Technical Report

UCAM-CL-TR-249
ISSN 1476-2986

Number 249

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

A formalisation of the VHDL simulation cycle

John P. Van Tassel

March 1992

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

© 1992 John P. Van Tassel

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

A Formalisation of the VHDL Simulation Cycle

John P. Van Tassel[†]

University of Cambridge Computer Laboratory
New Museums Site, Pembroke Street
Cambridge, CB2 3QG, England.

Abstract: *The VHSIC Hardware Description Language (VHDL) has been gaining wide acceptance as a unifying HDL. It is, however, still a language in which the only way of validating a design is by careful simulation. With the aim of better understanding VHDL's particular simulation process and eventually reasoning about it, we have developed a formalisation of VHDL's simulation cycle for a subset of the language. It has also been possible to embed our semantics in the Cambridge Higher-Order Logic (HOL) system and derive interesting properties about specific VHDL programs.*

[†]Research supported under grant number AFOSR-91-0246 from the United States Air Force Office of Scientific Research

Contents

1 Introduction **5**

2 Overview of VHDL **5**

3 Related Work **6**

4 Philosophy **7**

5 Intuition behind the Semantics **7**

6 The Femto-VHDL Subset **10**

7 The Semantic Framework **12**

8 The Semantics of Femto-VHDL **13**
- 8.1 Types and Functions 13
- 8.2 Rules for Boolean Expressions 15
- 8.3 Rules for Sequential Statements 16
- 8.4 Rules for Concurrent Statements 17

9 Femto-VHDL in HOL **18**

10 Conclusions and Future Work **22**

11 Acknowledgements **23**

References **23**

1 Introduction

When the VHSIC Hardware Description Language (VHDL) burst onto the design scene, it was described as the solution to many existing problems. But, when the language was designed very little thought was given to its formal semantics, much less to using such a semantics to reason about individual VHDL programs. The result has been a recent upsurge in research efforts addressing precisely these problems. The following is a discussion of our attempt to formalise the simulation cycle and timing model of VHDL.

Our exposition will begin with an overview of VHDL's history and uses and will be followed by a *précis* of other work on the application of formal methods to VHDL. We shall then explain the driving philosophy behind our work. The semantics we have developed is then presented, and we conclude with the description of some experimental results in the application of our semantics to specific VHDL programs.

2 Overview of VHDL

VHDL is a hardware description language (HDL) currently in use by a reasonably large part of the design community. It is an event-driven simulation language whose semantics is defined, at least informally, by the way in which the various language constructs are evaluated by the simulation model [6].

To better understand VHDL and its place in current design practice, we should consider the evolution of the language:

pre-1980's Chaos

> This is a time period characterised by the existence of many proprietary hardware description languages and simulators. Furthermore, no one HDL is used throughout the entire design process. In many cases, ad-hoc mechanisms are developed to translate between the tools and languages in use at different stages of the design. This lead to increased confusion and provided a further avenue for the introduction of bugs.

mid-1980's VHSIC programme

> The United States Air Force, a large consumer of custom electronics obtained from a variety of vendors, is faced with the need to rationalise its procurement process. The primary concern is that all designs

submitted as a part of the bidding process are written in the same non-proprietary HDLand meet certain design and documentation standards. VHDL emerges as the language.

late-1980's IEEE VHDL (STD 1076)

As a part of the VHSIC programme, large vendors such as IBM and Texas Instruments take an interest in VHDL. That interest, coupled with the growing need for an industry-standard HDL, causes VHDL to go through the IEEE standardisation process and emerge as the language in use today.

Since its introduction, VHDL's use has spread to many areas of digital design. While most of these applications are in the areas of design, there has also been recent interest in synthesis directly from VHDL. The language in its current state is not generally amenable to this task, but it is anticipated that modifications to the standard (VHDL'92) will overcome many of the problems.

3 Related Work

Past efforts at applying formal methods to VHDL programming were carried out in the late 1980's. These were intended to give additional simulation-time assurance that particular VHDL programs were performing as anticipated. One of these investigations led to the development of the VHDL Annotation Language (VAL) at Stanford [2], which allowed the user to decorate a program with specifications about its operation. Another of the early projects, conducted by the present author, attempted automatically generate VHDL assertions characterising the behaviour of VHDL programs using syntactic analysis [11, 12].

More recent research has been directed at the verification of VHDL programs. To accomplish their goals, all these projects have necessarily had to build up a notion of VHDL's semantics. Temporal logic is currently in use at the Aerospace Corporation as the framework in which their current research is being conducted [5]. Other useful results are emerging from work being conducted at IMAG in Grenoble on the application of more functional methods to the problem [10].

