The Semantics of Behavioral VHDL'93 Descriptions*

Egon Börger
Dipartimento di Informatica
Università di Pisa
I-56125 Pisa, Italy

Uwe Glässer
Heinz Nixdorf Institut
Universität-GH Paderborn
D-33098 Paderborn, Germany

Wolfgang Müller
Cadlab
Universität-GH Paderborn
D-33102 Paderborn, Germany

Abstract

We present a rigorous but transparent semantic definition of VHDL'93 covering the complete signal behavior and time model including the various wait statements and signal assignment statements. We present a VHDL interpreter in the form of rules of a concurrent evolving algebra which faithfully reflects and supports the view given in the VHDL'93 standard language reference manual.

1 Introduction

Approaching the definition of a formal semantics of the IEEE Std-1076 hardware description language VHDL'87 as well as of the new VHDL'93[1][VHDL93] standard is of high interest for the synthesis and the formal verification of VHDL models.

Borrione and Paillet [BoPa87] have done first investigations defining the semantics of a VHDL'87 subset in terms of a functional model. Further investigations can be found in [Sal92, SaBo93]. The definition of a subset of the VHDL'87 semantics in terms of Booyer-Moore Logic is presented by Read and Edwards in [ReEd94]. Wilsey [Wilsey90] defines the semantics of a small VHDL'87 subset based on interval temporal logic. Davis [Davis93] has introduced the denotational semantics of the VHDL simulation cycle by the use of an intermediate language that is derived from a very limited behavioral VHDL'87 subset. The structural operational semantics of a VHDL'87 subset (Femto-VHDL) for HOL verification is presented by Van Tassel in [VaTa93]. Damm et al. give the detailed structural operational semantics of a VHDL'87 subset based on transition systems in [DJS94]. Müller introduces a partly denotational and partly operational approach (High-Level Semantics of VHDL'93) in [Mue93]. Therein the dynamic semantics is sketched by partially ordered events defining Petri-Net-like structures. Olcoz and Colom introduce in [OlCo93] a formal model based on elaborated VHDL'87 and its execution in terms of Colored Petri-Nets which covers all basic behavioral properties of VHDL'87. In [OlCo94] they provide a detailed investigation of the VHDL'87 simulation cycle distinguishing three semantical layers: syntax checking and design library building, elaboration, and execution.

Almost all of these approaches consider rather restricted subsets of VHDL'87. None of the above approaches covers the new features of VHDL'93 in details. The given specifications, due to the applied formalisms, are often too complex to convey a correct understanding of the semantics of VHDL for educational purpose. In this paper we provide a rigorous but simple semantic definition of elaborated VHDL'93 including the new features of postponed processes and pulse rejection limit [VHDL93]. We define the formal semantics of VHDL'93 in terms of Gurevich's *concurrent evolving algebras* [Gur91, Gur94]. The flexibility of this concept allows us to produce our specification following the terminology and the view presented by the standard language reference manual [VHDL93].

The remainder of this paper is organized as follows. In Section 2 we briefly introduce what is needed from concurrent evolving algebras. In Section 3 we develop a mathematical definition of VHDL in terms of a *VHDL Algebra* considering the signal assignment and wait statements as well as the full computational model of interaction between the user defined processes and the kernel process. Section 4 gives a conclusion and future directions.

2 Evolving algebras

Evolving algebras (EAs) can be understood as 'pseudocode over abstract data', without any particular theoretical prerequisites. Here we list only the basic definitions and refer for a rigorous formalization to [Gur91, Gur94].

*Published in the Proceedings of EURO-DAC'94/EURO-VHDL'94, Grenoble, September 19-23, IEEE Press, Los Alamitos, CA, 1994.

[1] Some publications refer to this standard as VHDL'92

The abstract data come as elements of (possibly not furthermore specified) sets (domains, *universes*). The operations allowed on universes will be represented by *partial functions*, where we write $f(x) = undef$ if f is *undefined* at x. Dynamic changes are obtained through *function updates* of form " $f(t_1, \ldots, t_n) := t$ " whose execution is to be understood as *setting* the value of function f at given arguments. Note that the 0-ary functions play the role of *variables* in imperative programming languages.

