UC Irvine
ICS Technical Reports

Title
Obtaining functionally equivalent simulations using VHDL and a time-shift transformation

Permalink
https://escholarship.org/uc/item/0nj6w4kd

Author
Vahid, Frank

Publication Date
1991-04-02

Peer reviewed

eScholarship.org — Powered by the California Digital Library, University of California

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

Z
699
C3
no. 91-33

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

Frank Vahid

Technical Report #91-33
April 2, 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 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

1 **Introduction** 1

2 **The Barrier to Obtaining Functionally Equivalent VHDL** 2

3 **The Time-shift Transformation** 5

4 **Improvements** 9

5 **Examples** 10

6 **Results** 10

7 **Conclusion** 11

8 **Acknowledgements** 11

9 **References** 11

A **Appendix** 12
- A.1 Swap examples 13
- A.2 Overdriven Signal Example 15
- A.3 Signal Initialization Example 16

List of Figures

1 Various approaches for simulating domain specific languages 2
2 An example language based on a derivation of StateCharts 2
3 Various translation schemes considered 3
4 Two time scales found in many languages, including VHDL 3
5 Values for each delta point in hierarchical activation scheme, showing that swap fails . . 4
6 Values for each delta point in flattened activation scheme, showing that swap fails 4
7 The current method of mapping control computations to delta-time, causing interference with micro-time functionality 5
8 Time-shifted translation, which prevents control computations from interfering with micro-time functionality 6
9 Implementation of time-shifted translation, showing the transformation step followed by the translation step 6
10 Examples of time-shift applied to domain-specific language's statements 6
11 Time-shift transformation algorithm, applied to a model in the domain-specific language 7
12 Earlier example after the time-shift transformation is applied 8
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 1 fs 8
14 Sample VHDL simulation of earlier example, with inverse shifted and thus final simulation output 9
15 The time-shift can be adjusted to any VHDL units, providing more room for shifted micro-time units 10
16 Examples which cause problems for translation, all solvable using the time-shift 10
17 Examples simulated without and with the time-shift transformation 11

1 Introduction

The VHSIC Hardware Description Language (VHDL) became an IEEE standard several years ago [AyWS86, IEEE88, Wa86]. Due to this (and its technical capabilities), various industry and university VHDL tools have begun emerging, such as simulators and design synthesizers.

However, other languages are still being created and used, intended for different domains, e.g. [Ha87, JePA91, DuHG90, VaNG91]. This is based on the fact that 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. In this paper, we refer to these other languages as being *domain specific.* There are enormous advantages to be gained if these domain specific languages can be converted to VHDL, such as the use of existing powerful 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) more efficient simulations, since VHDL simulator manufacturers concentrate on simulation.

Thus many translation schemes have appeared from domain specific languages to VHDL [ArWC90, DuCH91, MaWa90, NaVG91, JePA91, TiLK90]. Intended uses for the generated VHDL include documentation, simulation, and/or synthesis.

However, we have found that the goal of obtaining completely correct simulations is not reached by many of the schemes, since they fail to create VHDL that is functionally equivalent to the domain specific language's description. The basic problem involves trying to perform 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 (this will be described in detail later). Thus the current translation schemes only work for a subset of descriptions, most failing in maintaining detailed concurrency semantics from the original language. While still useful, the translation is more of a quick and dirty simulation solution than a truly sound simulation 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; i.e. simulated, synthesized, and/or read for documentation purposes. The latter might use VHDL only as an intermediate representation for simulation purposes; a modeler sees only the domain specific language and simulation output. Though the VHDL model might not be functionally equivalent to the model in the domain specific language, the VHDL simulation output can be transformed to correct simulation output (see Figure 1).

By using VHDL as an intermediary format only, we can realize the goal of obtaining fully correct simulations while still benefitting from the advantages listed above. This paper will introduce the fundamental problem that prevents many current schemes from attaining functionally equivalent VHDL. The main goal of this paper is then to introduce a technique that essentially shifts time in the original language (what we call a time-shift transformation), translates to VHDL, simulates, and then shifts the simulation results back. This technique achieves fully functionally equivalent simulation, eliminating the final obstacle that has prevented implementations of domain specific languages from benefitting from VHDL simulators in a truly sound manner.

