolic binary decision diagram format (itself a representation that has greatly contributed to reducing complexity in recent years). This tool has not been coupled to a VHDL translator, but one or two teams are working in the area. Bayol et. al. [2], for example, present a translation of a subset of VHDL to a labelled transition system and report on semi-automated verification using the CCS-basedIVAL tool. These teams all report difficulties with complexity.

It may be simpler not to translate out of VHDL. Barringer et. al. had this idea [1] following their experiences with translation of ELLA to labelled transition system diagrams for fully automated model checking (they simplified the diagrams using behavioural equivalences in ELLA). We believe that this possibility has been overlooked in the case of VHDL, perhaps because confidence in its semantics has been lacking until now. The situation has been transformed in recent months by the efforts of several authors, including ourselves and our colleagues in Euroform (see [7] for a collection of stable points of view on VHDL semantics, and the special issue in which [5] appears for up to the minute positions). The present paper will set out a series of equational rules for the constructs of ordinary VHDL which are sufficient to obtain all the equalities that may ever be obtained, according to the semantic model of behavioural VHDL that has been reported in [4], for example. The rules will be introduced through a worked example; a VHDL code emulating an oscillator in parallel with a lagging follower.

The plan of our approach is (1) to extend VHDL code with the standard process algebra constructs in as minimal a way as possible, (2) to introduce into VHDL itself a new special construct which abbreviates a compound of existing VHDL statements, (3) to state
the algebraic laws governing interactions between the special construct and the existing VHDL constructs, (4) to observe that these laws suffice to reduce the subset corresponding to the original VHDL grammar to a normal form consisting of a set of traces of events, and therefore that (5) the laws presented are sufficient to decide equality or inequality between VHDL process behaviours over a given finite time interval (equality between behaviours over unbounded time is not always decidable for fundamental reasons).

The layout of this article is as follows. After the present introductory section, Section 2 goes on to explain the addition to the VHDL syntax required, and Section 3 sets out the process algebra laws, with a worked example. Section 4 offers a short summary.

1.1 VHDL

A structured VHDL architecture can be elaborated to a purely behavioural description and we will be interested here only in the behavioural component of VHDL. Figure 1 shows the portion of VHDL treated.

This portion includes positively delayed transport signal assignments and wait on statements. The semantics of inertial signal assignment is more complicated than transport assignment and we do not treat it here — though the treatment would be analogous, requiring only a different variant of the relevant reduction law. The major omission here is of delta delayed signal assignments. To represent these we would have to move to a doubly-linear model of time, in which we counted 0, 0+δ, 0+2δ, ..., 1, 1+δ, 1+2δ, ..., 2, 2+δ, 2+2δ, .... instead of plain 0, 1, 2; ...; one “delta” being used in each simulation cycle [3]. There is no intrinsic difficulty, but because we are presenting the algebra for the first time, we do not wish to complicate the explanation in this way.

VHDL allows several variants of the wait statement. Wait for time statements can, however, be constructed using a new channel A and a transport assignment to A after time followed by a wait on A. Wait until expression statements can be emulated with a while not expression loop containing a wait on the signals referenced in the expression. So we consider it justifiable to treat the wait on variant alone.

Note also that VHDL can always be reduced to a form in which no variables (as opposed to signals) appear. Variables which need not persist across time can be replaced with an appropriate expression (which may make calls on recursive external functions). For example, the code below uses the variable x:

\[ x := C + D - 1; \]
\[ E \leftarrow \text{transport} \; x+1 \; \text{after} \; 1ns; \]

and it can be transformed to the following code by unfolding \( x \) in the signal assignment:

\[ E \leftarrow \text{transport} \; C + D \; \text{after} \; 1ns; \]

Variables that persist across time can be emulated by signals; the process can signal to itself in the next quantum of time those values that it wishes to persist. For example, inserting a wait statement in the code above gives code in which \( x \) now persists across time:

\[ x := C + D - 1; \]
\[ \text{wait for} \; 1ns; \]
\[ E \leftarrow \text{transport} \; x+1 \; \text{after} \; 2ns; \]

but it can be made over into the following code in which the variable is volatible again:

\[ X \leftarrow \text{transport} \; C + D - 1 \; \text{after} \; 1ns; \]
\[ \text{wait for} \; 1ns; \]
\[ x = X; \]
\[ E \leftarrow \text{transport} \; x+1 \; \text{after} \; 2ns; \]

and now the variable can be eliminated. So we treat a subset of VHDL with signals but no variables.