An EA is defined by a finite set of *transition rules* of form " **if** *Cond* **then** *Updates* " where *Cond* (condition or guard) is an expression, the truth of which triggers *simultaneous* execution of all updates in the finite set of *Updates*. Our rules will always be constructed so that the guards imply *consistency* of updates. The resulting description determines the dynamics of a very large transition system. We are usually only interested in states reachable from some designated *initial states*, which may be specified in various ways.

An EA often comes together with a set of *integrity constraints*, i.e., extralogical axioms and/or rules of inference, specifying the intended domains. In applications of EAs one usually encounters a *heterogeneous* signature with several universes, which may in general grow and shrink in time—update forms are provided to extend a universe: "**extend** A **by** $t_1, \ldots, t_n$ **with** Updates **endextend**" where *Updates* may (and should) depend on t_i's, setting the values of some functions on *newly created* elements t_i of A. Without giving explicit declarations, we shall assume the availability of certain standard mathematical universes such as booleans, integers, lists of whatever etc (as well as the standard operations on them).

A concurrent EA Π, is a pair (Ag, Mod) where Ag is a finite set of *agents* and Mod is a function that associates a sequential EA with each element of Ag. A concurrent EA can be seen as the definition of a set of concurrently running agents. Each agent is specified through a finite set of *transition rules* operating on a globally shared structure; this also covers shared variables. We illustrate the basic concepts of EAs by two examples:

```
if $Condition$                  if $List \in LIST$
then $A := B$                   thenif $List \neq \langle\rangle$
     $B := A$                       then $List := tail(List)$
```

Example 1: Exchange Values

Example 2: Remove the First Element of a List

Example 1 defines simultaneous updates of the functions A and B to be performed each time *Condition* evaluates to *true*. Since the assignments are performed in parallel, A becomes the value of B and vice versa. *Example 2* defines a rule specifying that each nonempty *List* from the domain *LIST* is to be replaced by its corresponding list tail. The expression $List \in LIST$ is used as an abbreviation referring to any valid instantiation of *List* within the underlying domain *LIST*.[2]

The concepts of EAs directly apply to our view of VHDL whose agents are n user defined processes and one kernel process. Our VHDL specification comes in the form of two modules one for the kernel process and one for the asynchronously operating agents of user defined processes. Note that for a sequential user defined process, to execute a rule in which variables occur means to execute simultaneously all instantiations of this rule obtained by replacing the variables by elements of the corresponding domains. This intuitive explanation should be sufficient for a correct understanding of our rules; for a rigorous foundation of this so-called *lockstep interpretation* of sequential EAs see [Gur94].

3 The formal model

3.1 Basic concepts

[VHDL93] defines the semantics of the VHDL IEEE standard in terms of a simulator. In our description we adopt this view and also the basic terminology of [VHDL93] as far as possible (see also [OlCo94]).

The model of event driven simulation is based on a finite number of user defined processes $P \in PROCESS$ which—under the supervision of the simulation kernel process—concurrently compute new VALUEs for given SIGNALs. These signals may cause events at specified points in *TIME*. Given the underlying discrete VHDL time model, the domain *TIME* is linearly ordered and contains the distinguished element *current time* T_c.

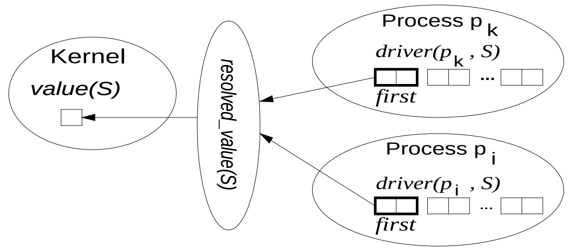

Figure 1: Updating Signals

A process P cannot directly assign a value to a signal S but has to schedule the signal value *val*, desired at time t, into an ordered list $driver(P, S)$ consisting of pairs (val, t), so–called *TRANSACTIONs* (see Figure 1). The transactions of a driver are linearly or-

[2]In the remainder of this paper domains are denoted by capitalized names whereas the related variables are represented by the same name but with only the first letter capitalized.

dered by their time components. For the first element of each driver holds that its time component is $\leq T_c$. The IEEE standard defines that the time components of the other transactions are $> T_c$. Since *value*(S) for each signal S usually is updated by a set *drivers(S)*, possible conflicts between the transactions of the active drivers of S are resolved via a user defined resolution function; we represent the latter by a *resolved_value* function which out of the first elements of active *drivers(S)* chooses one value for the update of S.

