A Formal Semantics for Verilog-VHDL Simulation Interoperability by Abstract State Machine

Hisashi Sasaki, Toshiba Corp., Yokohama Japan hisashi3.sasaki@toshiba.co.jp

Abstract

A formal semantic analysis for Verilog-HDL and VHDL is provided in order to give the simulation model especially focusing on signal scheduling and timing control mechanism. Our semantics is faithful to LRM and is expected to become a coherent first step for a future semantic interoperability analysis on multi-semantic-domain such as Verilog-AMS and VHDL-AMS. By ignoring the differences of the two simulation cycles, we can use the common semantic functions and the common simulation cycle.

1. Introduction

Many formal approaches have been investigated for VHDL in aiming to give the strict meaning to help understandings (see the references in [4]). Most of these works (except a few trials [5,20,21]) were considered just as theoretical frameworks because they treat only subsets of VHDL and they are not faithful to the LRM (Language Reference Manual). The Boerger 's work [5] is one of the successful and faithful analysis for almost full VHDL by ASM (Abstract State Machine), and its extended works give practical contributions [7,8,9] to the language validation in VHDL-AMS. Now the practical importance for standardization is strongly recognized. For instance, SLDL road map addresses the formal semantics as their major requirement [17].

On the other hands, there are relatively a few reports for Verilog-HDL. VHDL Designer's Reference [1] was the first comprehensive report to compare VHDL and Verilog-HDL, but it was a descriptive explanation not based on a formal method. Gordon [14] appealed the necessity of formal analysis. But in that time, he didn't give a formal semantics yet. It was the first success that Borrione [15] established the formal model under the restriction of synchronous designs. But, from the view of analysis of LRM, their work doesn't treat simulation aspect. To overcome this, our paper will establish the formal simulation semantics for Verilog-HDL with keeping the semantic interoperability with VHDL. Recent successive investigations of interoperability were done in [10,18,19], but they remain still in non-formal approach. Only one axiomatic description of Verilog semantics by Fiskio-Lasseter [22] is found as a potentially competitive to this work. But it has no attention to the interoperability issue.

There is an essential problem when we try to discuss the semantics on Verilog-HDL and VHDL. As both are based on the concepts on the general logic simulation, they share many fundamental concepts. But in general, two LRM may use different terms for the identical concepts in some cases. To treat semantics accurately, we must prepare two distinct simulation cycles according to each LRM, and then we should reconfirm the homomorphism between the two. This ideal approach is not adopted here because they have the differences. The queuing of event for two non-blocking assignments in the same block is different. (This difference frequently causes the misconception [16].) In addition, the stratification of simulation cycle of Verilog is different from the VHDL formulation. By ignoring these differences, we can exploit the fact that the most parts of two simulation cycles are same.

We will here extend the ASM approach [5] for a common semantics of both Verilog-HDL and VHDL. It is based on the semantics functions, the semantic predicates and the simulation cycle used in [5] as “the general semantics” of logic simulator by ignoring the above differences of the two simulation cycles. There are five reasons to adopt such an approach. (1) The above mentioned ideal homomorphism approach seems cumbersome. It requires the two independent formal semantics to wrestle with. (2) Our common simulation cycle is enough to explicitly and simply illustrates the difference and equivalence between the two. (3) Because our next target is the semantic comparison between their extensions, Verilog-AMS [3] and VHDL-AMS [2], the single common base is desirable. (4) ASM has already developed semantic models for many languages including the seeds for forthcoming SLDL such as SDL, C++, Java etc [13,23]. Our common semantics is expected be the first step to discuss the issue in multi-semantic domain environment such as the consistent and coherent semantic analysis for SLDL. (5) ASM is easy to learn without any specific knowledge of difficult formalism.

In this paper, we will formulate the semantic rules by considering on the following. (1) an explicit illustration of delay mechanism (inertial delay, modified transport delay, pure transport delay). (2) a signal waveform, which is composed by a single pair of value and current time. (3) a separation of thread of control (or a suspension of user-defined process) which triggers simulation kernel. It characterizes the non-blocking and blocking statements. (4) an override property of signal driver for continuous assignment and de-assignment statements. (5) a generation of user-defined process by fork-join statement. But we will not want to treat the data type issues and the signal resolution mechanism here. VHDL assumes packages should be responsible to implementation details how to resolve conflicted signals, while Verilog-HDL includes the resolution mechanism in advance as a predefined environment.