The promise construct abbreviates a compound sequence of VHDL transport assignment statements, and the algebra of VHDL will be described through laws of behavioural interaction between promises and other VHDL statements. The syntax is as follows:

\[ \text{Promise} ::= [\text{Chan}!\text{Const},\ldots] \text{Chan=}\text{Const}... \]

We note the syntax and leave the semantics till later.

The VHDL example shown in Figure 2 will be used in this paper to introduce the algebra. It is an oscillator process a in parallel with a follower process b. The oscillator cycles every 2 units of time and the follower follows it with a 1 unit delay.

1.2 Process Algebra

A native process algebra for VHDL is a set of equations which relate behaviourally equivalent fragments of VHDL code. In order to express actual events, the raw language has to be embellished with the usual process algebra constructs for events, parallelism, choice, and so on. The grammar is shown below:

\[
\text{Algebra} ::= \text{VHDL} \\
\quad | \text{Event Algebra} \\
\quad | \text{Algebra} || \text{Algebra} \\
\quad | \text{chaos} \\
\quad | \text{Algebra} \mid \text{Algebra} \\
\quad | \text{step} \\
\quad | \text{Channel}!\text{Value} \;
\]

\[
\text{Event} ::= \text{Channel}!\text{Value} \;
\]

\[
\text{Event\ Algebra} ::= \text{Algebra} || \text{Event\ Algebra} \\
\quad | \text{step} \\
\quad | \text{Channel}!\text{Value} \\
\]

\[
\text{embedded\ VHDL\ code} \\
\quad | \text{event\ prefixed\ to\ code} \\
\quad | \text{synchronous\ parallelism} \\
\quad | \text{the\ chaotic\ process} \\
\quad | \text{nondeterministic\ choice} \\
\quad | \text{no\ choice\ at\ all} \\
\]

\[
\text{Value} ::= \text{Value} \;
\]

\[
\text{Value\ output\ on\ channel} \\
\quad | \text{value\ output\ of\ channel} \\
\]
The algebra will be based on the *traces model* of processes (without refusals) [9]. A process is described as the set of its possible traces, where a trace is a sequence of *events*, ordered as they would occur in real time. In deference to the VHDL picture of the world, traces which differ only by the ordering of events on *different* channels will not be distinguished. Thus C1 C2 D1 D2 and C1 D1 D2 C2 are the same traces. The idea is that all the channels are clocked synchronously at the rate specified by the VHDL minimum simulation interval, and an event – often a sustaining event – occurs on each channel at each of these clock ticks. We can know when in real time an event occurs by counting back to see how many events/dock ticks have earlier occurred on that channel in the trace.

Non-deterministic choice between processes is modelled by a process with the union set of traces, *stop* is the process with the empty set of traces, parallelism is modelled by taking the intersection of sets of traces and *chaos* is the process with all possible traces. Accordingly, the choice and parallelism operators interact as in a boolean algebra, with choice as the “addition” or “or” operator, *stop* the “zero” or “false” value, parallelism the “multiplication” or “and” operator, and *chaos* the “unit” or “true” value.

The process algebra concept of “prefixing by an event” is modelled concretely by prefixing a given event to every trace in a set of traces. This associative operation distributes over choice and parallelism.

It is convenient to visualize a set of traces (i.e., a behavioural model of a process) as a tree. The nodes of the tree are states of the process and the arcs are labelled with events in such a way that the set of paths through the tree is the set of traces. The action of prefixing by a trace in the model is to graft the tree onto the stem provided by the trace. We may draw a directed (cyclic) graph to abbreviate an (infinite) tree.

## 2 Promises, Promises

In this section we extend VHDL with the single construct that we call a *promise*. Interactions with it will capture the semantics of the rest of VHDL. The extension is *conservative* (the new syntax and the extra logic do not change the logic of the pure language). A promise is consistently seen here as equivalent to a proper compound of core VHDL transport signal assignments, plus a record of the current state. This is an example of a promise on a signal C, initially zero:

```
C!0!1!0!1!1!0!1!0!1!1!0!1
 0 1 2 3 4 5 6 7 8 9 10 11
10 11 12 13 14 15 16 17 18
```

---

**Figure 1:** The syntax of the subset of VHDL treated here. The special “promise” abbreviation is explained later, but it serves to initialize the schedule of events on given channels.