During a simulation cycle a process becomes *suspended* when reaching a wait statement which delays the process execution until (a) the *timeout* of a specified TIMER is reached, or (b) one of the corresponding signals is updated, or (c) a given expression becomes true if one of the corresponding signals is updated.

If all user defined processes are suspended, the kernel process becomes *enabled* and (i) determines the value for the next time T_n; (ii) sets the new current simulation time T_c if required; (iii) updates the relevant signals and resets the corresponding active drivers to *inactive*; and (iv) resumes the suspended processes (the processes are invoked by becoming *enabled*).

The kernel decides the type of processes which are invoked during the next simulation cycle ($enabled = postponed$ or $enabled = process$)[3]. Thus, we have to distinguish three consecutive phases for the kernel (see also Figure 4): the *delta cycle* in which the *current time* T_c does not change, but new values might be assigned to signals and suspended non–postponed processes are invoked; the *postponed cycle* when, just before T_c will advance, the postponed processes are *enabled*; and the *time cycle* where the simulation time is advanced.

As initialization we suppose the kernel to be in phase *delta cycle*, current time T_c to be set to 0, $enabled = process$, and the initialization of drivers according to the definitions in [VHDL93]. We also assume that first the non-postponed processes and then the postponed processes are invoked once.

3.2 User defined processes

The rules in this section constitute an agent, one for each user defined process defining the semantics of signal assignment and of various wait statements.

3.2.1 Processing statements

In order to concentrate on the essential behavioral features of VHDL'93, we assume that the control flow of each (sequential) iterative process is determined by

[3] $enabled = kernel$ denotes that the kernel process is active.

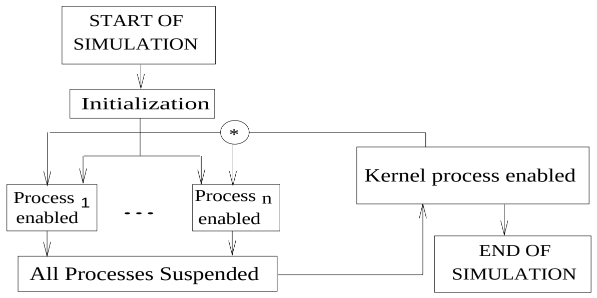

Figure 2: The Basic Simulation Cycle

the environment which provides the dynamic changes of values for the external function *program_counter*[4]. To express that a user defined process P can be executed only when P is not suspended and when all processes of the same type as P are enabled, we use the following abbreviation:

$Process\ does$ `statement` $\equiv$
$suspended(Process) = false \wedge type(Process) = enabled$
$\wedge program\_counter(Process) =$ `statement`

Recall that *enabled* can assume the three values *kernel, process, postponed* indicating that the agents of that type are the ones currently executed. Since $type(P)$ is either *postponed* or *process* the condition $type(P) = enabled$ ensures that in each simulation cycle only the corresponding processes are executed—either postponed or non-postponed processes. Type *process* stands for non–postponed processes.

3.2.2 Signal assignments

In this paper we consider only the most general form of signal assignment—the inertial signal assignment with an explicit specification of the rejection pulse limit. This statement newly introduced to VHDL'93 supersedes the behavior of the transport signal assignment and the inertial signal assignment of VHDL'87. The statement is of form $\langle S \Leftarrow \underline{\text{REJECT}}\ Pulse\ \underline{\text{INERTIAL}}\ X_1\ \underline{\text{AFTER}}\ Time_1, \ldots\rangle$. In VHDL'93, the optional rejection pulse limit *Pulse* specifies a time interval of transactions which are not rejected (removed) from the driver when scheduling the new waveform elements. For $Pulse = Time_1$, the behavior of the statement is the same as a transport signal assignment. When $Pulse = 0$ the statement is equivalent to the VHDL'87 inertial signal assignment.

