Obtaining Functionally Equivalent Simulations Using VHDL and a Time-shift Transformation

Frank Vahid

Technical Report #91-33
April 2, 1991
Revised October 8, 1991

Dept. of Information and Computer Science
University of California, Irvine
Irvine, CA 92717
(714) 856-7063

vahid@ics.uci.edu

Abstract

The advent of VHDL has brought about a number of VHDL simulators. Many translation schemes from domain specific languages to supposedly functionally equivalent VHDL have been developed as an approach to obtaining simulations. However, functionally equivalent VHDL can not be created for the general case, due to a theoretical limitation to this approach. It is a very subtle point and has thus been overlooked until now, but it is extremely important since it can cause incorrect simulation, therefore making translations to VHDL an unsound simulation technique. In this paper, we introduce this fundamental limitation. In addition, we propose an alternative approach which strives for functionally equivalent **simulation** *rather than functionally equivalent VHDL, while still taking advantage of VHDL simulators. Our method uses a novel time-shift transformation, also introduced in this paper, in conjunction with almost any translation scheme. The method makes correct simulations easily obtainable, thus bridging the gap to a truly sound and highly advantageous use of VHDL as a tool for simulating domain specific languages.*

Contents

List of Figures

1 Introduction

The adoption of the VHSIC Hardware Description Language (VHDL) as an IEEE standard [IEEE88] has given rise to many VHDL tools, such as simulators and design synthesizers. However, other languages are still being introduced [Ha87, JePA91, DuHG90, VaNG91], since no single language is ideal for all possible domains. For example, many systems are naturally described as a hierarchy of state transition diagrams, which might be tedious to manually describe in VHDL. We refer to these other languages as being *domain specific*. Translating such languages to VHDL can yield enormous advantages, such as the use of existing VHDL simulators. Advantages over writing a new simulator include: (1) *much* less implementation time to obtain a simulation capability, (2) more reliable simulations, since VHDL is a standard and thus its simulators are widely used, and (3) faster simulations, since VHDL simulator manufacturers concentrate on simulation efficiency. Many such translation schemes have thus appeared [JePA91, ArWC90, DuCH91, MaWa90, NaVG91, TiLK90].

However, the goal of obtaining completely correct simulations is unattainable in many cases, since it is not possible to create VHDL that is functionally equivalent to the domain specific language's description. The basic problem involves trying to execute behavior related to control (e.g. changing states of a state transition diagram) in VHDL's delta time, which then interferes with delta time behavior for non-control behavior. The current translation schemes only work for a subset of descriptions, and are thus more of a quick and dirty simulation solution than a solid technique.

This does not mean that these domain specific languages can not benefit from the advantages of VHDL simulators. We distinguish between (1) obtaining functionally equivalent *VHDL*, and (2) obtaining functionally equivalent *simulations*. The former creates a VHDL file that can be treated as any other VHDL file. The latter might use VHDL only as an intermediate representation; a modeler sees only the domain specific language and simulation output. Though the VHDL model might not be functionally equivalent, the VHDL simulation output can be transformed to correct output (see Figure 1).

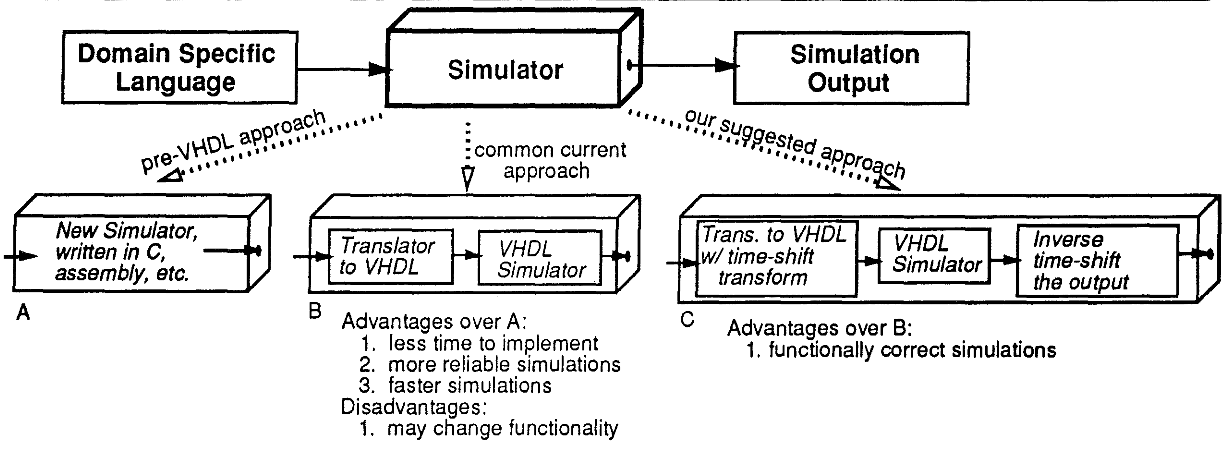
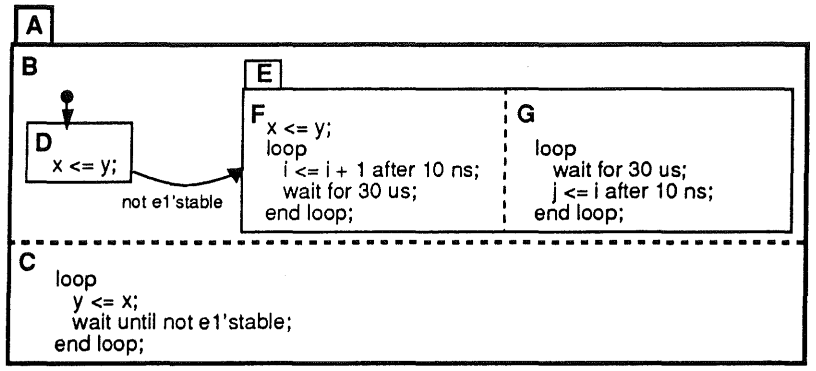


Figure 1: Various approaches for simulating domain specific languages

This paper introduces the fundamental problem that prevents translation to functionally equivalent VHDL. We then introduce a technique that shifts time in the original language, translates to VHDL, simulates, and then time-shifts the results back. This achieves functionally equivalent simulation, eliminating the final obstacle that has prevented implementations of domain specific languages from benefitting from VHDL simulators in a sound manner.

2 The Barrier to Obtaining Functionally Equivalent VHDL

For simplicity, we will focus on translating languages based on some variation of StateCharts [Ha87]. The problem introduced below generalizes to many other languages. Briefly, StateCharts provides for specification as a hierarchy of concurrent and sequential states. Most languages based on StateCharts have added activities that are performed in a given state [MaWa90, JePA91, DuHG90, VaNG91]. Figure 2 gives a simple example of these types of languages. Figure 3 shows three translation schemes that we consider; for simplicity, only the control for activating processes is shown (i.e. deactivation and other details are omitted).



When in A, we are simultaneously in both B and C. When in B, we are initially in D. If e1 changes, we are in E, which means we are in F and G simultaneously. When in a state that has activities, we execute those activities. If we reach the end of the statements of an activity, that activity is idle.

Things to note:

1. When first in A, the values of x and y should be swapped (by C and D)
2. When in E, j should always have the value of i but with a 30 us phase delay
3. When in D and C and e1 changes, x and y should again be swapped (by C and F)

Figure 2: An example language based on a derivation of StateCharts

In [ArWC90], a scheme is presented for translating StateCharts to VHDL. Each state is represented as a procedure, and substates are executed by calling each substate's procedure. A procedural model is a sequential model; thus concurrent items in a StateChart become sequential in the VHDL (Figure 3a). Many activities will produce quite different results when executed sequentially rather than concurrently, such as those associated with states **F** and **G**. Thus we do not consider the procedural model further.

Developers of other schemes have focused on translating to a concurrent model of VHDL, such as the process model [MaWa90, JePA91, NaVG91, DuCH91] which consists of a set of concurrent processes, each either active or suspended. Generally each StateChart state is translated to a VHDL process. Activities associated with a state are translated to VHDL sequential statements in that state's process.