**Figure 2:** VHDL code for an oscillator a on signal C and a follower b outputting to signal D, with flow diagrams.
It promises to send S.O.S. out on signal C, in Morse code. The planned waveform looks like this:

![Waveform Diagram]

How does one think about constructing a promise? In this case we want to send out a short 1-valued pulse, padded by a zero on either side, and repeat two more times. Then three long 1-valued pulses, then three short. The initial zero state is recorded in the promise by the C ! 0. To put out the first short pulse, we promise a 1-value in 1 ns time: \[ C ! 1 \], followed by a padding zero in 2 ns time: \( C ! 0 \). The second short pulse is begun by a 1-value in 3 ns time: \( C ! 1 \), and so on.

Starting when the signal has the value zero, the promise is equivalent to the sequence of VHDL simple transport assignments which begins with the to-be-earliest assignment, and ends with the to-be-last.

\[
\begin{align*}
C &\Leftarrow \text{transport 1 after 1ns;} \\
\cdots \\
C &\Leftarrow \text{transport 0 after 21ns;}
\end{align*}
\]

The promise notation is really just a concise way of writing several simple transport assignments at once. The power of the notation comes from the way promises combine with wait statements to produce real events, and that will be considered in the next section. There is one part of the notation still to be introduced, however. If \( I \) is an input channel to a process, but not an output channel for it, then we write a promise for that process with the \( I \) as a subscript:

\[
[C ! 1 _{\text{\(I\)}}] \Rightarrow \text{transport 1 after 1ns;}
\]

indicating that an input value \( e \) for \( I \) is required in synchrony with every output on \( C \). The " = 0" indicates that the initial value on the channel is zero. A promise should be subscripted by all the inputs in the environment of the given process, but when the information is not required the subscript may be dropped.

3 The Process Algebra

This section sets out the rules governing interactions between all the VHDL constructs of interest. The rules are sound (see Appendix A).

3.1 Promises and Transport Assignments

Suppose that a transport assignment is scheduled for time 7, in the middle of a planned sequence stretching from time 4 to time 13. Then the promise followed by the transport assignment is equivalent to a single promise in which the sequence has been cut short at 6 and now terminates differently:

\[
[\text{H!0!1!0!1!1; H}\Leftarrow \text{transport 1 after 7ns;}}
\]

Abtracting to the general case, the following rule holds. If \( t_i < t \leq t_i + 1\):

\[
[C ! c_0 \ldots ! c_n]; C \Leftarrow \text{transport } c \text{ after } r \text{ns;}}
\]

I.e., the transport assignment pre-empts the parts of the promise that are later than the assignment, promising its own value instead. But if the transport assignment is strictly later than all parts of the promise, then it adds the extra value on the end of the promise. I.e., if \( t_m < t \), then:

\[
[C ! c_0 \ldots ! c_m]; C \Leftarrow \text{transport } c \text{ after } r \text{ns;}}
\]

Because transport assignments live inside processes, and processes initialize their signals to zero, as captured by a promise of the shape \([C ! 0;] \), so all transport assignments effectively follow a promise and the treatment above suffices to explain how they interact.

3.2 Promises as Process Headers

Processes carry a header not represented in the grammar of Figure 1. Each process declares the input and output signals for the process:

\[
\text{Pid : process Ins Outs begin VHDL end}
\]

but the encapsulation can be represented using a promise to initialize the output signals and declare the inputs. For example, the header of process \( B \) in Figure 2 is \( B : \text{process [C; in] [D; out] begin \ldots end} \) and it can be represented by a promise that sets \( D \) to zero and reports input \( C \) as being set to zero:

\[
[0 ! 0]_{D\Rightarrow e}; \ldots
\]

We choose not to represent process headers explicitly because that would introduce an extra layer into the algebra and provoke a duplication of the algebraic laws.
3.3 Waiting and Waiting

The interaction of a promise with a wait statement is what turns promises into reality. Let $t_k$ be the first promised time at which signal $C$ will change, say from initial value $c_0$ to $c_2 \neq c_0$. Then, the following promise by a wait statement causes the promised values to be emitted on the channel, in sequence, and the wait statement is discharged with a certain number of promises remaining to be fulfilled:

$$[C!c_0 \ldots !c_k \ldots !c_n]; \text{wait on } C$$

(3)

$$= C!c_0 \ldots C!c_k \ldots C!c_n$$

for $t_k$ times.