4 Philosophy

Our formalisation of VHDL takes a rather pragmatic view. The semantics of the language, as given informally in [6], emphasizes the way in which the simulation behaves in the presence of events and transactions, and the way in which language constructs are executed by that simulation model. Our plan, therefore, is to codify these same principles more formally. The result not only reflects the actual VHDL simulation cycle, but corresponds intuitively to the way in which VHDL users think about the language. For purposes of our research, it has been necessary to utilise only a subset of full VHDL. This subset is extensible within the context of our semantics by simply formalising the execution of the desired extensions by the simulation model of VHDL. It is, however, sufficient to demonstrate the most salient features of the simulation model.

5 Intuition behind the Semantics

Our intention is to formalise as closely as possible the simulation model of [6]. To make this formalisation more understandable, we now explain what actually takes place during the course of a VHDL simulation. We begin by introducing a few definitions.

Event: A change in the value of a signal.

Transaction: The value that a signal should take on at a particular time in the future. A transaction may (or may not) be converted into an event at that time.

Point of computation: The point at which a particular collection of transactions is processed.

Process: A VHDL concurrent statement equipped with a set of signal names, or sensitivity list, guarding its activation.

Simulation cycle: The evaluation of all the processes in a program at a particular point of computation.

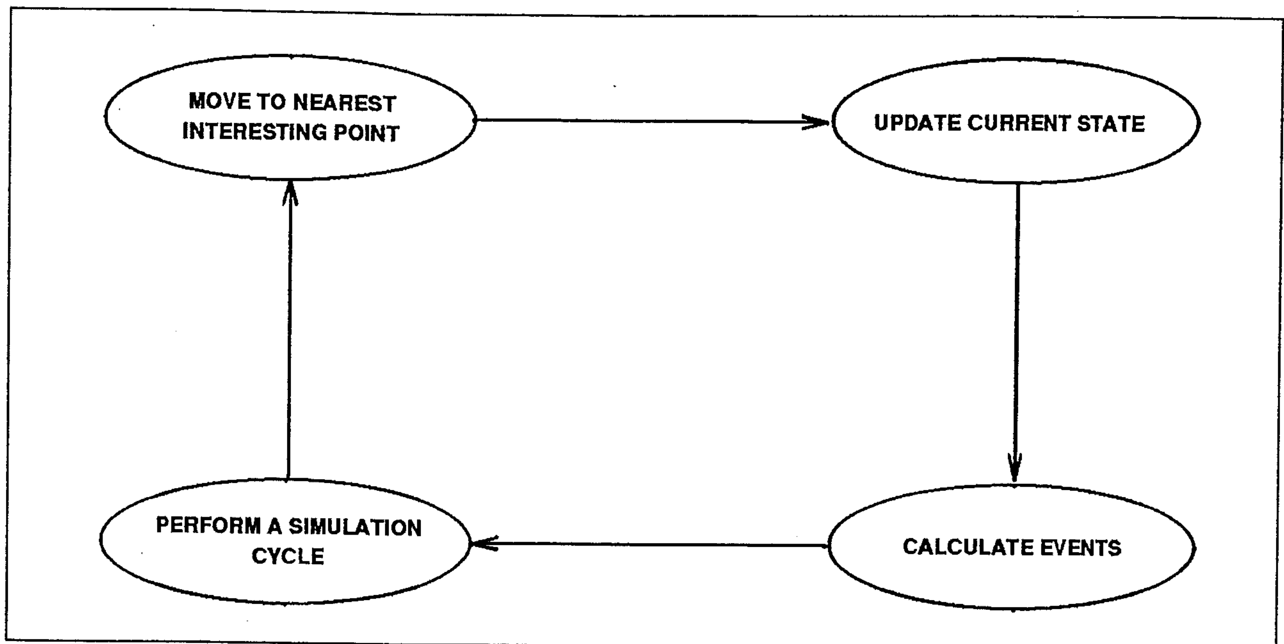

Figure 1: Progression of a simulation

The top-level simulation model of VHDL may be viewed as a four step sequence, and is illustrated in Figure 1. We begin by moving forward to the nearest interesting point of computation (i.e. one where there are transactions to process), and set the current time to be the physical time unit associated with that point of computation. The state of the signal values is then updated by those that they are supposed to take on during the present point of computation. We then move on to determine the signals for which this update represents a change in value, or event. A simulation cycle is then performed based on the our new state of the world. The cycle is then repeated until there are no more transactions to process. We also present the following pseudo-code as a more concise description of the simulation loop:

```
while transactions remain to be processed
    go to nearest point of computation with transactions to process
    update the state from the current transactions
    determine which updates represent events
    perform a simulation cycle based on new state and events
end while
```

The state referred to should be understood to contain the values of the individual signals.