Most of these schemes maintain a hierarchical activation scheme in the VHDL (see Figure 3b); that is, some processes are created only to activate other processes, just as some StateChart states exist only to be composed into substates (such as state **B**). These processes are now referred to as control processes. This scheme causes a subtle but important change in functionality. In Figure 2, when state **A** is entered, it means both **B** and **C** are entered. When **B** is entered, it means **D** is entered. Thus note that the statement $x <= y$ in **D** and $y <= x$ in **C** should be executed simultaneously, so that the values of x and y are swapped. However, this will not happen in this scheme. To understand why not, we must first understand VHDL delta timing.

VHDL is based on a continuous repetition of a simulation cycle. Briefly, each cycle consists

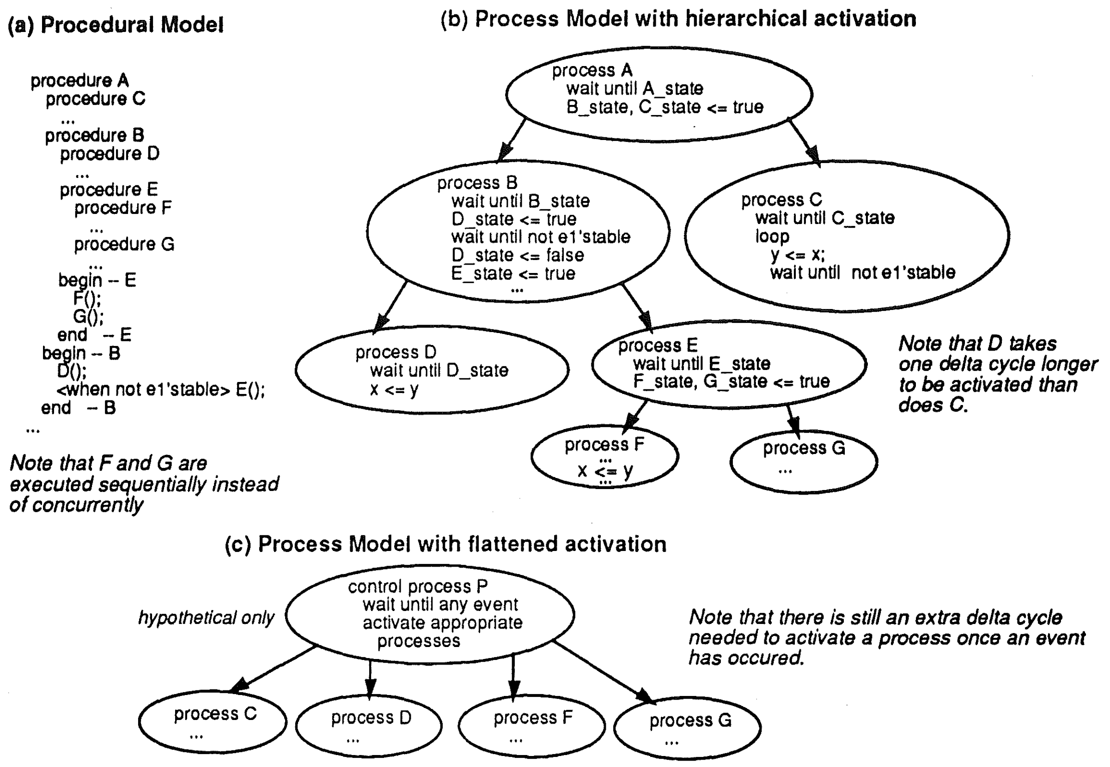


Figure 3: Various translation schemes considered

of (1) advancing time to the next 'interesting' point, (2) updating signals that should change at this time, and (3) executing activated processes until they suspend (e.g. reach a wait statement). If a signal such as x is assigned to, an infinitely small delay time, called a delta delay, is implicit (unless an *after* clause states an explicit delay) (see Figure 4). Thus 'advancing time' might advance by some number of real time units (e.g. seconds) or by one delta time unit. A common misunderstanding is that a delta unit is the smallest real-time unit supported by the language, such as femtoseconds. This is incorrect; delta-units are on a separate scale from real-time units.

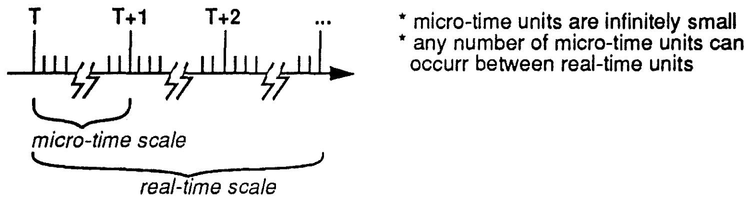

T
T+1
T+2
...
* micro-time units are infinitely small
* any number of micro-time units can occurr between real-time units
micro-time scale
real-time scale

Figure 4: Two time scales found in many languages, including VHDL

Figure 5 shows values for x and y at each delta point for the example of Figure 3b, where we assume A_state was set to true at time 50 ns. Note that since **D** is one level deeper in the hierarchy than is **C**, there is one extra process which must be activated and executed, and thus one extra delta time unit to execute $x <= y$ than for $y <= x$. This causes incorrect simulation results. This problem can occur whenever the 'activation tree' (see Figure 3b) is unbalanced.

To solve this, consider a hypothetical scheme which flattens all activation into a single control process (Figure 3c). It waits for any event, calculates which processes to activate, and then sets signals which activate those processes. Note that such a process would likely be extremely complex (as also noted in [MaWa90]), and there is no published scheme which does

time	A_state	B_state	C_state	D_state	x	y	Description of simulation cycle
50 ns	true	false	false	false	6	7	update A_state, execute process A; scheduled: B_state and C_state to get true
delta 1	true	true	true	false	6	7	update B_state and C_state, execute processes B and C; scheduled: D_state to get true and y to get 6
delta 2	true	true	true	true	6	6	update D_state and y; execute process D; scheduled: x to get 6
delta 3	true	true	true	true	6	6	update x

Figure 5: Values for each delta point in hierarchical activation scheme, showing that swap fails

this. However it is useful to consider, since it represents the ideal scheme with respect to the above problem, because we would never have an imbalance in the number of activation levels. A second problem still exists.

time	C_state	F_state	x	y	Description of simulation cycle
100 ns	true	false	7	6	e1 gets new value, execute processes P and C; scheduled: F_state to get true, y to get 7;
delta 1	true	true	7	7	update F_state and y, execute process F; scheduled: x to get 7
delta 2	true	true	7	7	update x;

Figure 6: Values for each delta point in flattened activation scheme, showing that swap fails

Consider the case where we are in states **D** and **C**, and the swap has already been performed. Thus no computations are being performed and we are waiting for e1 to change (so that the arc from **D** will be traversed and the wait statement in **C** will terminate). When e1 changes, **C** should execute $y <= x$ (it loops back to this statement). At the same time, we should enter **E** and thus execute $x <= y$. This implements another swap of x and y. Figure 6 shows the values of x and y for several delta points assuming that e1 changes at time 100 ns. Once again, the swap failed. The point to notice is that the statement *wait until event* followed by an action is identical, in the semantics of the StateCharts based language, to an arc labeled with *event* pointing to a state with the same action. However, the generated VHDL requires one extra delta for the latter, due to the fact that after the event occurs, the control process must still activate the appropriate state process (one more delta) before the actual action can be executed. This is an example of what we refer to as a *control computation*: any computation performed solely for obtaining correct simulation.

We can now understand the general problem: *the control computations should be performed in zero time in the generated VHDL, but instead require at least one delta.* Micro-time computations in the domain specific language are also translated to delta-time and may be affected by this extra delta. Note that this problem generalizes to any domain specific language which uses a micro-time scale and whose simulation control requires the use of delta-time when translated to VHDL.

One possible solution might consider the fact that VHDL variables permit zero time computation. The flattened activation scheme already assumed that variables were used within the control process, but it could not use variables to communicate with the other processes since variables are not defined outside of a process. This is due to variables not being defined over time; thus they cannot be shared over time, so signals must be used for interprocess communication. One might replicate the control process (or its relevant parts) within *each* of the other processes, thus eliminating the need for a separate control process and for interprocess control communication.