The $C!c$ represents a value $c$ output on channel $C$ during one unit of time.

This is enough information to generate a first example. Consider the VHDL oscillator process $a$ of Figure 2. It has one output channel, $C$, and no input channels. We can represent it as a tail recursion preceded by an initial setup of zero on channel $C$. That is, VHDL code preceded by a promise:

$$[C!0]; a$$

where $a = C \leftarrow \text{transport not } C \text{ after } \text{ins}; \text{wait on } C; a$

The equations developed so far then tell us that the following reductions are valid (the sequence is represented graphically in Figure 3):

$$[C!0]; a$$

$$= [C!0]; C \leftarrow \text{transport not } C \text{ after } \text{ins}; \text{wait on } C; a$$

$$= [C!1]; \text{wait on } C; a$$

$$= C0 \ [C!1]; a$$

The tree depicted in Figure 3 can be turned into a graph by the simple expedient of labelling each node with the code for its depending subtree, and identifying nodes with equal labels as they appear. This procedure can dramatically reduce the complexity of the graph passed to a model checker.

3.4 Input

Consider the second process $b$ in Figure 2. It outputs 1 ns later on channel $D$ whatever it has received on channel $C$, starting at the initialization time when all signals are zero. The beginning promise is:

$$[D!0]; e \leftarrow C; b$$

The initialization followed by the process is

$$[D!0]; e \leftarrow C; b$$

where $b = D \leftarrow \text{transport } C \text{ after } \text{ins}; \text{wait on } C; h$

We expand out the initial promise followed by an assignment as follows:

$$[D!0]; e \leftarrow C; b$$

$$= [D!0]; e \leftarrow C; D; a$$

Time has not moved on, so the initial conditions, in which $C$ and $D$ carry a zero, are maintained. But following this promise with a wait statement results in time moving on. What unfolds is a “choice” between two possibilities; a zero must appear on channel $C$ since it is the present state and there are no delta-delayed assignments to change that, but the promised next state of $C$ may be either a zero or a one. If it is a one, then the wait statement releases after the unit time delay which has just passed. If it is a zero, the wait continues to block. In the meantime, the promised values of $D$ “discharge” and generate real events.

$$[D!0]; e \leftarrow C; b$$

$$= C0 \ D0 \ [D!0]; e \leftarrow C$$

The alternate in the choice continues to offer up possibilities, according to what transpires on channel $C$:

$$= C0 \ D0 \ [D!0]; e \leftarrow C$$

$$| \ C0 \ D0 \ C0 \ D0 \ [D!0]; e \leftarrow C$$

$$| \ C0 \ D0 \ C0 \ D0 \ C0 \ D0 \ [D!0]; e \leftarrow C; b$$

and so on.

We will be running this follower in parallel with the oscillator process $a$, and that produces a zero, one, zero, one, ... sequence on channel $C$. This trace is
incompatible with all but the first alternates above, and so we can ignore the others. They represent long periods with no change in C.

\[
[D!0]_{0}^\infty; \text{wait on } C
= C_0 \diamond_0 [D!0]_{0}^\infty | \ldots
\]

Follow this by a recursion into the follower b again:

\[
C_0 \diamond_0 [D!0]_{0}^\infty; \text{wait on } C, b | \ldots
= C_0 \diamond_0 C_1 \diamond_0 [D!0]_{0}^\infty; b
| C_0 \diamond_0 C_1 \diamond_0 C_0 \diamond_0 C_1 \diamond_0 C_0 \diamond_0 [D!0]_{0}^\infty; b | \ldots
\]

The second, third, etc. alternates will lead to repeated values of one on channel C, which is incompatible with the oscillator supplying the input. So these traces can also be dropped from consideration.