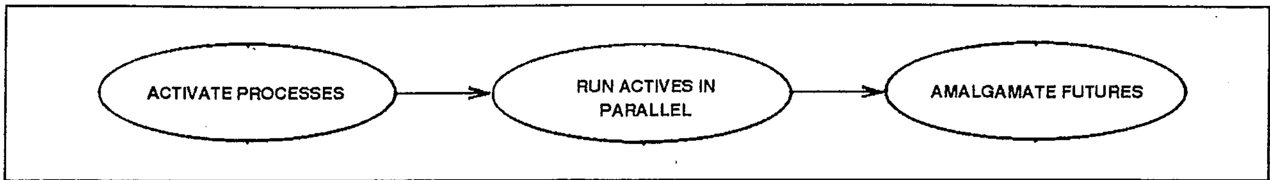

Figure 2: Progression of a simulation cycle

We emphasize the use of the phrase "point of computation" above. While one could often equate this concept with that of a time slice, a point of computation is more than that within the context of VHDL's particular simulation model. One could, for instance, say that the points of computation P_1 through P_n represent the time units 1 through n. Alternatively, they could represent 1 through n δ-delays between two major time units.

The concept of δ-delay is the way in which VHDL deals with 0-delay signal assignments. During each iteration through the simulation loop we are using a static state of the world and are scheduling transactions to be performed at some future time point. This scheduling applies to any transaction, whether it is to occur at a delay offset zero units from now, or at some larger one. Each time we go around the simulation loop (Figure 1) in zero time is called a δ-delay. An individual δ does *not*, therefore, represent a quantifiable unit of time. Rather it is simply a simulator artifact.

As mentioned earlier, a component of the simulation loop is the performance of a "simulation cycle". The unfolding of such a cycle as shown in Figure 2 begins with the activation of those processes that are sensitive to any of the current events. We then proceed to run these active processes in parallel. During their execution, each process schedules transactions to be processed in the future. When all the processes have terminated, we gather those separate futures into an amalgamated view of future behaviour. The above steps are again more concisely stated as:

```
determine which processes are active based on current events
run the active processes in parallel
join all activated process' scheduled transactions into a collective whole
```

6 The Femto-VHDL Subset

The VHDL subset that is used in defining our semantics is rather small (hence the name "Femto-VHDL"), but it contains enough constructs of the language to exercise the important features of the simulation model. We support only one kind of VHDL concurrent statement, namely `PROCESS` statements with explicit sensitivity lists. The sequential statements addressed are the `IF-THEN-ELSE` conditional and the transport delay signal assignment. We assume that:

- Signals may only be Boolean-valued. This is for simplicity, and does not impact our exposition of the simulation model in any way.
- Resolved signals are not allowed. The implication is that constructs such as wired-and's and wired-or's are not permitted.

Given the above listing of supported statements and assumptions, one can ask what actually constitutes a Femto-VHDL program. At the outermost level, such a program is made up of one or more concurrent process statements, each of which comprises some sequential statements. This structuring of constructs is described by to the following abstract syntax for Femto-VHDL:

```
bexp ::= sig | not bexp | bexp and bexp | bexp or bexp | bexp nand bexp |
         bexp nor bexp | bexp xor bexp | bexp xnor bexp | true | false
  ss ::= ss ; ss | bexp => ss | ss | sig := (bexp,dly)
  cs ::= cs || cs | sl : ss
```

bexp ranges over Femto-VHDL Boolean expressions, which are made up of single signal names, or of compound expressions using the various VHDL Boolean operators. Sequential statements (*ss*) may be either two statements in sequence (;), a conditional statement guarded by some Boolean expression, or a signal assignment statement that assigns a Boolean value to some signal after some delay expressed in terms of the natural numbers. Finally, *cs* shows the structure of concurrent statements. A concurrent statement may be either two such statements in parallel (||), or it may be a single process composed of a sensitivity list (*sl*) and sequential statements (*ss*).

We now present a pair of examples that demonstrate the abstract syntax of Femto-VHDL and its relationship to real VHDL code. The translation between the abstract syntax and actual VHDL is meant to be obvious and is raised here to avoid confusion when results are presented later. The first example,

```
{C} : C := (NOT C,dly)

PROCESS (C)
BEGIN
 C <= TRANSPORT (NOT C) AFTER dly;
END PROCESS;
```

Figure 3: A uniprocess oscillator

```
({H} : H := (NOT H,dly)) || ({H} : C1 := (H,0 NS))

PROCESS (H)
BEGIN
 H <= TRANSPORT (NOT H) AFTER dly;
END PROCESS;

PROCESS (H)
BEGIN
 C1 <= TRANSPORT H AFTER 0 NS;
END PROCESS;
```

Figure 4: A dual process oscillator

shown in Figure 3, is a uniprocess program that describes an oscillator with a half-period of `dly`. The second (Figure 4) is a simple extension of the previous device which adds a second process representing a 0-delay buffer. While this second example might seem contrived, it will be used to demonstrate certain properties of the formal semantics later. Both examples are drawn from [1].