The general intuitive meaning of a signal assignment when carried out by some process P is to schedule future values, on the driver identified through

[4] An external function in the sense of [Gur91] is a function which is not updated by the rules of the system under consideration; nevertheless such a function might be updated by the environment and thus represents a precise interface for the system.

$driver(P, S)$ of process P of signal S. This is done by inserting new transactions (waveform elements) to the driver and removing previously scheduled transactions (preemptive scheduling). Recall, that VHDL requires the sequence of the waveform elements of the statements as well the transactions in the driver to be strictly increasing w.r.t. the time component. Each driver has exactly one distinguished element ($first$) which is $\leq T_c$. Note here that in our model time is represented by the absolute time w.r.t. T_c.

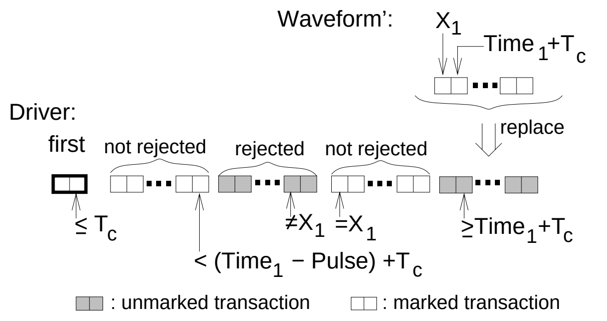

Figure 3: Preemptive Scheduling for Inertial Delay

In the special case of $Time_1 = 0$, the whole driver is replaced by the list $\langle (X_1, Time'_1), \ldots, (X_n, Time'_n) \rangle$ where $Time'_i = Time_i + T_c$ denotes the absolute time w.r.t. the current simulation time T_c. Since this means that the first transaction is replaced, the driver has to be set to $active$ by definition.

In the other case, when $Time_1 > 0$, the waveform, which by definition is linearly ordered, is simply appended to the previously shrunken driver. The driver is shrunk by only keeping the transactions with $< Time_1 + T_c$. We describe this by the function $\upharpoonright_<$: $DRIVER \times TIME \to DRIVER$ which yields the driver containing precisely those transactions which have time component $< Time_1 + T_c$. The further actions are defined in [VHDL93] by a 5 step algorithm in terms of marking transactions and removing the unmarked transactions in Step 5.
In Step 1, the newly inserted waveform elements (new transactions) are marked. Step 2 marks those transactions whose time components are "less than the time at which the first new transaction is projected to occur minus the pulse rejection limit." Step 3 defines that "for each remaining unmarked, old transaction, the old transaction is marked if it immediately precedes an unmarked transaction and its value component is the same as that of the marked transaction". Finally, Step 4 marks the first element.

Corresponding to this algorithm, in our model, the resulting driver $driver(S, P)$ is obtained as a composition of four separate sublists[5]: the first element, the part kept by the rejection pulse limit, the not rejected rest, and the new transactions. In this composition $driver'$ reflects the shrunken driver. $driver''$ corresponds to Step 2 of the above algorithm. The function $reject$ implements Step 3. $reject$ only keeps those transactions at the right end of the driver whose value is equal to the value of the first new transaction (X_1):

```
$reject(TList, Val) \equiv$
**if** $TList = \langle\rangle \vee value(last(TList)) \neq Val$
**then return** $\langle\rangle$
**else return** $reject(front(TList, Val))^\wedge last(TList)$
```

The full specification of the statement is given by the following transition rule[6].

```
**if** $Process\ does\ \langle S \Leftarrow \underline{\text{REJECT}}\ Pulse\ \underline{\text{INERTIAL}}$
        $X_1\ \underline{\text{AFTER}}\ Time_1, \ldots, X_n\ \underline{\text{AFTER}}\ Time_n\rangle$
**thenif** $Time_1 = 0$
    **then** $driver(Process, S) := Waveform'$
        $active(driver(Process, S)) := true$
    **else** $driver(Process, S) :=$
        $first(driver(Process, S))^\wedge driver''^\wedge$
        $reject(driver''', X_1)^\wedge Waveform'$
**where**
    $driver' = tail(driver(Process, S) \upharpoonright_< (Time_1 + T_c))\wedge$
    $driver'' = (driver' \upharpoonright_< ((Time_1 - Pulse) + T_c))\wedge$
    $driver''' = driver' - driver''$
```

In this rule $Waveform'$ is composed by replacing the relative time of each element of $Waveform$ by the absolute time $Time'_j$: $Waveform' = \langle (X_1, Time'_1), \ldots \rangle$.