Another cycle of expansion now yields the result:

\[
[D!0]_{0}^\infty; b
= C_0 \diamond_0 C_1 \diamond_0 [D!0]_{0}^\infty; b | \ldots
= C_0 \diamond_0 C_1 \diamond_0 C_0 \diamond_0 C_1 \diamond_0 C_0 \diamond_0 C_0 \diamond_0 [D!0]_{0}^\infty; b | \ldots
\]

and it can be seen that the cycle repeats with D following C after the initial conditions stabilize. Starting with a value of one on channel D would have induced stabilization immediately. The full transition system for the follower, including the paths that have not been traced here (because it has been known that the oscillator will not allow them) is shown at left in Figure 4. The tree of traces has been reduced to a graph by equating syntactically identical subtrees. Further algebraic reductions give the graph at right. Syntactically distinct subprograms with equal behaviours have been equated.

3.5 Parallelism

We want to place the oscillator a in parallel with the follower b. The construct is written:

\[
a \parallel b
\]

in the algebra, and the standard process algebra laws apply. For example:

\[
(\varepsilon a) \parallel (\varepsilon b) = \varepsilon (a \parallel b)
\]

Only traces which are common to both processes survive in the conjoined pair.

When performing calculations on a set of processes running in parallel, either all the traces of each may be calculated separately, and then some eliminated and then some eliminated after the conjunction, as shown in Figure 5, or the elimination may be carried out during the subprocess calculations, as in the foregoing section. The latter is much the more efficient technique.

We can also write “compound” promises using parallel composition. A promise to write on two channels C and D initialized to 0 and 2 respectively, for example, can be expressed as two parallel promises, each listening to the other:

\[
[C!0]_{0}^\infty || [D!1]_{0}^\infty
\]

Other environmental inputs, if any, are written as extra inputs to both promises.

Note that the development given in this paper constitutes a formal proof that the oscillator and follower pair behave as may be expected.

4 Summary and Future Directions

A process algebra has been set out for the subset of VHDL excluding delta delayed signal assignments and compound signal resolution functions. The algebra is suitable for reasoning about and reducing the complexity of VHDL code before it is further analysed.

The equational treatment here is complete in that it correctly detects differences or proves equivalences in process behaviours over a bounded time interval. The
natural process algebra semantics also gives rise to a labelled transition system semantics for VHDL code.

We have found recently that it is possible and convenient to synthesise VHDL code from labelled transition diagrams using the algebra set out here. All diagrams in which the exit arcs from each node carry compatible events are synthesisable (see Appendix B). The diagrams are an easily understood design medium and synthesis proceeds by replacing each subdiagram in turn with code of algebraically equivalent trace semantics, using a library of pattern matches. The resulting code is not more complex than the design, in contrast to code produced by compilation of LOTOS, for example [8].

References


A Correctness

What assurance do we have that the rules here are correct? Only that they derive from the (equivalent) model-based and axiomatic semantics set out in [4] and related references. The model-based semantics, for example, describes VHDL statements $R$ as a relation between two history+schedule pairs. A history can be understood as the complete sequence of events, one per clock tick, on each channel since the start of a VHDL simulation, and a schedule can be understood as the current promise (in the VHDL terminology, a set of drivers). Time moves forward through the extension of the history part and the alteration and contraction of the promise part. If the model semantics says that $R$ relates a history of events $h$ and a promised schedule $s$ to a new, extended, history $h'$ and new promised schedule $s'$:

$$(h, s) \mathcal{R} (h', s')$$

then we set up rules here of the form

$h; s; R; S = h'; s'; S$

and are assured that they conform to the published model. Since $h$ appears on both sides (and covers the zero case), it can be removed:

$s; R; S = \Delta; s'; S$

but very often the compound $s'; S$ on the right hand side has been replaced here by a combination $s''; S'$ that has the same model-theoretic semantics, but a more concise form.

B Classification

Theorem 1 The class of state transition diagrams that can be refined to VHDL code through the algebra given here is precisely the class in which every exit from a given state carries compatible event patterns.

The proof shows that every transition diagram constructed via the algebraic laws given here either introduces or preserves this form. Wait statements and signal assignments introduce new diagrams. All other VHDL statement forms consolidate given diagrams into new compounds.

Conversely, a state from which every exit carries the same event pattern can be represented via a wait on statement if it has a self-loop, and by a wait for if not, according to the algebraic laws.