7 The Semantic Framework

We now give a brief overview of the formalism to be used in defining the the semantics of Femto-VHDL. In our semantics we will make use of a collection of labelled transition relations in the style of [9]. Each of these relations is defined inductively as the least relation closed under a set of rules describing the way in which different classes of Femto-VHDL constructs are evaluated by the simulation model.

To better understand the concept of inductively defined relations, let us examine an example taken from [4] using the even natural numbers. We could define the properties necessary to describe even naturals to be:

1. 0 is an even number.
2. if n is even, then $n + 2$ is also even.

These statements could be rephrased by the following set of deduction rules holding of a predicate even. Such rules are generally made up of *hypotheses* above the line and the *conclusion* below it. As a point of terminology, rule 1 is an *axiom* as it has no hypotheses at all.

$$(1) \quad \frac{}{\mathsf{even}\ 0} \qquad\qquad (2) \quad \frac{\mathsf{even}\ n}{\mathsf{even}\ n+2}$$

Since the rules correspond directly to the minimal properties required of the even numbers, we can define even to be the least realation for which the properties hold. It then follows by definition that even is satisfied by rules 1 and 2. Furthermore, because even is defined to be the least such relation, it is a subset of any other relation satisfying these rules. The net result of all this is that even describes only those values that it must describe to characterise the even naturals.

If we wish to express that some relation P is closed under the rules for even, the following characteristic statement is made:

$$(P\ 0) \land (\forall\, n.\ P\ n \supset P(n+2))$$

Or, in the case of even itself we could define:

$$\forall\, m.\ \mathsf{even}\ m = \forall\, P.\ ((P\ 0) \land (\forall\, n.\ P\ n \supset P(n+2))) \supset P\ m$$

Now, if we wished to prove that some property P' holds of even, we are required to show that the set $S = \{n \mid P'\ n\}$ is closed under the rules for even. Because even is defined to be the least such set, we have that even $\subseteq S$. This means that the property in question (P') holds of even.

The induction principle just used in reasoning about even is known as rule induction, and is necessary for many proofs involving inductive definitions. The way this principle is embedded in the HOL system is discussed in [8].

The rules for Femto-VHDL will be of the same form as those above, but will be made up of more complex components. As mentioned, the formulas will be elements of some transition system, the theoretical underpinnings of which are given in [9]. In general, formulas will look like:

$$env \vdash a \xrightarrow{trans} b$$

where $\xrightarrow{trans}$ is the name of the transition relation, and $\vdash$ separates the environment (env) used by the formula from the items being related (a and b). Such a formula should be read, "in the environment env, a evaluates to b in the relation $trans$".

8 The Semantics of Femto-VHDL

We now present the rules used to describe the semantics of Femto-VHDL. A good starting point is the rules that specify the evaluation of Boolean expressions. We will then give the rules for Femto-VHDL sequential statements and finish our exposition by formalising the operation of Femto-VHDL concurrent statements. But, before introducing any rules it will be necessary to explain the infrastructure upon which they are built.

8.1 Types and Functions

The types needed to make the semantics work are really very simple. They were developed to correspond with VHDL users' intuition about the structuring of information in VHDL itself. We will make use of set theory in much of what follows, as it is not only a convenient notation for expressing the concepts that we are interested in, but also provides many simple operations that capture the essence of the actions required by the semantics.

Three aliases are used to make what is explained later more readable. We use the name "value" to represent Booleans, "time" for the natural numbers, and "name" for signal names. By employing these aliases, we may define the types used in the semantics as follows:

Type	*Ranges Over*	*Type*
γ	events	(name) set
σ	state	(name $\times$ value) set
τ	transactions	time $\rightarrow$ state
ρ	environment	(time $\times$ state $\times$ events)

Recall that an event is a change in signal value. γ is simply a collection of signal names for which this is the case. State (σ) is used to represent the value of all the signals in a program. It is analogous to the "state" employed in the algorithms of Section 5. Transactions (τ) ensure a way of mapping from future times to state, and will provide the underlying framework for scheduling. Environment (ρ) is used to represent the static view of the world in which a given simulation cycle unfolds, and allows us to keep track of the current time for scheduling purposes, the current state of the signal values, and the signals for which there are currently events.

Three functions are used in defining the semantics. The first is valcalc, whose purpose is to calculate the value of a signal *sig* in a state σ.

$$\mathsf{valcalc}\ sig\ \sigma = \mathsf{choice}\ \{val \mid (sig, val) \in \sigma\}$$

The function choice used in this definition selects an arbitrary element from a non-empty set. The use of choice is justified here because the set from which the choice is made will always be a singleton set (signals have unique values).