3.2.3 Wait statements

The rules for wait statements define how processes are suspended due to wait requirements for a specified time period, a signal, or the truth of a condition.

For modeling *WAIT FOR* statements, we introduce the concept of timers. Timers are objects which are set when a *WAIT FOR* statement is evaluated. That is, if *Process WAITs FOR Time*, then a new timer with timeout $T_c + Time$ is created at which the *Process* is suspended by $suspended(Process) := true$. The function *waiting* identifies the process which is waiting for the expiration time.

```
**if** $Process\ does\ \langle \underline{\text{WAIT}}\ \underline{\text{FOR}}\ Time\rangle$
**then** $suspended(Process) := true$
    **extend** $TIMER$ **by** $t$ **with**
    $timeout(t) := Time + T_c$
    $waiting(t) := Process$
    **endextend**
```

If a Process *WAITs ON* a set of *Signals* then the *Process* is suspended and added to the set of processes which are waiting for changes of a signal, i.e., each signal in this sensitivity list holds in $waiting(s)$ the

[5] The composition of lists is defined by the $^\wedge$ operator.

[6] Since we want to abstract from the details of expression evaluation we interpret X_i as placeholders for the corresponding values

set of processes which are suspended on s. If a Process *WAITs UNTIL* an expression *Expr* it is resumed when the expression evaluates to true. The evaluation of the current waiting condition, which is stored in *waitcond(Process)*, is performed iff at least one signal in this expression changes. Those signals are extracted from the expression by *condsignals(Expr)*.

```
if $Process\ does\ \langle \underline{\text{WAIT}}\ \underline{\text{UNTIL}}\ Expr \rangle$
then $waitcond(Process) := Expr$
     $suspended(Process) := true$
     if $s \in condsignals(Expr)$
     then $waiting(s) := waiting(s)^\frown\langle Process \rangle$
```

By *WAIT UNTIL* suspensions a *Process* is added to *waiting*(s) of each $s \in condsignals(Expr)$.

3.3 The kernel process

The kernel is an agent which starts executing only when all user defined processes are suspended. We abbreviate this by:

```
$AllProcessesSuspended \equiv$
$\forall P \in PROCESS : type(P) = enabled \Rightarrow$
$suspended(P) = true \wedge$
$enabled \in \{process, postponed\}$
```

This specifies that all processes of the currently enabled type *process* or *postponed* have to be suspended. Then the kernel computes whether the next simulation cycle has to be a delta cycle, a postponed cycle, or whether the time can be advanced and sets *phase* $\in$\{*delta_cycle, postponed_cycle, time_cycle*\} accordingly (see Figure 2).

If the expected next time T_n is equal to the current time T_c,[7] the kernel goes into phase *delta cycle*. Otherwise, if $T_n > T_c$, the kernel goes either from *delta cycle* to *postponed cycle* or from the latter to *time cycle* phase (see Figure 4). T_n is computed by taking the minimum of all timeouts $\geq T_c$ (*mintimer*) and of times of all drivers[8] (*mindriver*).

```
if $AllProcessesSuspended$
then $enabled := kernel$
     if $T_n = T_c$
     then $phase := delta\_cycle$
     elsif $phase = delta\_cycle$
          then $phase := postponed\_cycle$
          else $phase := time\_cycle$
               $UpdateDrivers(T_n)$
where $T_n = min\{mindriver,\ mintimer\}$
```

$mintimer =$
$\quad min\{timeout(t) \mid t \in TIMER, timeout(t) \geq T_c\}$ *and*
$mindriver =$
$\quad min\{time(t) \mid \exists d \in DRIVER : active(d) = i, t = t^i\}$,
where $t^{true} = first(d)$ *and* $t^{false} = scd(d)$.