We have strayed quite far from the simple hierarchical activation scheme. This has made the VHDL code extremely complex, since an inherently hierarchical and concurrent model is being forced into a flattened and sequential one. Intuitively it would seem that this solution approach will still create problems, since it aims only to solve the specific problem of activating processes without an extra control delta, rather than the more general problem of performing *any* control computation without an extra delta. This is indeed the case. For example, some domain-specific languages permit declarations to be associated with an activity. Thus *signal i : integer := 3* could have been declared with F's activity. The semantics of this is that every time **F** is entered, i is re-initialized to 3. This requires one delta since i is a signal. The above solution does not handle this. There are *numerous* such cases where control must use one or more deltas [NaVa90].

To summarize, we have shown how translation schemes currently perform control computations in VHDL's delta-time, and have shown that this causes incorrect simulation. We demonstrated not only the complexity, but also the futility of trying to have control computations coexist with other delta-time computations without changing the functionality. We can tune the VHDL to solve one specific problem, but it is impossible to solve in the general case. The conclusion is that there is currently no practical way to translate such languages to fully *functionally equivalent VHDL.*

3 The Time-shift Transformation

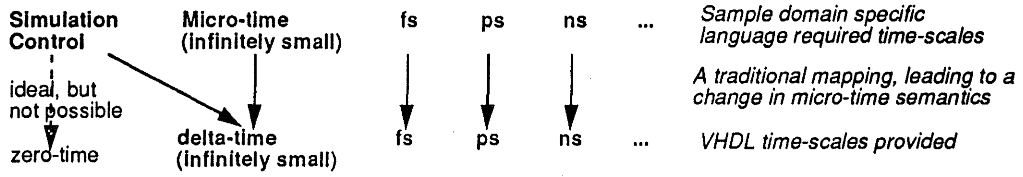

Simulation Control
Micro-time (infinitely small)
fs
ps
ns
...
Sample domain specific language required time-scales
ideal, but not possible
A traditional mapping, leading to a change in micro-time semantics
zero-time
delta-time (infinitely small)
fs
ps
ns
...
VHDL time-scales provided

Figure 7: The current method of mapping control computations to delta-time, causing interference with micro-time functionality

The problem described in the previous section is shown graphically in Figure 7. Several computations needed for control are performed in delta-time in the VHDL, which interferes with the domain-specific language's micro-time functionality, which is also performed in delta-time in the VHDL. Ideally, we would perform these computations in a smaller time scale than delta-time, but VHDL offers no smaller time-unit. By stating the problem in this manner, a simple solution becomes clear: perform the control computations in delta-time in the VHDL, and perform micro-time computations in the next *higher* VHDL time scale. Everything that used this higher time scale must be done in the next higher time scale, and so on. Essentially we are *shifting time to make room at the lower end for the control computations.* We call this a *time-shifted translation* (Figure 8). The simulation output will now represent correct functionality except that the times are incorrect (shifted). A shifting back will solve this, and if the translation scheme was correct, then the resulting output represents a completely functionally equivalent simulation of the domain language.

The time-shifted translation is implemented by applying a *time-shift transformation* to the domain-specific language, and then applying a VHDL translation scheme (Figure 9). The transformation can be applied to any language used to describe activities. We will introduce the transformation assuming that the activities are described using VHDL sequential statements. We do this because the time related statements of other languages can be easily implemented

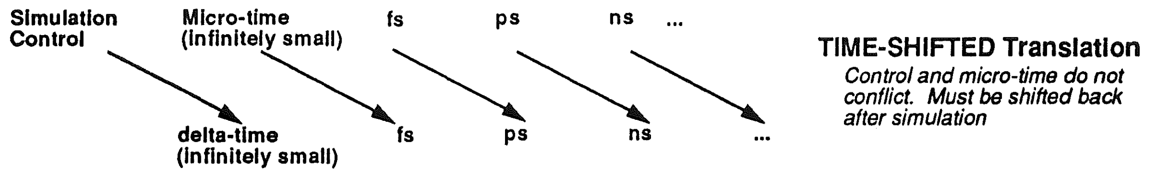

Simulation Control
Micro-time (infinitely small)
fs
ps
ns ...
delta-time (infinitely small)
fs
ps
ns
...
TIME-SHIFTED Translation
Control and micro-time do not conflict. Must be shifted back after simulation

Figure 8: Time-shifted translation, which prevents control computations from interfering with micro-time functionality

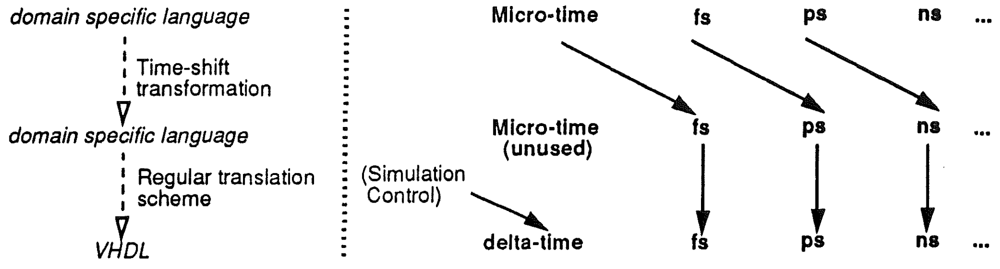


Figure 9: Implementation of time-shifted translation, showing the transformation step followed by the translation step

by VHDL's time related statements. Thus, the transformation is easily modified to account for other languages' statements. Remember that the transformation deals with the ***domain specific language's*** statements.

```
variable t : time := 10 ns;   ──►  variable t : time := 10 us;
t := t + 30 ns;               ──►  t := t + 30 us;
s <= 1 after 10 ns;           ──►  s <= 1 after 10 us;
wait for t + 10 ns;           ──►  wait for t + 10 us;
s <= 1;  ──► (s <= 1 after 1 micro-unit)  ──► s <= 1 after 1 fs;
```

Figure 10: Examples of time-shift applied to domain-specific language's statements

A signal assignment statement sets a signal's value at a specified time. It's relevant form is: *some_signal <= expression <after time_expression>*. If the *after* clause is omitted, *after 0 ns* is implicit; we first make these explicit. We then shift all occurrences of time units in all expressions. Thus *fs* become *ps*, *ps* become *ns*, and so on. This shifts the real-time but not the micro-time scale, since an assignment *after 0 ns* really means after 1 micro-unit. Since the units of 0 are irrelevant, shifting to *0 us* still means 1 micro-unit. The 0 should have been shifted to *1 fs*. We can account for this by using a function *ShiftIfZero(time_expression)*, which returns 1 fs if its parameter is 0, else it returns the parameter:

```
function ShiftIfZero(time_expression : in time) return time is
begin
   if (time_expression = 0 fs) then  -- units of 0 are irrelevant
      return(1 fs);  -- 1 micro-time unit shifted
   else
      return(time_expression);
   end if;
end;
```

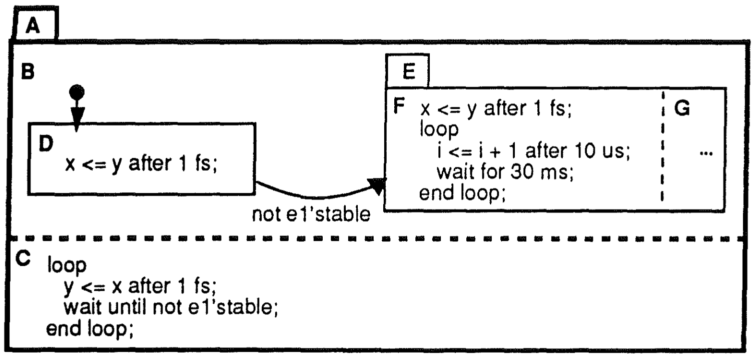
We need a function since the *after* clause might contain an expression (e.g. $t + s$) instead of just a literal 0. This discussion also applies to the *for* time_expression clause of a wait

```
For each signal assignment with no after clause
     create the after clause:    after 0 ns
Replace each occurence of  'fs' with 'ps', of 'ps' with 'ns',
  etc., in all expressions.
For each signal assignment, and each wait statement with a for clause
       replace the statement's time expression (the after or for clause) by:
             ShiftIfZero(time_expression)
```

Figure 11: Time-shift transformation algorithm, applied to a model in the domain-specific language

statement. Specifically, a *wait for 0 ns* is identical to a *wait for 1 micro-unit*; thus we again replace the time expression by *ShiftIfZero(time_expression)*.