Whenever a signal assignment statement is encountered, it becomes necessary to "post", or schedule, a transaction to take place at some future point of computation to effect that assignment. The function post does this, while at the same time implementing the pre-emptive scheduling of transactions demanded by VHDL. post uses the current transactions τ, the signal for which the assignment is to be made *sig*, the value that signal is supposed to take on *val*, and the time at which that assignment is to take place t to create a new version of future transactions by returning a function from time to state. If the time supplied to this function is the one at which the signal assignment is to take place, we return a set consisting of a "stripped" version of the projected state with the given signal-value pair inserted into it. Otherwise,

we simply return the stripped version of the projected state. This stripping process is performed by a function local to post called stripped, which in fact does the pre-emption part of scheduling a new transaction. stripped, when given a time t'', will return a version of our starting τ in which all pending transactions on signal sig have been removed when t'' is greater than or equal to the time at which the current scheduling is to take place. The function insert below should be understood as simple set insertion.

$$
\begin{array}{l}
\mathsf{post}\ sig\ val\ \tau\ t = \\
\quad let\ \mathsf{stripped}\ t'' = \\
\qquad (\tau\ t'') - \{(x,y) \mid (x,y) \in (\tau\ t'') \wedge (x = sig) \wedge (t'' \geq t)\} \\
\quad in \\
\qquad \lambda t'.\ \mathsf{if}\ (t' = t)\ \mathsf{then}\ ((sig, val)\ \mathsf{insert}\ (\mathsf{stripped}\ t')) \\
\qquad\quad \mathsf{else}\ (\mathsf{stripped}\ t')
\end{array}
$$

The final function defined is employed during the execution of concurrent statements. As we will see, each process makes use of the same starting τ when activated and goes on to post new transactions into it. When two processes finish their execution, it is necessary to combine their individual views of future behaviour into an overall one. The function zip does this by returning a function from time to state in which a set union is done at any given future point of computation on the sets of signal-value pairs found there. It is essential to bear in mind that this simple view of transaction resolution falls apart when the restriction that signals not have multiple drivers is removed. In that case the union would have to be replaced by a call to some function which ensures that appropriate user-defined resolution functions are applied to multiply driven signals, while continuing just to add to the global future those which are not [6].

$$\tau'\ \mathsf{zip}\ \tau'' = \lambda t.\ (\tau'\ t) \bigcup (\tau''\ t)$$

8.2 Rules for Boolean Expressions

The rules defining the semantics for the evaluation of VHDL Boolean expressions ($\xrightarrow{Bool}$) are numerous but rather simple. The single unary operator (not) applied to an expression e is defined to be nothing more than the logical negation of whatever e evaluates to. A similar pattern arises for the binary

operators in rules b_{ii} through b_{vii}. It is only when we come to rule b_{viii} that something interesting happens. Here we have an axiom stating that the value of a signal *sig* in a given state σ is obtained by calculating its value using the function valcalc described above.

$$\{\mathsf{b}_i\} \quad \frac{\sigma \vdash e \xrightarrow{Bool} x}{\sigma \vdash (\mathsf{not}\ e) \xrightarrow{Bool} \neg x} \qquad \{\mathsf{b}_{ii}\} \quad \frac{\sigma \vdash e_0 \xrightarrow{Bool} x \quad \sigma \vdash e_1 \xrightarrow{Bool} y}{\sigma \vdash (e_0\ \mathsf{and}\ e_1) \xrightarrow{Bool} (x \wedge y)}$$

$$\{\mathsf{b}_{iii}\} \quad \frac{\sigma \vdash e_0 \xrightarrow{Bool} x \quad \sigma \vdash e_1 \xrightarrow{Bool} y}{\sigma \vdash (e_0\ \mathsf{or}\ e_1) \xrightarrow{Bool} (x \vee y)}$$

$$\{\mathsf{b}_{iv}\} \quad \frac{\sigma \vdash e_0 \xrightarrow{Bool} x \quad \sigma \vdash e_1 \xrightarrow{Bool} y}{\sigma \vdash (e_0\ \mathsf{nand}\ e_1) \xrightarrow{Bool} \neg(x \wedge y)}$$

$$\{\mathsf{b}_{v}\} \quad \frac{\sigma \vdash e_0 \xrightarrow{Bool} x \quad \sigma \vdash e_1 \xrightarrow{Bool} y}{\sigma \vdash (e_0\ \mathsf{nor}\ e_1) \xrightarrow{Bool} \neg(x \vee y)}$$

$$\{\mathsf{b}_{vi}\} \quad \frac{\sigma \vdash e_0 \xrightarrow{Bool} x \quad \sigma \vdash e_1 \xrightarrow{Bool} y}{\sigma \vdash (e_0\ \mathsf{xor}\ e_1) \xrightarrow{Bool} (x \oplus y)}$$

$$\{\mathsf{b}_{vii}\} \quad \frac{\sigma \vdash e_0 \xrightarrow{Bool} x \quad \sigma \vdash e_1 \xrightarrow{Bool} y}{\sigma \vdash (e_0\ \mathsf{xnor}\ e_1) \xrightarrow{Bool} \neg(x \oplus y)}$$

$$\{\mathsf{b}_{viii}\} \quad \frac{}{\sigma \vdash sig \xrightarrow{Bool} \mathsf{valcalc}\ sig\ \sigma}$$