[7] This is the case if there are some active drivers or if at least one timeout is set to T_c.

[8] In the case of active drivers the time of the newly scheduled first element has to be considered. In the case of inactive drivers the time of the second element has to be considered.

Note here, that in the case of a *time_cycle* all drivers have to be updated w.r.t. the new time T_n by *UpdateDrivers*. If any transaction is scheduled for the T_n in any driver these drivers are updated to their tails, i.e., the first element is removed. By definition these drivers become active. Due to the ordering of drivers, we can determine these drivers by checking their second element to T_n.

```
$UpdateDrivers(t) \equiv$
if $d \in DRIVER \wedge tail(d) \neq \langle\rangle \wedge time(scd(d)) = t$
then $d := tail(d)$
     $active(d) := true$
```

There are three rules describing what the kernel does in each of its three different phases (*delta_cycle, postponed_cycle, time_cycle*). In the delta cycle and time cycle phase, the kernel checks whether there may be any events on signals (*EventOnSignals*). Thereafter the kernel changes to subphase *process_resumption* in order to resume the user defined processes. In phase *time_cycle*, the kernel additionally has to advance the current time to T_n before. In the case that T_n exceeds the limit $TIME'HIGH$ the simulation completes by setting *enabled* := *undef* (cf. Figure 2). When in phase *postponed_cycle*, the kernel simply invokes all already resumed postponed processes by setting *enabled* := *postponed*.

```
Delta Cycle :
    if $enabled = kernel \wedge phase = delta\_cycle$
    then $EventOnSignals$
         $enabled := process\_resumption$
Time Cycle :
    if $enabled = kernel \wedge phase = time\_cycle$
    then $EventOnSignals$
         if $T_n > TIME'HIGH$
         then $enabled := undef$
         else $T_c := T_n$
              $enabled := process\_resumption$
    where $T_n = min\{mindriver,\ mintimer\}$
Postponed Cycle :
    if $enabled = kernel \wedge phase = postponed\_cycle$
    then $enabled := postponed$
```

EventOnSignals sets an event on those signals which have at least one active driver and whose current value *value* is different from the newly resolved value. In the case of an event, the current value is replaced by

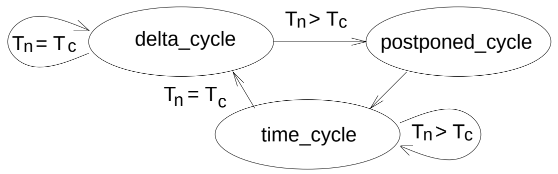

Figure 4: Different Phases of the VHDL Simulator

the resolved value (*resolved_value*) applying the user defined *resolution function* (see also Figure 1). The active drivers are reset since this property holds only for one simulation cycle.

```
$EventOnSignals \equiv$
if $\exists d \in drivers(Signal) : active(d) = true \wedge$
   $value(Signal) \neq resolved\_value(Signal)$
then $event(Signal) := true$
     $value(Signal) := resolved\_value(Signal)$
     $active(d) := false$
```

In subphase *process_resumption* which applies to delta and time cycles only processes may resume on signals and on expired timers. For starting the resumed processes, *phase* has to be set to *enabled* again.

```
if $enabled = process\_resumption \wedge$
   $phase = delta\_cycle \vee phase = time\_cycle$
then $ResumeOnTimers$
     $ResumeOnSignals$
     $enabled := process$
```

ResumeOnTimers resumes all processes in *waiting* whose timeout has reached the already updated T_c. *ResumeOnSignals* resumes the processes which are sensitive to signals s with *event*(s) := *true*. All processes of the set *waiting* of each of these signal are resumed. In case of suspension by WAIT UNTIL, i.e., *waitcond* is defined, the corresponding condition *condvalue* has to be checked. When applying this function, each appearance of each signal S now refers to the already updated current values of S. Finally, *event* and *waitcond* have to be initialized for the next simulation cycle.

```
$ResumeOnTimers \equiv$
if $Timer \in TIMER \wedge timeout(Timer) = T_c$
then $suspended(waiting(Timer)) := false$

$ResumeOnSignals \equiv$
if $s \in SIGNAL \wedge event(s) = true$
then $event(s) := false$
  if $Process \in waiting(s)$
  thenif $waitcond(Process) = undef$
       then $suspended(Process) := false$
       elsif $condvalue(waitcond(Process)) = true$
          then $waitcond(Process) := undef$
               $suspended(Process) := false$
```