This paper is organized as follows: At first, we will review the basic notations to prepare formal discussion based on [5]. Next, the semantic rules for both Verilog and VHDL will be given in order to compare them. Finally we shall conclude this paper.

2. Basic Notations

We describe here the minimum amount of non-trivial *predicates* and *functions* as general semantic objects of logic simulator by ASM. See [5] for more details.

- *driver(P, S):* the driver of the signal *S* in the process *P* , which will be processed by the simulation kernel.
- *active(d)*: the boolean predicate to indicate the active status of driver *d*
- *Waveform*: the list of value-time pair to denote the scheduled signal as waveform.
- $d\ |_{<}\ t$: the function $|_{<}$ yields the driver containing transactions in driver *d* which have time component is less than time *t*
- ^ : the concatenation of sub-list
- < >: an empty list
- *value(E)* : the value of expression *E*
- *first(L), last(L)*: first/last element of list *L*
- *suspended(P)*: the boolean predicate to indicate the suspended status of process *P*
- T_C*:* the current time of simulation
- *reject:* the function of pulse rejection to implement inertial delay.
- *timeout(P)*: the time when the process *P* expected to be resumed.
- *waiting(S)* : the set of processes which are sensitive to the signal value change of *S*
- *waitcond(P)* : the current wait condition of process *P*

3. Formal Model

The simulation cycle of our common model is identical to the one found in [5]. We ignore the two differences: (1) the queuing order for two non-blocking assignments in the same block, and (2) the stratification of Verilog simulation cycle (five regions of the simulation queue).

3.1 Processing Statements

In order to concentrate on the essential behavioral semantics of VHDL'93, we assume that the control flow of each sequential iterative process is determined by the environment, which provides the dynamic changes of values for the external function *program_counter*. The *program_counter* of each process is initialized by pointing to the first statement of that process. After having processed the last statement it returns to the first statement again. Then we formulate this as follows:

```
if Process does < statement >
then  <execution by abstract state machine>.
```

3.2 Signal Assignments

We recall the rules for VHDL [5].

- **P2: TRANSPORT DELAY (non-blocking assignment)**

```
if Process does
< S <= transport Expr_1 after Time_1, ... , Expr_n after Time_n >
then if $Time_1 = 0$
    then driver(Process, S) := Waveform
            active(driver(Process, S)) := true
    else driver(Process, S) :=
                (driver(Process, S)$|_{<}$ ($T_C$ + $Time_1$)^Waveform
where  Waveform = <($X_1$, $Time_1$'), ... , ($X_n$, $Time_n$')>
 ∧ $Time_j' = Time_j + T_C$ ∧  $X_j$ = value($Expr_j$)
```

- **P3: INERTIAL DELAY (non-blocking assignment)**

```
if Process does
< S  <= inertial Expr_1 after Time_1, ... Expr_n after Time_n >
then
   if $Time_1 = 0$
    then driver(Process, S) := Waveform
            active(driver(Process, S)) := true
    else driver(Process, S) :=
            first(driver(Process, S)) ^reject(driver', $X_1$)^Waveform
where Waveform = < ($X_1$, $Time_1$'), ... , ($X_n$, $Time_n$') >
 ∧  $Time_i' = Time_i + T_C$ ∧  $X_i$ = value($Expr_i$)
 ∧  driver' = tail(driver(Process, S) $|_{<}$ ($Time_1$ + $T_C$ ) )
reject(TransList, Val) ≡
 if TransList = <>  ∨ value(last(TransList)) ≠ Val
 then return <>
 else
   return reject(front(TransList, Val))^last(TransList)
```

Now, we shall provide the semantic rules for various assignment statements in Verilog-HDL. For procedural assignments, delay mechanism is transport delay. For continuous assignment, delay mechanism is inertial delay. Note that all rules use the waveform *SingleWaveform* instead of *Waveform*. In the rule VP1, blocking behavior is implemented by the immediate transfer of control thread to simulation kernel: the execution of *suspended(Process)* := **true.** Accurately speaking, our model is almost faithful to the LRM because #0 blocking assignment should be processed as an inactive event. To get full faithfulness, simulation kernel must be updated (but fortunately our rule has effectively equivalent behavior in almost cases without the update). Additionally the (modified) transport delay suggests that *S* is a register.