Figure 11 summarizes the time-shift transformation. Since the micro-time scale is unused after the transformation, only control computations will be implemented in VHDL's delta time after translation (see Figure 9). The VHDL simulation output will now be correct except that the times at which events occur will be wrong. An inverse time-shift must be performed on this output. Thus *fs* become micro-time units, *ps* become *fs*, etc.



(The ShiftIfZero function is not called in this example since all time expressions are literals. In the general case, it would be called, e.g. i <= i + 1 after ShiftIfZero(10 us))

Figure 12: Earlier example after the time-shift transformation is applied

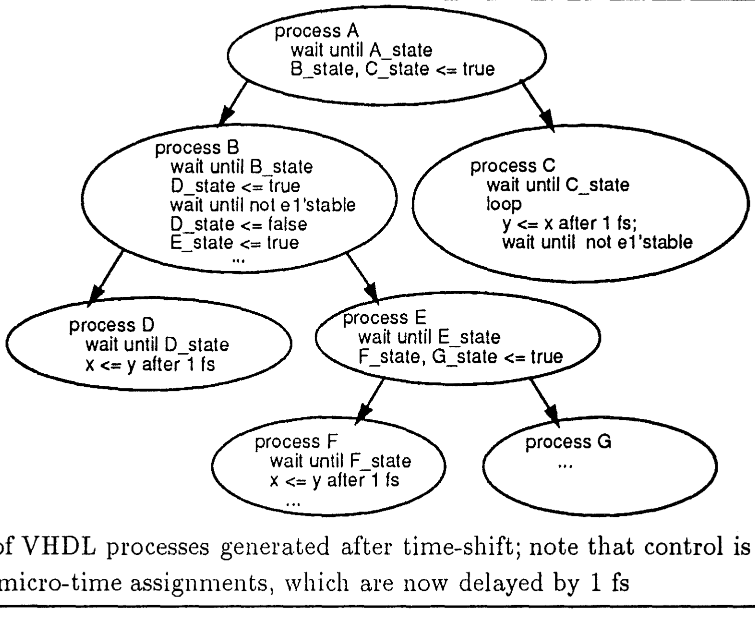


Figure 13: Sample of VHDL processes generated after time-shift; note that control is done in a different time scale than are micro-time assignments, which are now delayed by 1 fs

Figure 12 shows Figure 2 after the time-shift transformation. Figure 13 shows the relevant VHDL generated by the scheme of Figure 3b. Analysis of Figure 13 demonstrates that the time-shift has worked: the swap will be achieved in both of the problem cases given in the previous section. The reason is that control is performed in delta time, whereas the assignments have been shifted to the *fs* time domain, so there is no interference. Figure 14 shows sample simulation results of the generated VHDL. Note that the inverse shift essentially divides the time by 1000, and that any time increments of 1 fs are simply removed, since they are mapped to delta units which traditionally are not shown. Also note that shifting delta-time back to zero time requires no change: since delta-time is not shown, events separated by delta-time and those separated by zero time are indistinguishable; only the order is important. See Figure 1 to review the context in which these transformations are used.

VHDL simulation output	*Comments*	**Inverse shifted simulation output**
	(assume x = 6, y = 7)	
50,000,000,000 fs (50 us)	*Corresponds to time 50 ns without time-shift*	50,000,000 fs (50 ns)
A_state = true	*These are actually each separated by 1 delta, but simulators don't usually show this explicitly.*	A_state = true
B_state = true		B_state = true
C_state = true	*C_state going true activates C's process, which schedules y to get 6 after 1 fs.*	C_state = true
D_state = true	*D_state going true activates D's process, which schedules x to get 7 after 1 fs.*	D_state = true
50,000,000,001 fs		x = 7

x = 7
y = 6
	The swap worked	y = 6

	(assume e1 changes)	
100,000,000,000 fs (100 us)		100,000,000 fs (100 ns)
e1 = 99	*'not e1'stable' evaluates to true, which activates processes B and C. Process C schedules y to get 7 after 1 fs.*	e1 = 99
E_state = true	*Process E is activated.*	E_state = true
F_state = true	*Process F is activated, which schedules x to get 6 after 1 fs.*	F_state = true
100,000,000,001 fs		x = 6

x = 6
y = 7
	The swap worked	y = 7

Figure 14: Sample VHDL simulation of earlier example, with inverse shifted and thus final simulation output

4 Improvements

It should be noted that the time-shift might create VHDL which exceeds a simulator's largest range of time-units (e.g. *femtoseconds* to *seconds* is too large a range). This is easily accounted for by shifting micro-time up to higher unused units. Also note that the number of allowed micro-time steps goes from infinity to a fixed number after shifting. The time-shift is again easily modified to permit a number of steps that would likely never be exceeded (e.g. 1,000,000). Lastly, if a model uses only higher real-time units (e.g. *ns*), those units need not be shifted.

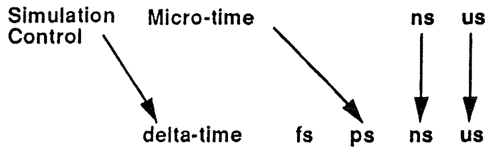


Figure 15: Improvements: higher units need not be shifted; micro-time can be shifted higher to meet time-range constraint

5 Examples

Figure 16 shows examples of several problems that can arise due to the delta-time conflict problem. Figure 16a,b show examples which will simulate incorrectly using a hierarchical activation scheme in the generated VHDL. In the first case, concurrency is affected (the first swap example of this report). In the second, the driver for one state is not shut off before that of another is turned on, causing an overdriven signal error. Figure 16c requires an extra delta for state activation (second swap example of this report). Figure 16d gives an example where an extra delta is needed to initialize a signal upon each entry of a state, causing incorrect results. Figure 16e shows an example in which an extra delta used for handshaking will cause a swap to fail. In the appendix of this report is shown the original specification, the non-shifted VHDL and its simulation results, and the shifted VHDL with its simulation results, for several of these examples.

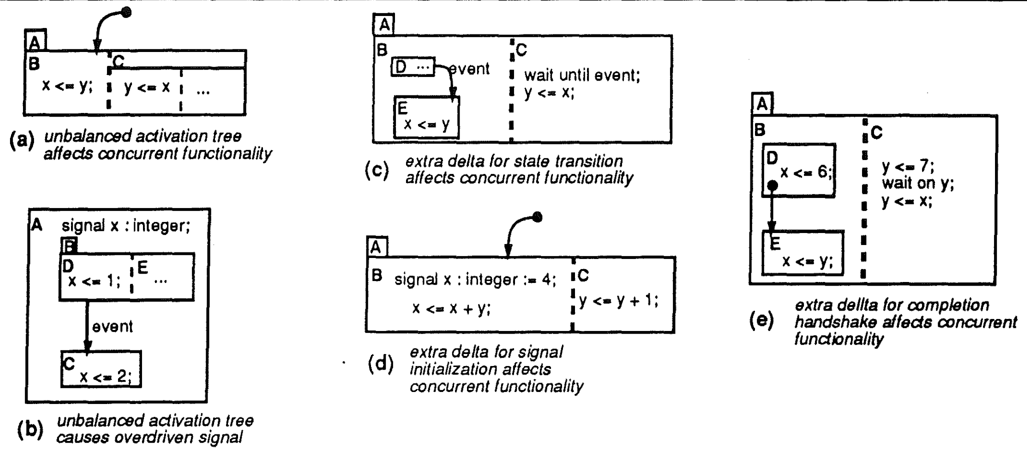

A
B
C
x <= y;
y <= x
...
(a) unbalanced activation tree affects concurrent functionality
A signal x : integer;
B
D
x <= 1;
E
...
event
C
x <= 2;
(b) unbalanced activation tree causes overdriven signal
A
B
D ...
event
C
wait until event;
y <= x;
E
x <= y
(c) extra delta for state transition affects concurrent functionality
A
B signal x : integer := 4;
C
x <= x + y;
y <= y + 1;
(d) extra delta for signal initialization affects concurrent functionality
A
B
C
D
x <= 6;
y <= 7;
wait on y;
y <= x;
E
x <= y;
(e) extra dellta for completion handshake affects concurrent functionality

Figure 16: Examples which cause problems for translation, all solvable using the time-shift

6 Results

The time-shift transformation has been implemented in C for the domain-specific language described in [VaNG91]. A translator from this language to VHDL is also implemented [NaVa90] (we believe this translation scheme to be the most complete and straightforward of any existing scheme for StateCharts derived languages, but the reasons for this are beyond the scope of this paper). Numerous examples were tested which, using the translator only, created VHDL which simulated incorrectly (using a commercial simulator), which would also occur in other schemes [MaWa90, JePA91, DuHG90]. When the time-shift was applied before translation, the results were correct (see Figure 17; the lettered examples correspond to Figure 16). The transformation was also applied to examples that previously simulated correctly. We currently perform the inverse time-shift visually on the simulation results, which is a trivial task (e.g. divide all times by 1000).