8.3 Rules for Sequential Statements

The relation $\xrightarrow{Seq}$ describes the evaluation of Femto-VHDL sequential statements. The relation makes use of $\xrightarrow{Bool}$ in many places, and the σ that is employed should be understood to be the state component of the environment in rules ss_i through ss_{iv}. Sequencing of Femto-VHDL statements is covered in rule ss_i, and the evaluation of conditionals is defined in rules ss_{ii} and ss_{iii}. The most important statement, from the point of view of describing the simulation model of VHDL, occurs in the postulation of rule ss_{iv}. Here we show

what happens during the evaluation of a transport delay signal assignment. For a signal assignment statement sig := (e, dly) where e evaluates to x in σ, the current time , and the current transactions τ, we say that the result of performing the assignment are new transactions created by using the function post to schedule into τ the value x for signal sig at time $(t + dly)$.

$$\{\mathsf{ss}_i\} \quad \frac{\rho \vdash \langle ss_0, \tau\rangle \xrightarrow{Seq} \tau' \qquad \rho \vdash \langle ss_1, \tau'\rangle \xrightarrow{Seq} \tau''}{\rho \vdash \langle ss_0\ ;\ ss_1, \tau\rangle \xrightarrow{Seq} \tau''}$$

$$\{\mathsf{ss}_{ii}\} \quad \frac{\sigma \vdash e \xrightarrow{Bool} \mathsf{true} \qquad (t,\sigma,\gamma) \vdash \langle ss_0, \tau\rangle \xrightarrow{Seq} \tau'}{(t,\sigma,\gamma) \vdash \langle e \texttt{ => } ss_0 \mid ss_1, \tau\rangle \xrightarrow{Seq} \tau'}$$

$$\{\mathsf{ss}_{iii}\} \quad \frac{\sigma \vdash e \xrightarrow{Bool} \mathsf{false} \qquad (t,\sigma,\gamma) \vdash \langle ss_1, \tau\rangle \xrightarrow{Seq} \tau'}{(t,\sigma,\gamma) \vdash \langle e \texttt{ => } ss_0 \mid ss_1, \tau\rangle \xrightarrow{Seq} \tau'}$$

$$\{\mathsf{ss}_{iv}\} \quad \frac{\sigma \vdash e \xrightarrow{Bool} x}{(t,\sigma,\gamma) \vdash \langle sig := (e, dly), \tau\rangle \xrightarrow{Seq} \mathsf{post}\ sig\ x\ \tau\ (t + dly)}$$

8.4 Rules for Concurrent Statements

The rules that describe the execution of Femto-VHDL concurrent statements are really those which define how a VHDL simulation cycle works. Rule cs_i shows that if a process' sensitivity list has no signals in common with the current events (γ), then the process does not activate, and the transactions (τ) remain unchanged. If, however, there is an intersection (rule cs_{ii}), then the new transactions (τ') are derived from the execution of the process' sequential statements using $\xrightarrow{Seq}$. Finally, when two processes are executed concurrently, the resulting transactions are created by an application of zip to collect together the transactions derived by running each process separately. Note that the execution is deterministic because we do not allow signals to have multiple drivers (i.e. a signal name for which there is a transaction will appear at any given time in either τ' or τ'', but not both). Syntactically, we require that a signal not appear on the left-hand side of an assignment in more than one process. If we remove the restriction, zip will have to be modified as described in Section 8.1.

$$\{\mathsf{cs}_i\} \quad \frac{}{(t,\sigma,\gamma) \vdash \langle sl : ss, \tau \rangle \stackrel{Cyc}{\longrightarrow} \tau} \quad (sl \bigcap \gamma) = \{\}$$

$$\{\mathsf{cs}_{ii}\} \quad \frac{(t,\sigma,\gamma) \vdash \langle ss, \tau \rangle \stackrel{Seq}{\longrightarrow} \tau'}{(t,\sigma,\gamma) \vdash \langle sl : ss, \tau \rangle \stackrel{Cyc}{\longrightarrow} \tau'} \quad (sl \bigcap \gamma) \neq \{\}$$

$$\{\mathsf{cs}_{iii}\} \quad \frac{\rho \vdash \langle cs_0, \tau \rangle \stackrel{Cyc}{\longrightarrow} \tau' \quad \rho \vdash \langle cs_1, \tau \rangle \stackrel{Cyc}{\longrightarrow} \tau''}{\rho \vdash \langle cs_0 \mid\mid cs_1, \tau \rangle \stackrel{Cyc}{\longrightarrow} \tau' \ \mathsf{zip}\ \tau''}$$

9 Femto-VHDL in HOL

Having given the semantics of Femto-VHDL as a suite of inductively defined relations, we may now use it to perform some experiments with actual Femto-VHDL programs. Throughout the following examples, we will use the rules as they are embedded in the HOL system. We have also coded the abstract syntax of Femto-VHDL as a recursive type definition in the logic [7]. The semantics can be viewed as an interpreter which "runs" Femto-VHDL programs by proof. We will then be able to see what the characteristic future transactions are which express the behaviour of a given program within some environment. Note that the examples are coded in actual VHDL rather than the abstract syntax which we have defined in order to ease readability.