- **VP1: blocking assignment**

```
if Process does < S  = # Time Expr >
then
   if Time = 0
    then driver(Process, S) := SingleWaveform
           active(driver(Process, S)) := true
           timeout(Process) :=  $T_C$
           suspended(Process) := true
    else  driver(Process, S) :=
          (driver(Process, S)$|_<$ Time')^SingleWaveform
           timeout(Process) := Time + $T_C$
           suspended(Process) := true
  where SingleWaveform = < (X, Time') >
     $\wedge$ Time' = Time + $T_C$ $\wedge$ X = value(Expr)
     $\wedge$ driver' =tail(driver(V-Process, S) $|_<$ (Time + $T_C$ ) )
```

The rule VP1 explicitly specifies the suspension of process by the assignment *suspended(Process)* := **true**, but the rule VP2 has no such suspension. This reflects the non-blocking property. We don't need the Verilog stratification of event queue because this predicates controls the event: instead of the stratification, kernel invocation gives the processing order of queue indirectly.

- **VP2: non-blocking assignment**

```
if Process does < S  <= # Time Expr >
then
   if Time = 0
    then driver(Process, S) := SingleWaveform
          active(driver(Process, S)) := true
    else driver(Process, S) :=
         (driver(Process, S)$|_<$ Time')^SingleWaveform
  where SingleWaveform = < (X, Time') >
        $\wedge$ Time' = Time + $T_C$ $\wedge$ X = value(Expr)
```

Since Verilog-HDL has only the concept of the concurrent behavior for continuous assignments, we try to virtually define a sequential concept for continuous assignments, and then derive the actual semantic rule based on it. The virtual sequential concept adopts inertial delay mechanism, therefore, the rule VP3 uses the *reject* function of P3, which suggests *S* is a net.

- **VP3: (sequential) continuous assignment**

```
if Process does < assign S  = # Time Expr >
then
   if Time = 0
    then driver(Process, S) := SingleWaveform
           active(driver(Process, S)) := true
    else driver(Process, S) :=
           first(driver(Process ,S))^reject(driver', $X_1$)^SingleWaveform
where
     SingleWaveform = < (X, Time') >
 $\wedge$ Time' = Time + $T_C$ $\wedge$ X = value(Expr)
 $\wedge$ driver' = tail(driver(Process, S) $|_<$ (Time + $T_C$ ) )
```

Then the actual continuous assignment is finally given by the following equivalent process:

```
     always @( <list of signals in Expr>) begin
     <sequential continuous assignment>
     end
```

There is no counterpart of procedural continuous assignment and its corresponding de-assignment statement in VHDL. To re-load the previous driver for future **de-assign** statement, *saved_driver(Process, S)* is introduced. The priority over assignments is implemented by the re-load. The procedure *remove_all* removes all the values of the drivers *driver(Process, S)* to be resolved and reset it empty. Thus the pure transport delay is implemented. (The rule VP2 specifies the modified transport delay by the signal resolution mechanism.) The force and release procedural statements can be formulated by the similar way.

- **VP4: procedural continuous assignment**

```
if Process does < assign # Time S = Expr >
then
    begin
    saved_driver(Process, S) := driver(Process, S)
    remove_all(driver(Process, S))
    driver(Process, S) :=
           first(saved_driver(Process, S))^SingleWaveform
     active(driver(Process, S)) := true
    end
where SingleWaveform = < (X, Time') >
 $\wedge$ Time' = Time + $T_C$ $\wedge$ X = value(Expr)
```

- **VP5: procedural continuous de-assignment**

```
if Process does < deassign  S  >
then
     driver(Process, S) := saved_driver(Process, S)
     active(driver(Process, S)) := true
```

A disconnect statement is the signal assignment of null waveform which represents the special driving element of "absence". It is similar to the procedural continuous assignment above but it has no reconstruction functionality of the previous value for the driver.

- **P10: disconnect**

```
if Process does < disconnect S after Time >
then begin
saved_driver(Process, S) := driver(Process, S)
remove_all(driver(Process, S))
if Time = 0
   then driver(Process, S) := NullWaveform
           active(driver(Process, S)) := true
   else driver(Process, S) :=
           saved_driver(Process, S)^NullWaveform
end
 where NullWaveform = < (null, Time') >
        ∧ Time' = Time + $T_c$
```

Finally, we note that intra-assignments including @ timing control can be treated by the interpretation of equivalent process described in the LRM p.118. (see VP13.)