7 Conclusion

This paper introduced an until now unnoticed and unsurmountable limitation that prevents translation from certain languages to *functionally equivalent VHDL*. One conclusion is that

Example	Problem	Simulation correct after transform?
(a)	Some processes take longer than others to activate (example in this paper), causing swap to fail	yes
(b)	Unbalanced activation causes overdriven signal	yes
(c)	State transition requires extra delta, causing swap to fail	yes
(d)	Extra delta for initializing signal	yes
(e)	Extra deltas for a control handshake	yes
draco	none	yes
cont_counter	none	yes
processor	none	yes

Figure 17: Examples simulated without and with the time-shift transformation

translating from one HDL to another is perhaps more complex than previously believed, and much attention must be given to preserving semantics. While the time-shift introduced provides for correct simulation, hiding the VHDL can be a very high a price to pay, since other tools such as VHDL debuggers can not then be used without modification to prevent showing the time-shifted VHDL to a user.

Thus several possible outcomes of the translation limitation include: (1) An extension to VHDL itself that eliminates the limitation. This is even more likely when one considers that the limitation is not just a translation issue. It is also a VHDL modeling issue, e.g. how does one write a VHDL model for a system which is conceptualized as a hierarchy of behaviors? The same problems will arise as during translation, since modeling is essentially a translation from a modeler's system conceptualization to VHDL. (2) Domain-specific language developers will use the time-shift in conjunction with VHDL tools as hidden pieces of their own tools. (3) Domain-specific language developers will lower their expectations of the usefulness of VHDL tools. For example, they may write a new simulator rather than trying to use VHDL simulators.

Until any VHDL extensions occur, the time-shift introduced makes possible an advantageous and sound use of VHDL simulators for simulating domain specific languages.

8 Acknowledgements

This work was supported by the Semiconductor Research Corporation (grant #90-DJ-146). We are grateful for their support. We would also like to thank Sanjiv Narayan, Nikil Dutt, Tedd Hadley, and Loganath Ramachandran for their help and suggestions.

9 References

[ArWC90] Arsenault, A., Wong, J.J., Cohen, M., "VHDL Transition from System to Detailed Design", VHDL User's Group Meeting, Boston, April 1990.

[AyWS86] Aylor, J., Waxman, R., and Scarratt, C., "VHDL–Feature Description and Analysis", IEEE Design and Test, April 1986.

[DuCH91] Dutt, N., Cho, J., and Hadley, T., "A User Interface for VHDL Behavioral Modeling," CHDL, April 1991.

[DuHG90] Dutt, N., Hadley, T., and Gajski, D., "An Intermediate Representation for Behavioral Synthesis," DAC, 1990.

[Ha87] Harel, D., "StateCharts : A Visual Formalism for Complex Systems", Science of Computer Programming 8, 1987 pp 231-274.

[IEEE88] IEEE Standard VHDL Language Reference Manual, IEEE, March 1988.

[JePA91] Jerraya, A., Paulin, P., and Agnew, D., "Facilities for controllers modeling and synthesis in VHDL", VHDL Users' Group Conference, April 1991.

[MaWa90] MacDonald, R., and Waxman, R., "Operational Specification of the SINCGARS Radio in VHDL", AFCEA-IEEE Tactical Communications Conference, April 1990.

[NaVa90] Narayan, S. and Vahid, F., "Translating SpecCharts to VHDL", UC Irvine, TR 90-21, July 1990.

[NaVG91] Narayan, S. Vahid, F., and Gajski, D., "Translating System Specifications to VHDL", The European Conference on Design Automation, Amsterdam, February 1991.

[Sh85] Shahdad, M., et al., "VHSIC Hardware Description Language", Computer, February 1985.

[TiLK90] Tikanen T., Leppanen T., and Kivela J., "Structured Analysis and VHDL in Embedded ASIC Design and Verification", EDAC, 1990.

[VaNG91] Vahid, F., Narayan, S., and Gajski, D., "SpecCharts: A Language for System Level Synthesis," CHDL, April 1991.

[Wa86] Waxman, R., "The VHSIC Hardware Description Language – A Glimpse of the Future", IEEE Design and Test, April 1986.

A Appendix

This appendix gives the details of several examples. The domain-specific language used is the StateChart based language called *SpecCharts* [VaNG91]. For each example, a textual dump of the original SpecChart is given. Simulation results for VHDL files generated automatically for non-shifted and shifted examples are then given. The translator has a flag that indicates that a time-shift should be performed, so it is done automatically. A few of the VHDL files are shown, but space does not permit displaying all of them. The time-shifted simulation output should be mentally shifted back by dividing times by 1000. Remember that events separated by 1 fs simply get shifted to the same time (they are actually delta events).

A.1 Swap examples

This is the example of Figure 2, showing the swap problems discussed in the report. The swap problems also correspond to Figure 16a,c. Note from the VHDL outputs that without the time shift the swap fails, but with it, the swap succeeds both times (i.e. x and y change values from 6,7 to 7,6 and back to 6,7).

SpecChart

```
state
{
   name {A}
   declarations
   {
      signal x : integer := 6;
      signal y : integer := 7;
      signal i : integer := 1;
      signal j : integer;
      signal e1 : integer := 99;
   }
   concurrent substates
   {
      B : ;
      C : ;
   }
}

state
{
   name {B}
   sequential substates
   {
      D : (EI, not e1'stable, E);
      E :;
   }
}

state
{
   name {C}
   declarations
   {
      signal cs : integer := 1;
   }
   code
   {
      loop
         y <= x;
         wait until not e1'stable;
      end loop;
   }
}

state
{
   name {D}
   code
   {
      x <= y;
      e1 <= e1 + 1 after 100 fs;
   }
}

state
{
   name {E}
   concurrent substates
   {
      F :;
      G :;
   }
}

state
{
   name {F}
   code
   {
      x <= y;
      loop
         i <= i + 1 after 10 fs;
         wait for 30 ps;
      end loop;
   }
}

state
{
   name {G}
   code
   {
      loop
         wait for 30 ps;
         j <= i after 10 fs;
      end loop;
   }
}
```