In the first example, we will endeavour to show that a process with an absent (empty) sensitivity list never activates, and therefore does not create any new transactions. We make use of the following process (also taken from [1]).

```
PROCESS
BEGIN
 C <= TRANSPORT (NOT C) AFTER dly;
END PROCESS;
```

If we create the following term in HOL, where ρ, τ and τ' are variables:

$$"\rho \vdash \left\langle \begin{array}{l}\texttt{PROCESS}\\ \texttt{BEGIN}\\ \;\texttt{C <= TRANSPORT (NOT C) AFTER dly;}\\ \texttt{END PROCESS;}\end{array}, \tau\right\rangle \xrightarrow{Cyc} \tau'"$$

we can ask HOL to "run" the term to generate the following theorem (the symbol |- signifies a theorem in HOL):

$$\texttt{|-} \forall \rho\, \tau\, \tau'.\; \rho \vdash \left\langle \begin{array}{l}\texttt{PROCESS}\\ \texttt{BEGIN}\\ \;\texttt{C <= TRANSPORT (NOT C) AFTER dly;}\\ \texttt{END PROCESS;}\end{array}, \tau\right\rangle \xrightarrow{Cyc} \tau' = (\tau' = \tau)$$

Here we see that the theorem states that the future transactions τ' are the same as the initial transactions τ. The result is what we wanted, and demonstrates that for any environment ρ the process never activates.

We now add a sensitivity list that will cause the process to activate whenever there is an event on the signal `C`. In so doing, we would expect the process to cause a transaction to be posted `dly` units from now that will give the signal `C` its inverse value.

```
PROCESS (C)
BEGIN
 C <= TRANSPORT (NOT C) AFTER dly;
END PROCESS;
```

As in the previous example, we first create a term in HOL that sets the process up for an execution within the $\xrightarrow{Cyc}$ transition system. A point to note is that we have given a somewhat more concrete environment this time. In particular, we state that there is an event on `C`.

```
                          PROCESS (C)
                          BEGIN
"(now, σ,{C}) ⊢ ⟨          C <= TRANSPORT (NOT C) AFTER dly;  ,τ⟩ --Cyc--> τ' "
                          END PROCESS;
```

Execution of the process leads to the following theorem, and the result of the execution is as expected.

```
|- ∀ now σ τ τ'.
                          PROCESS (C)
                          BEGIN
   (now, σ,{C}) ⊢ ⟨         C <= TRANSPORT (NOT C) AFTER dly;  ,τ⟩ --Cyc--> τ' =
                          END PROCESS;
   (τ' = (post C ¬(valcalc C σ) τ (now + dly)))
```

We see that the future transactions result from posting into the current transactions on signal `C` its present inverse at `dly` offset from *now*. Special attention should be paid to the fact that the theorem holds for all *now*, σ, τ and τ'. That is, it describes the characteristic behaviour of the process whenever there is an event on the signal `C`. We should also note that `dly` can be just `0` NS. This reinforces the notion that δ-delay is nothing more than a normal occurrence of a standard simulation cycle rather than something special.

As a final example, we add a second process to our Femto-VHDL program which inserts a 0-delay buffer into the system. We expect the behaviour of our new program to involve the an application of `zip` to combine the future transactions arrived at separately by each process. This result should be characterised by the future transactions derived in the previous example being combined with a statement about `C1` always getting the current value of `C`.

```
PROCESS (C)
BEGIN
 C <= TRANSPORT (NOT C) AFTER dly;
END PROCESS;

PROCESS (C)
BEGIN
 C1 <= TRANSPORT C AFTER 0 NS;
END PROCESS;
```

As in the two previous examples, we create a term to express the relationship we wish to investigate. Here again, we have declared that there is currently an event on C.

"$(now, \sigma, \{\texttt{C}\}) \vdash \langle$

```
PROCESS (C)
BEGIN
 C <= TRANSPORT (NOT C) AFTER dly;
END PROCESS;

PROCESS (C)
BEGIN
 C1 <= TRANSPORT C AFTER 0 NS;
END PROCESS;
```

$, \tau\rangle \xrightarrow{Cyc} \tau'$ "

The result after execution is as expected. In particular we note that the individual future transactions are combined using `zip` to form an overall notion of the future. We also see that `C1` takes on whatever `C`'s value is in the current environment.