3.3 Procedural Timing Controls

At first we recall the rules for VHDL [5].

- **P4: WAIT FOR**

```
if Process does < wait for Time >
then timeout(Process) := Time + $T_c$
       suspended(Process) := true
```

- **P5: WAIT ON**

```
if Process does < wait on Signals >
then
    suspended(Process) := true
    if S ∈ Signals
    then waiting(S) := waiting(S) ∪ { Process }
```

- **P6: WAIT UNTIL**

```
if Process does < wait until Expr >
then
    waitcond(Process) := Expr
    suspended(Process) := true
    if S ∈ condsignals(Expr)
    then waiting(S) := waiting(S) ∪ { Process }
```

- **P7: WAIT FOREVER**

```
if Process does < wait >
then suspended(Process) := true
```

Now, we shall describe the semantic rules for Verilog-HDL. Denote *E* as a event expression in Verilog-HDL. Also we intentionally overload an implicit signal semantics for *E* to help the coherent description with VHDL.

- **VP6: # Time (Delay Specification)**

```
if Process does < # Time >
then timeout(Process) := Time + $T_c$
       suspended(Process) := true
```

- **VP7: Triggering event S**

```
if Process does < - > (E) >
then
    driver(Process, E) := <not E, $T_c$ >
    active(driver(Process, E)) := true
```

- **VP8: @(*E*)**

```
if Process does < @(E) >
then
    waitcond(Process) := Expr
    suspended(Process) := true
    if S ∈ condsignals(Expr)
    waiting(S) := waiting(S) ∪ { Process }
```

- **VP9: @(posedge E)**

```
if Process does < @(posedge E) >
then
    waitcond(Process) := Expr
    suspended(Process) := true
    if S ∈ condsignals(Expr)
    then waiting(S) := waiting(S) ∪ { Process }
 where Expr = positive_edge_expr(E)
```

The function *positive_edge_expr* gives the expression *Expr* which is the conditional *S='1'* to detect a positive edge of *E*. The rule VP10 for negative edge is similarly defined.

- **VP10: @(negedge E)**

```
if Process does < @(posedge E) >
then
    waitcond(Process) := Expr
    suspended(Process) := true
    if S ∈ condsignals(Expr)
    then waiting(S) := waiting(S) ∪ { Process }
 where Expr = negative_edge_expr(E)
```

- **VP11: Wait(*Expr*)**

```
if Process does < wait(Expr) >
then
    if Value(Expr) = false
    then
        waitcond(Process) := Expr
        suspended(Process) := true
        if S ∈ condsignals(Expr)
        then waiting(S) := waiting(S) ∪ { Process }
```

The *Expr* is the expression to represent the wait conditional. Note that in Verilog-HDL, *Process* is not suspended when *Value(Expr)* = **true**. In the case of VHDL, the wait statement always causes the suspension of the process because the rules P4-P7 don't contain such a test of *Value(Expr)*. This difference of operation promotes the accurate understanding for the difference in concept in such an obvious way.

3.4 Verilog-HDL Specific Statements

A fork-join statement and an intra-assignment with @ timing control are Verilog-HDL specific functionality.

- **VP12: Fork & join**

```
if Process does < fork Statement$_1$, ... join) >
then begin
        create subProcess(Process) ;
        choose $P_i$ : subProcess(Process)
               $P_i$ does < Statement$_i$ >
         end choose ;
        destroy subProcess(Process) ;
        end
```

The fork-join construct generates a new thread of control, so we introduce the creation and destruction of processes. Fork functionality is represented by the **choose** construct of ASM.

- **VP13: S <= @ Event Expr**

```
if Process does < S <= @ Event Expr >
then begin
        create subProcess(Process) ;
        execute subProcess(Process) ;
        destroy subProcess(Process) ;
        end
```

The dynamic creation, execution and destruction are performed by generating an instantaneous process. (Note that **wait on** Event or @(Event) suspends the execution until the Event occurs.) Such a process *subProcess(Process)* is defined by the following equivalent process:

```
(VHDL)                  (Verilog-HDL)
process
begin                       begin
    temp = Expr;               temp = Expr ;
    wait on Event ;            @(Event)
    S <= temp ;                S <= temp ;
    wait ;                    end
end ;
```

4. Conclusions and Further Studies

We have provided the first faithful and common formal semantics for Verilog-HDL and VHDL in focusing on signal scheduling and timing control statements. Our model helps the accurate understanding on the semantic interoperability not only for IC designers but CAD engineers, especially for standardization contributors. It also reduces the learning time by avoiding us from wrestling with two distinct formal model. Finally our result also suggests that the further improvement of deeper semantic interoperability could be attainable at the intermediate level language [11]. We would like to extend this work for their AMS-extensions.