Non-shifted VHDL simulation results

Note that X and Y do not get swapped.

```
0 FS
    SMON:     ACTIVE /AE/INA (value = TRUE)
   SMON6:     ACTIVE /AE/INA_INIT (value = TRUE)
  SMON10:     ACTIVE /AE/A/A_INIT/GUARD (value = TRUE)
  SMON10:     ACTIVE /AE/A/A_INIT/GUARD (value = FALSE)
   SMON5:     ACTIVE /AE/E1 (value = 99)
   SMON3:     ACTIVE /AE/I (value = 1)
   SMON2:     ACTIVE /AE/Y (value = 7)
   SMON1:     ACTIVE /AE/X (value = 6)
   SMON7:     ACTIVE /AE/DONEA_INIT (value = TRUE)
   SMON8:     ACTIVE /AE/INA_ORIG (value = TRUE)
   SMON6:     ACTIVE /AE/INA_INIT (value = FALSE)
  SMON10:     ACTIVE /AE/A/A_INIT/GUARD (value = FALSE)
  SMON12:     ACTIVE /AE/A/A_ORIG/INC (value = TRUE)
  SMON11:     ACTIVE /AE/A/A_ORIG/INB (value = TRUE)
   SMON7:     ACTIVE /AE/DONEA_INIT (value = FALSE)
  SMON10:     ACTIVE /AE/A/A_INIT/GUARD (value = FALSE)
  SMON16:     ACTIVE /AE/A/A_ORIG/C/INC_INIT (value = TRUE)
  SMON13:     ACTIVE /AE/A/A_ORIG/B/IND (value = TRUE)
  SMON15:     ACTIVE /AE/A/A_ORIG/C/CS (value = 1)
   SMON1:     ACTIVE /AE/X (value = 7)
  SMON17:     ACTIVE /AE/A/A_ORIG/C/DONEC_INIT (value = TRUE)
  SMON18:     ACTIVE /AE/A/A_ORIG/C/INC_ORIG (value = TRUE)
  SMON16:     ACTIVE /AE/A/A_ORIG/C/INC_INIT (value = FALSE)
  SMON17:     ACTIVE /AE/A/A_ORIG/C/DONEC_INIT (value = FALSE)
   SMON2:     ACTIVE /AE/Y (value = 7)
100 FS
   SMON5:     ACTIVE /AE/E1 (value = 100)
  SMON14:     ACTIVE /AE/A/A_ORIG/B/INE (value = TRUE)
  SMON13:     ACTIVE /AE/A/A_ORIG/B/IND (value = FALSE)
   SMON2:     ACTIVE /AE/Y (value = 7)
   SMON1:     ACTIVE /AE/X (value = 7)
110 FS
```

```
   SMON3:    ACTIVE /AE/I (value = 2)
30110 FS
   SMON3:    ACTIVE /AE/I (value = 3)
   SMON4:    ACTIVE /AE/J (value = 2)
60110 FS
   SMON4:    ACTIVE /AE/J (value = 3)
   SMON3:    ACTIVE /AE/I (value = 4)
90110 FS
   SMON3:    ACTIVE /AE/I (value = 5)
   SMON4:    ACTIVE /AE/J (value = 4)
120110 FS
   SMON4:    ACTIVE /AE/J (value = 5)
   SMON3:    ACTIVE /AE/I (value = 6)
```

Time-shifted VHDL simulation results

Note that X and Y do get swapped two times.

```
0 FS
    SMON:    ACTIVE /AE/INA (value = TRUE)
   SMON6:    ACTIVE /AE/INA_INIT (value = TRUE)
  SMON10:    ACTIVE /AE/A/A_INIT/GUARD (value = TRUE)
  SMON10:    ACTIVE /AE/A/A_INIT/GUARD (value = FALSE)
   SMON5:    ACTIVE /AE/E1 (value = 99)
   SMON3:    ACTIVE /AE/I (value = 1)
   SMON2:    ACTIVE /AE/Y (value = 7)
   SMON1:    ACTIVE /AE/X (value = 6)
   SMON7:    ACTIVE /AE/DONEA_INIT (value = TRUE)
   SMON8:    ACTIVE /AE/INA_ORIG (value = TRUE)
   SMON6:    ACTIVE /AE/INA_INIT (value = FALSE)
  SMON10:    ACTIVE /AE/A/A_INIT/GUARD (value = FALSE)
  SMON12:    ACTIVE /AE/A/A_ORIG/INC (value = TRUE)
  SMON11:    ACTIVE /AE/A/A_ORIG/INB (value = TRUE)
   SMON7:    ACTIVE /AE/DONEA_INIT (value = FALSE)
  SMON10:    ACTIVE /AE/A/A_INIT/GUARD (value = FALSE)
  SMON16:    ACTIVE /AE/A/A_ORIG/C/INC_INIT (value = TRUE)
  SMON13:    ACTIVE /AE/A/A_ORIG/B/IND (value = TRUE)
  SMON15:    ACTIVE /AE/A/A_ORIG/C/CS (value = 1)
  SMON17:    ACTIVE /AE/A/A_ORIG/C/DONEC_INIT (value = TRUE)
  SMON18:    ACTIVE /AE/A/A_ORIG/C/INC_ORIG (value = TRUE)
  SMON16:    ACTIVE /AE/A/A_ORIG/C/INC_INIT (value = FALSE)
  SMON17:    ACTIVE /AE/A/A_ORIG/C/DONEC_INIT (value = FALSE)
1 FS
   SMON1:    ACTIVE /AE/X (value = 7)
   SMON2:    ACTIVE /AE/Y (value = 6)
100000 FS
   SMON5:    ACTIVE /AE/E1 (value = 100)
  SMON14:    ACTIVE /AE/A/A_ORIG/B/INE (value = TRUE)
  SMON13:    ACTIVE /AE/A/A_ORIG/B/IND (value = FALSE)
100001 FS
   SMON2:    ACTIVE /AE/Y (value = 7)
   SMON1:    ACTIVE /AE/X (value = 6)
110000 FS
   SMON3:    ACTIVE /AE/I (value = 2)
30110000 FS
   SMON3:    ACTIVE /AE/I (value = 3)
   SMON4:    ACTIVE /AE/J (value = 2)
60110000 FS
   SMON4:    ACTIVE /AE/J (value = 3)
   SMON3:    ACTIVE /AE/I (value = 4)
90110000 FS
   SMON3:    ACTIVE /AE/I (value = 5)
   SMON4:    ACTIVE /AE/J (value = 4)
```