$\vdash \forall\ now\ \sigma\ \tau\ \tau'.$

$(now, \sigma, \{\mathtt{C}\}) \vdash \langle$

```
PROCESS (C)
BEGIN
 C <= TRANSPORT (NOT C) AFTER dly;
END PROCESS;

PROCESS (C)
BEGIN
 C1 <= TRANSPORT C AFTER 0 NS;
END PROCESS;
```

$, \tau\rangle \xrightarrow{Cyc} \tau' =$

$(\tau' = (\mathsf{post}\ \mathtt{C}\ \neg(\mathsf{valcalc}\ \mathtt{C}\ \sigma)\ \tau\ (now + \mathtt{dly}))$

zip

$(\mathsf{post}\ \mathtt{C1}\ (\mathsf{valcalc}\ \mathtt{C}\ \sigma)\ \tau\ now))$

10 Conclusions and Future Work

We have seen that the semantics of the VHDL simulator, in particular a single simulation cycle, is a useful way of expressing the behaviour of individual language constructs. We have also seen how this semantics can easily and naturally be given as a collection of inductively-defined transition relations. In doing so, we have made both unit- and 0- (or δ-) time form a part of the same whole, with no special interpretation required in the treatment of one versus the other.

Using our formalisation, we have been able to animate the semantics as an interpreter within the HOL system. This allows us to examine specific Femto-VHDL programs and automatically prove concise theorems describing their characteristic behaviour. We have, as a result of this embedding in HOL, been able to see that our semantics relates closely to the intuition behind the VHDL simulation cycle.

In future, we shall investigate ways in which different Femto-VHDL programs purporting to describe the same thing may be proved to be so under some notion of equivalence. A case in point was raised in our discussion of results: how do we know that the uniprocess oscillator and the dual process one represent the same thing? The first task is, of course, to develop the notion of equivalence itself.

11 Acknowledgements

I am grateful to Dr. Juanito Camilleri for his help in coming to grips with this semantics. Further thanks are also due to Mike Gordon, Tom Melham and Monica Nesi for their comments during the development of this report.

References

[1] R. Airiau, J.-M. Bergé, V. Olive, and J. Rouillard, *VHDL: du Langage à la Modélisation*, Presses Polytchniques et Universitaires Romandes (1990).

[2] Larry M. Augustin, Benoit A. Gennart, Youm Huh, David C. Luckham, and Alec C. Stanculescu, 'An Overview of VAL', Technical Report CSL-TR-88-367, Stanford University, Stanford, California (October 1988).

[3] R. Boulton, M. Gordon, J. Herbert, and J. Van Tassel, 'The HOL Verification of ELLA Designs', in: *1991 International Workshop on Formal Methods in VLSI Design*, edited by P.A. Subrahmanyam, (also Univ. of Cambridge Tech Report 199, August 1990) Springer-Verlag (1991).

[4] Juanito Camilleri, 'Symbolic Compilation and Execution of Programs by Proof: A Case Study in HOL', Technical Repot 240, University of Cambridge Computer Laboratory, Cambridge, England (December 1991).

[5] Ivan V. Filippenko, 'VHDL Verification in the State Delta Verification System (SDVS)', in: *1991 International Workshop on Formal Methods in VLSI Design*, edited by P.A. Subrahmanyam, Springer-Verlag (1991).

[6] Institute of Electrical and Electronics Engineers, *IEEE Standard VHDL Language Reference Manual*, IEEE Press, New York (1988).

[7] T.F. Melham, 'Automating Recursive Type Definitions in Higher-Order Logic', in: *Current Trends in Hardware Verification and Automated Deduction*, edited by G. Birtwistle and P.A. Subrahmanyam, Springer-Verlag (1988).

[8] T.F. Melham, 'A Package for Inductive Relation Definitions in HOL', in proceedings: *1991 International Workshop on the HOL Theorem Proving System and its Applications*, IEEE Computer Society Press (1991), in publication.

[9] Gordon Plotkin, 'A Structural Approach to Operational Semantics', Technical Report DAIMI FN-19, Computer Science Dept., Aarhus Univ. (September 1981).

[10] A. Salem and D. Borrione, 'Formal Reasoning About Signal Attributes in VHDL', in proceedings: *VHDL Forum for CAD in Europe*, Spring 1991 meeting.

[11] John P. Van Tassel, *The Semantics of VHDL with VAL and HOL: Towards Practical Verification Tools*, M.Sc. Thesis, Dept. of Computer Science and Engineering, Wright State University (1989).

[12] John Van Tassel and David Hemmendinger, 'Toward Formal Verification of VHDL Specifications', in: *Applied Formal Methods For Correct VLSI Design*, edited by Luc Claesen, Elsevier Science Publishers (1990).