4 Conclusion & future directions

We presented a rigorous but yet intuitive VHDL'93 algebra aiming at a clean and complete model which supports the understanding of the VHDL'93 language reference manual. In this paper, due to limited space, we did not treat the behavior of shared variables and ports (cf. [BGM94]). In a next step we will start to investigate the implementation of adequate tools for machine assisted analysis and verification of EA models. We are also working an UDL/I algebra with the ultimate goal to provide a uniform framework for comparison of VHDL and UDL/I properties.

Acknowledgements

We would like to thank Uschi Hudson, Christel Oczko, Serafin Olcoz, Franz J. Rammig, Simon Read as well as the unknown reviewers for their valuable comments when reviewing this article.

References

[BGM94] E. Börger, U. Glässer and W. Müller. A Formal Specification of the Semantics of Behavioral VHDL'93 Descriptions. Technical Report, 1994 (to appear).

[BoPa87] D. Borrione and J.L. Paillet. An approach to the formal verification of VHDL descriptions. Report No. 683-I, IMAG/ARTEMIS, Grenoble, Nov. 1987.

[DJS94] W. Damm, B. Josko, and R. Schlör. Specification and Verification of VHDL-based System-Level Hardware Designs. In *Specification and Validation Methods*, E. Börger (ed.). OUP, Oxford, 1994 (to appear).

[Davis93] K.C. Davis. A Denotational Definition of the VHDL Simulation Kernel. In *CHDL'93*, Ottawa, May 1993, North-Holland, Amsterdam, 1993.

[Gur91] Y. Gurevich. Evolving Algebras – A Tutorial Introduction. In *Bulletin of the EATCS*, Feb. 1991, No.43, pp.264-284.

[Gur94] Y. Gurevich. Evolving Algebra 1993: Lipari Guide. In *Specification and Validation Methods*, E. Börger (ed.). OUP, Oxford, 1994 (to appear).

[Mue93] W. Mueller. Approaching the Denotational Semantics of Behavioral VHDL92 Descriptions. In *1st APCHDLSA*, Brisbane, Australia, Dec. 1993.

[OlCo93] S. Olcoz and J.M. Colom. Toward a Formal Semantics of IEEE Std. VHDL 1076. In *EURO-VHDL/EURO-DAC'93*, Hamburg, Sept. 1993.

[OlCo94] S. Olcoz and J.M. Colom. The Discrete Event Simulation Semantics of VHDL. In *Int. Conf. on Simulation and HDLs*, Tempe, Arizona, SCS, Jan. 1994.

[ReEd94] S. Read and M. Edwards. A Formal Semantics of VHDL in Booyer-Moore Logic. In *CEEDA'94*, Poole, UK, April 7-8, SCSI, San Diego, 1994.

[SaBo93] A. Salem and D. Borrione. Formal Semantics of VHDL timing constructs. In *VHDL for simulation, synthesis, and formal proof*, J. Mermet (ed.). Kluwer, London, 1993.

[Sal92] A. Salem. Verification formelle des circuits digitaux decrits en VHDL. PhD Thesis, Universite Joseph Fourier, Grenoble, October 1992.

[VaTa93] J. P. Van Tassel. Femto-VHDL: The Semantics of a Subset of VHDL and its Embedding in the HOL Proof Assistant. PhD Thesis, University of Cambridge, July 1993.

[VHDL93] IEEE Standard VHDL Language Reference Manual. IEEE Std 1076-1993. The Institute of Electrical and Electronics Engineers, New York, USA, 1994.

[Wilsey90] P.A. Wilsey. Developing a Formal Semantics Description of VHDL. In *Proc. of the 1st European Conf. on VHDL*, Sept. 5-7, IMT, Marseille, 1990.