Non-shifted VHDL (generated automatically)

```
use work.A_pack.all;

entity AE is
end;


Architecture AA of AE is
   signal inA   : boolean := false;
   -- NOTE: A's decls (except variables) have been pulled up to here.
   type A_integer_RES is array (natural range <>) of integer;
   function A_integer_RESfct( INPUT :  A_integer_RES ) return integer i
   begin
      assert (INPUT'length = 1) report "overdriven signal,
              type: A_integer_RES" severity warning;
      return INPUT(0);
   end;
   signal x : A_integer_RESfct integer register;
   signal y : A_integer_RESfct integer register;
   signal i : A_integer_RESfct integer register;
   signal j : A_integer_RESfct integer register;
   signal e1 : A_integer_RESfct integer register;
   signal inA_init : boolean :=false;
   signal doneA_init : boolean :=false;
   signal inA_orig : boolean :=false;
   signal doneA_orig : boolean :=false;
begin
   A: block
   begin
       A_init: block (inA_init and not(inA_init'stable))
       begin
           code: process
               variable REMAIN_TIME: time;
               variable GLOBAL_TIME: time;
           begin
           if guard then
           REMAIN_TIME := 0 fs;
           e1 <=  99;
           i <=  1;
           y <=  7;
           x <=  6;
           wait  for REMAIN_TIME;
           doneA_init <= transport true;
           wait  until not (inA_init) ;
           doneA_init <= transport false;
           end if;
           x <= transport null;
           y <= transport null;
           i <= transport null;
           e1 <= transport null;
           wait  on guard;
           end process code;
       end block A_init;
       A_orig: block
           signal inB : boolean :=false;
           signal inC : boolean :=false;
       begin
           B: block
               signal inD : boolean :=false;
               signal inE : boolean :=false;
           begin
               D: block (inD and not(inD'stable))
               begin
                   code: process
                       variable REMAIN_TIME: time;
                   begin
                   if guard then
                   D_Loop : loop
                   x <=  y;
      e1 <=  e1 + 1 after 100 fs;
      wait  until not (inD) ;
      if (not inD ) then
      exit D_Loop;
      end if;
      exit D_Loop;
      end loop D_Loop;
      end if;
      e1 <= transport null;
      x <= transport null;
      wait  on guard;
      end process code;
  end block D;
  E: block
      signal inF : boolean :=false;
      signal inG : boolean :=false;
  begin
      F: block (inF and not(inF'stable))
      begin
          code: process
              variable REMAIN_TIME: time;
          begin
          if guard then
          x <=  y;
          loop
          i <=  i + 1 after 10 fs;
          wait  for 30 ps;
          end loop ;
          wait ;
          end if;
          i <= transport null;
          x <= transport null;
          wait  on guard;
          end process code;
      end block F;
      G: block (inG and not(inG'stable))
      begin
          code: process
              variable REMAIN_TIME: time;
          begin
          if guard then
          loop
          wait  for 30 ps;
          j <=  i after 10 fs;
          end loop ;
          wait ;
          end if;
          j <= transport null;
          wait  on guard;
          end process code;
      end block G;

      control: process begin
            if (inE and not(inE'stable)) then
               inF <= transport true;
               inG <= transport true;
            end if;
         wait until (not inE'stable);
      end process control;
  end block E;
  control: process begin
        if (inB and not(inB'stable)) then
           inD <= transport true;
        elsif (inD and (not e1'stable )) then
           inD <= transport false;
           inE <= transport true;
     end if;
     wait until (not inB'stable)
```

```
                or (inD and (not e1'stable ));
    end process control;
end block B;
C: block
    type C_integer_RES is array (natural range <>)
         of integer;
    function C_integer_RESfct
    ( INPUT :  C_integer_RES ) return integer is
    begin
       assert (INPUT'length = 1) report
       "overdriven signal, type: C_integer_RES"
       severity warning;
       return INPUT(0);
    end;
    signal cs : C_integer_RESfct integer register;
    signal inC_init : boolean :=false;
    signal doneC_init : boolean :=false;
    signal inC_orig : boolean :=false;
    signal doneC_orig : boolean :=false;
begin
    C_init: block (inC_init and not(inC_init'stable))
    begin
        code: process
            variable REMAIN_TIME: time;
            variable GLOBAL_TIME: time;
        begin
        if guard then
        REMAIN_TIME := 0 fs;
        cs <=  1;
        wait  for REMAIN_TIME;
        doneC_init <= transport true;
        wait  until not (inC_init) ;
        doneC_init <= transport false;
        end if;
        cs <= transport null;
        wait  on guard;
        end process code;
    end block C_init;
    C_orig: block (inC_orig and not(inC_orig'stable))
    begin
        code: process
            variable REMAIN_TIME: time;
            variable GLOBAL_TIME: time;
        begin
        if guard then
        REMAIN_TIME := 0 fs;
        loop
        y <=  x;
        GLOBAL_TIME := now;
        wait  until not e1'stable ;
        GLOBAL_TIME := now - GLOBAL_TIME;
        REMAIN_TIME := MAX(REMAIN_TIME - GLOBAL_TIME,0 fs);
        end loop ;
        wait  for REMAIN_TIME;
        doneC_orig <= transport true;
        wait  until not (inC_orig) ;
        doneC_orig <= transport false;
        end if;
        y <= transport null;
        wait  on guard;
        end process code;
    end block C_orig;
    control: process begin
          if (inC and not(inC'stable)) then
             inC_init <= transport true;
          elsif (doneC_init and (true)) then
             inC_init <= transport false;
             inC_orig <= transport true;
```

```
                elsif (doneC_orig and (true)) then
                   inC_orig <= transport false;
                end if;
                wait until (not inC'stable)
                        or (doneC_init and (true))
                        or (doneC_orig and (true));
             end process control;
          end block C;

          control: process begin
                if (inA_orig and not(inA_orig'stable)) then
                   inB <= transport true;
                   inC <= transport true;
                end if;
             wait until (not inA_orig'stable);
          end process control;
       end block A_orig;
       control: process begin
             if (inA and not(inA'stable)) then
                inA_init <= transport true;
             elsif (doneA_init and (true)) then
                inA_init <= transport false;
                inA_orig <= transport true;
             elsif (doneA_orig and (true)) then
                inA_orig <= transport false;
          end if;
          wait until (not inA'stable)
                  or (doneA_init and (true))
                  or (doneA_orig and (true));
       end process control;
    end block A;

start: process begin
  inA <= transport true;
  wait;
end process start;

end AA;
```

A.2 Overdriven Signal Example

This is the example of Figure 16b. Note that the non-shifted VHDL simulation output has an error indicated that a signal was overdriven. The shifted VHDL has no such problem.

SpecChart

```
state
{
   name {A}
   declarations
   {
      signal x : integer ;
      signal evnt : integer ;
   }
   sequential substates
   {
      B : (EI, not (evnt'stable) , C);
      C : ;
   }
}
state
{
   name {B}
   concurrent substates
   {
      D : ;
      E : ;
   }
}
state
{
   name {C}
   code
   {
      x <=  2;
   }
}
state
{
   name {D}
   code
   {
      x <=  1;
   }
}
state
{
   name {E}
   code
   {
      evnt <=  1 after 10 ns;
   }
}
```

Non-shifted VHDL simulation results

Note the overdriven signal error.

```
0 FS
    SMON:    ACTIVE /AE/INA (value = TRUE)
   SMON3:    ACTIVE /AE/INB (value = TRUE)
   SMON6:    ACTIVE /AE/A/B/INE (value = TRUE)
   SMON5:    ACTIVE /AE/A/B/IND (value = TRUE)
   SMON8:    ACTIVE /AE/A/B/D/GUARD (value = TRUE)
   SMON9:    ACTIVE /AE/A/B/E/GUARD (value = TRUE)
   SMON9:    ACTIVE /AE/A/B/E/GUARD (value = FALSE)
   SMON8:    ACTIVE /AE/A/B/D/GUARD (value = FALSE)
   SMON1:    ACTIVE /AE/X (value = 1)
10 FS
   SMON2:    ACTIVE /AE/EVNT (value = 1)
   SMON4:    ACTIVE /AE/INC (value = TRUE)
   SMON3:    ACTIVE /AE/INB (value = FALSE)
   SMON7:    ACTIVE /AE/A/C/GUARD (value = TRUE)
Assertion WARNING in AA: "overdriven signal, type: A_integer_RES"
   SMON6:    ACTIVE /AE/A/B/INE (value = FALSE)
   SMON5:    ACTIVE /AE/A/B/IND (value = FALSE)
   SMON7:    ACTIVE /AE/A/C/GUARD (value = FALSE)
   SMON1:    ACTIVE /AE/X (value = 2)
   SMON8:    ACTIVE /AE/A/B/D/GUARD (value = FALSE)
   SMON9:    ACTIVE /AE/A/B/E/GUARD (value = FALSE)
   SMON9:    ACTIVE /AE/A/B/E/GUARD (value = FALSE)
   SMON8:    ACTIVE /AE/A/B/D/GUARD (value = FALSE)
   SMON1:    ACTIVE /AE/X (value = 2)
1000000000 FS
```

Time-shifted VHDL simulation results

Note the overdriven signal error is eliminated.

```
0 FS
    SMON:    ACTIVE /AE/INA (value = TRUE)
   SMON3:    ACTIVE /AE/INB (value = TRUE)
   SMON6:    ACTIVE /AE/A/B/INE (value = TRUE)
   SMON5:    ACTIVE /AE/A/B/IND (value = TRUE)
   SMON8:    ACTIVE /AE/A/B/D/GUARD (value = TRUE)
   SMON9:    ACTIVE /AE/A/B/E/GUARD (value = TRUE)
   SMON9:    ACTIVE /AE/A/B/E/GUARD (value = FALSE)
   SMON8:    ACTIVE /AE/A/B/D/GUARD (value = FALSE)
 1 FS
   SMON1:    ACTIVE /AE/X (value = 1)
 10000 FS
   SMON2:    ACTIVE /AE/EVNT (value = 1)
   SMON4:    ACTIVE /AE/INC (value = TRUE)
   SMON3:    ACTIVE /AE/INB (value = FALSE)
   SMON7:    ACTIVE /AE/A/C/GUARD (value = TRUE)
   SMON6:    ACTIVE /AE/A/B/INE (value = FALSE)
   SMON5:    ACTIVE /AE/A/B/IND (value = FALSE)
   SMON7:    ACTIVE /AE/A/C/GUARD (value = FALSE)
   SMON8:    ACTIVE /AE/A/B/D/GUARD (value = FALSE)
   SMON9:    ACTIVE /AE/A/B/E/GUARD (value = FALSE)
   SMON9:    ACTIVE /AE/A/B/E/GUARD (value = FALSE)
   SMON8:    ACTIVE /AE/A/B/D/GUARD (value = FALSE)
 10001 FS
   SMON1:    ACTIVE /AE/X (value = 2)
 1000000000 FS
```