5. References

[1] J-M. Berge et.al, chapter 9, Verilog and VHDL, in VHDL Designer's Reference, pp.231-317, Kluwer Academic Publishers, 1992.

[2] IEEE 1076.1 Working Group, Definition of Analog and Mixed Signal Extensions to IEEE Standard VHDL, (Integrated Draft of VHDL-AMS) April 17, 1998. http://www.vhdl.org/analog/wwwpages/documentation.html

[3] OVI, Verilog-AMS Language Reference Manual, Analog & Mixed-Signal Extensions to Verilog HDL, Version 1.1.1, March 20, 1998. (for evaluation) http://www.ovi.org/VA-TSC/index.htm

[4] Carlos D. Kloos, et al., Formal Semantics for VHDL, 1995, Kluwer Academic Publishers.

[5] Egon Boerger et al., A Formal Definition of an Abstract VHDL'93 Simulator by EA-Machines, in Formal Semantics for VHDL, pp.107-139, Kluwer Academic Publishers, 1995. http://www.eecs.umich.edu/gasm/hardware.htm#vhdl

[6] Natividad Martinez Madrid, et al., A semantic model for VHDL-AMS, CHARME'97, October 1997.

[7] Hisashi Sasaki, et al., Semantic Validation of VHDL-AMS by Abstract State Machine, pp.61-68, IEEE/VIUF BMAS97, October, 1997. http://katayama-www.cs.titech.ac.jp/~sasaki/vhdl_ams/

[8] Takeshi Sasaki, et al., Semantic Analysis of VHDL-AMS by Attribute Grammar, FDL'98. September 1998.

[9] Tom Kazmierski, A formal description of VHDL-AMS analogue systems, DATE'98, pp. 916-920.

[10] Victor Berman, Standard Verilog-VHDL Interoperability, pp.2-9, OVI/IEEE 3rd. Int. Verilog HDL Conference, 1994.

[11] John Willis (Editor), AIRE / CE: Advanced Intermediate Representation with Extensibility / Common Environment, IIR Specification Version4.4 (Trial Implementation Draft of 12/15/97) Including Digital VHDL & VHDL-AMS Support.

[12] IEEE Standard Hardware Description Language Based on the Verilog Hardware description Language, IEEE Std 1364-1995. 14 October 1996.

[13] Uwe Glaesser et al., Abstract State Machine Semantics of SDL, Journal of Universal Computer Science, vo.3, no.12, 1997, pp.1382-1414. http://www.eecs.umich.edu/gasm/proglang.html#sdl

[14] Mike Gordon, The Semantic Challenge of Verilog-HDL, LICS'95, 1995. http://www.cl.cam.ac.uk/users/mjcg/

[15] Dominique. Borrione, et al., An approach to Verilog-VHDL interoperability for synchronous designs, CHARME'97. October 1997. http://www-tima-vds.imag.fr/Publications/Charme97.ps

[16] Clifford E. Cummings, Verilog Non-blocking Assignments Demystified, IVC/VIUF March, 1998.

[17] David L. Barton, System Level Design Section of the Industry Standards Roadmap, July 8, 1998. http://www.inmet.com/SLDL/sysroad/index.html

[18] John Willis et al., Verilog: Dialect of VHDL ?, VIUF Spring 1996, pp.239-244.

[19] Douglas J. Smith, VHDL & Verilog Compared & Contrasted – Plus Modeled Example Written in VHDL, Verilog and C, DAC'96.

[20] Serafin Olcoz, A Formal Model of VHDL Using Colored Petri Nets, in [4]

[21] Ralf Reetz et al., A Flow Graph Semantics of VHDL: A Basis for Hardware Verification with VHDL, in [4].

[22] John Fiskio-Lasseter, A Formal Description of Behavioral Verilog Based on Axiomatic Semantics, Tech. Rep. TR-98-04, Univ. of Oregon, 25 August, 1998. http://www.cs.uoregon.edu/~johnfl/thesis/

[23] http://www.eecs.umich.edu/gasm/papers.html