Time-shifted VHDL (generated automatically)

```
use work.A_pack.all;

entity AE is
end;


Architecture AA of AE is
   signal inA    : boolean := false;
   -- NOTE: A's decls (except variables) have been pulled up to here.
   function ShiftIfZero( time_expression : in time ) return time is
   begin
      if (time_expression = 0 fs) then
      return (1 fs);
      else
      return (time_expression);
      end if;
   end;
   type A_integer_RES is array (natural range <>) of integer;
   function A_integer_RESfct( INPUT :  A_integer_RES )
            return integer is
   begin
      assert (INPUT'length = 1) report
            "overdriven signal, type: A_integer_RES"
             severity warning;
      return INPUT(0);
   end;
   signal x : A_integer_RESfct integer register;
   signal evnt : A_integer_RESfct integer register;
   signal inB : boolean :=false;
   signal inC : boolean :=false;
begin
   A: block
   begin
      B: block
           signal inD : boolean :=false;
           signal inE : boolean :=false;
      begin
          D: block (inD and not(inD'stable))
          begin
              code: process
                  variable REMAIN_TIME: time;
              begin
              if guard then
              D_Loop : loop
              x <=  1 after ShiftIfZero(1 fs);
              wait  until not (inD) ;
              if (not inD ) then
              exit D_Loop;
              end if;
              exit D_Loop;
              end loop D_Loop;
              end if;
              x <= transport null;
              wait  on guard;
              end process code;
          end block D;
          E: block (inE and not(inE'stable))
          begin
              code: process
                  variable REMAIN_TIME: time;
              begin
              if guard then
              E_Loop : loop
              evnt <=  1 after ShiftIfZero(10 ps);
              wait  until not (inE) ;
              if (not inE ) then
              exit E_Loop;
              end if;
              exit E_Loop;
              end loop E_Loop;
              end if;
              evnt <= transport null;
              wait  on guard;
              end process code;
          end block E;

          control: process begin
                if (inB and not(inB'stable)) then
                   inD <= transport true;
                   inE <= transport true;
                elsif (inB=false and not(inB'stable)) then
                   inD <= transport false;
                   inE <= transport false;
                end if;
             wait until (not inB'stable);
          end process control;
      end block B;
      C: block (inC and not(inC'stable))
      begin
          code: process
              variable REMAIN_TIME: time;
          begin
          if guard then
          x <=  2 after ShiftIfZero(1 fs);
          wait ;
          end if;
          x <= transport null;
          wait  on guard;
          end process code;
      end block C;
      control: process begin
            if (inA and not(inA'stable)) then
               inB <= transport true;
            elsif (inB and (not (evnt'stable) )) then
               inB <= transport false;
               inC <= transport true;
         end if;
         wait until (not inA'stable) or (inB and (not (evnt'stable) ))
      end process control;
   end block A;

start: process begin
```

```
   inA <= transport true;
   wait;
 end process start;

 end AA;
```

A.3 Signal Initialization Example

This is the example of Figure 16d. Assuming y is initially 0, the final value of x during simulation should be 4. In the non-shifted VHDL, y is incremented earlier than x is updated, thus x is 5, which is incorrect.

SpecChart

```
state
{
   name {A}
   declarations
   {
      signal y : integer :=0;
   }
   concurrent substates
   {
      B : ;
      C : ;
   }
}
state
{
   name {B}
   declarations
   {
      signal x : integer :=4;
   }
   code
   {
      x <=  x + y;
   }
}
state
{
   name {C}
   code
   {
      y <=  y + 1;
   }
}
```

Non-shifted VHDL simulation results

Note that x equal 5 at the end, which is incorrect.

```
0 FS
    SMON:    ACTIVE /AE/INA (value = TRUE)
   SMON2:    ACTIVE /AE/INA_INIT (value = TRUE)
   SMON6:    ACTIVE /AE/A/A_INIT/GUARD (value = TRUE)
   SMON6:    ACTIVE /AE/A/A_INIT/GUARD (value = FALSE)
   SMON1:    ACTIVE /AE/Y (value = 0)
   SMON3:    ACTIVE /AE/DONEA_INIT (value = TRUE)
   SMON4:    ACTIVE /AE/INA_ORIG (value = TRUE)
   SMON2:    ACTIVE /AE/INA_INIT (value = FALSE)
   SMON6:    ACTIVE /AE/A/A_INIT/GUARD (value = FALSE)
   SMON8:    ACTIVE /AE/A/A_ORIG/INC (value = TRUE)
   SMON7:    ACTIVE /AE/A/A_ORIG/INB (value = TRUE)
   SMON3:    ACTIVE /AE/DONEA_INIT (value = FALSE)
   SMON6:    ACTIVE /AE/A/A_INIT/GUARD (value = FALSE)
  SMON14:    ACTIVE /AE/A/A_ORIG/C/GUARD (value = TRUE)
  SMON10:    ACTIVE /AE/A/A_ORIG/B/INB_INIT (value = TRUE)
  SMON14:    ACTIVE /AE/A/A_ORIG/C/GUARD (value = FALSE)
   SMON1:    ACTIVE /AE/Y (value = 1)
   SMON9:    ACTIVE /AE/A/A_ORIG/B/X (value = 4)
  SMON11:    ACTIVE /AE/A/A_ORIG/B/DONEB_INIT (value = TRUE)
  SMON12:    ACTIVE /AE/A/A_ORIG/B/INB_ORIG (value = TRUE)
  SMON10:    ACTIVE /AE/A/A_ORIG/B/INB_INIT (value = FALSE)
  SMON11:    ACTIVE /AE/A/A_ORIG/B/DONEB_INIT (value = FALSE)
   SMON9:    ACTIVE /AE/A/A_ORIG/B/X (value = 5)
  SMON13:    ACTIVE /AE/A/A_ORIG/B/DONEB_ORIG (value = TRUE)
  SMON12:    ACTIVE /AE/A/A_ORIG/B/INB_ORIG (value = FALSE)
  SMON13:    ACTIVE /AE/A/A_ORIG/B/DONEB_ORIG (value = FALSE)
1000000000 FS
```

Time-shifted VHDL simulation results

Note that x equals 4 at the end, which is correct.

```
0 FS
    SMON:    ACTIVE /AE/INA (value = TRUE)
   SMON2:    ACTIVE /AE/INA_INIT (value = TRUE)
   SMON6:    ACTIVE /AE/A/A_INIT/GUARD (value = TRUE)
   SMON6:    ACTIVE /AE/A/A_INIT/GUARD (value = FALSE)
   SMON1:    ACTIVE /AE/Y (value = 0)
   SMON3:    ACTIVE /AE/DONEA_INIT (value = TRUE)
   SMON4:    ACTIVE /AE/INA_ORIG (value = TRUE)
   SMON2:    ACTIVE /AE/INA_INIT (value = FALSE)
   SMON6:    ACTIVE /AE/A/A_INIT/GUARD (value = FALSE)
   SMON8:    ACTIVE /AE/A/A_ORIG/INC (value = TRUE)
   SMON7:    ACTIVE /AE/A/A_ORIG/INB (value = TRUE)
   SMON3:    ACTIVE /AE/DONEA_INIT (value = FALSE)
   SMON6:    ACTIVE /AE/A/A_INIT/GUARD (value = FALSE)
  SMON14:    ACTIVE /AE/A/A_ORIG/C/GUARD (value = TRUE)
  SMON10:    ACTIVE /AE/A/A_ORIG/B/INB_INIT (value = TRUE)
  SMON14:    ACTIVE /AE/A/A_ORIG/C/GUARD (value = FALSE)
   SMON9:    ACTIVE /AE/A/A_ORIG/B/X (value = 4)
  SMON11:    ACTIVE /AE/A/A_ORIG/B/DONEB_INIT (value = TRUE)
  SMON12:    ACTIVE /AE/A/A_ORIG/B/INB_ORIG (value = TRUE)
  SMON10:    ACTIVE /AE/A/A_ORIG/B/INB_INIT (value = FALSE)
  SMON11:    ACTIVE /AE/A/A_ORIG/B/DONEB_INIT (value = FALSE)
1 FS
   SMON1:    ACTIVE /AE/Y (value = 1)
   SMON9:    ACTIVE /AE/A/A_ORIG/B/X (value = 4)
  SMON13:    ACTIVE /AE/A/A_ORIG/B/DONEB_ORIG (value = TRUE)
  SMON12:    ACTIVE /AE/A/A_ORIG/B/INB_ORIG (value = FALSE)
  SMON13:    ACTIVE /AE/A/A_ORIG/B/DONEB_ORIG (value = FALSE)
1000000000 FS
```