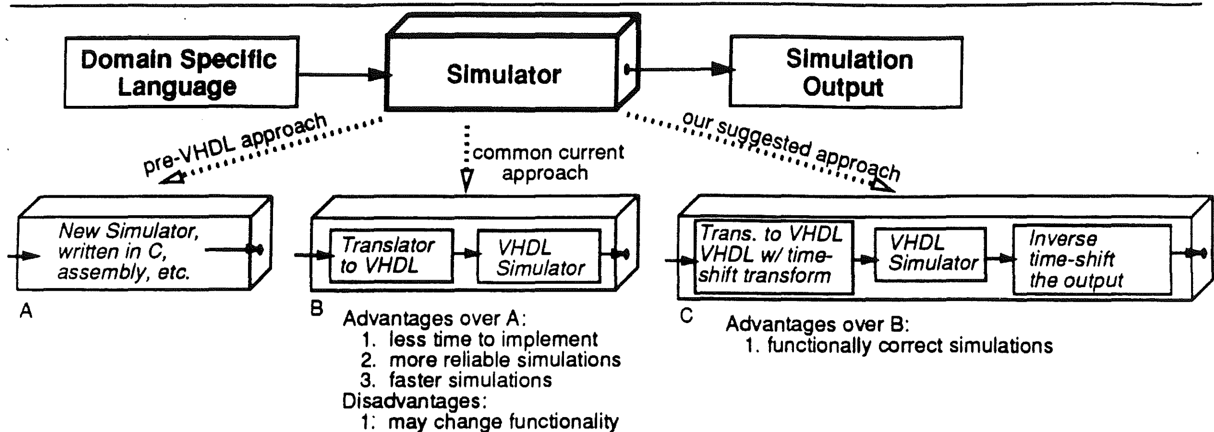
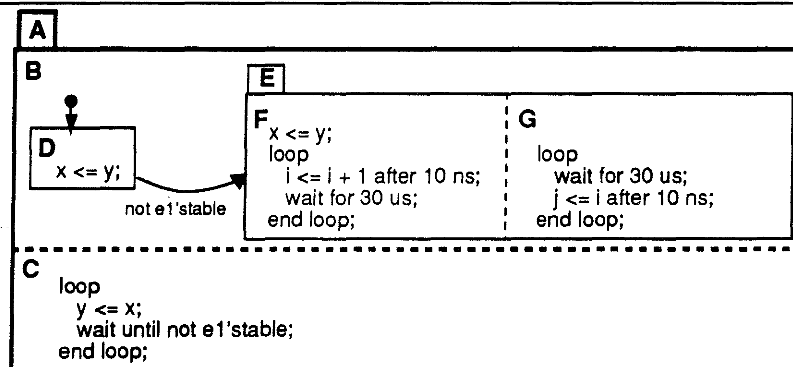


Figure 1: Various approaches for simulating domain specific languages

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 built 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*
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*

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

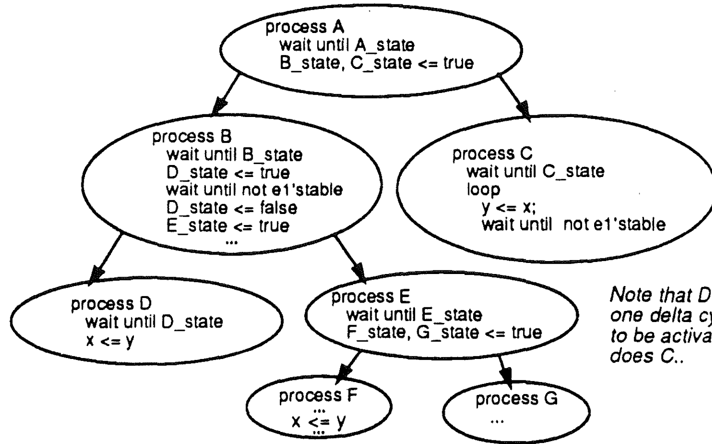
In [ArWC90], a scheme is presented for translating StateCharts to VHDL. Each state is represented as a procedure, and substates are executed by calling each substate's procedure. A procedural model is a sequential model; thus concurrent items in a StateChart become sequential in the VHDL (Figure 3a). 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.

(a) Procedural Model

```
procedure A
  procedure C
  ...
  procedure B
    procedure D
    ...
    procedure E
      procedure F
      ...
      procedure G
      ...
    begin -- E
      F();
      G();
    end  -- E
  begin -- B
    D();
    <when not e1'stable> E();
  end  -- B
  ...
```

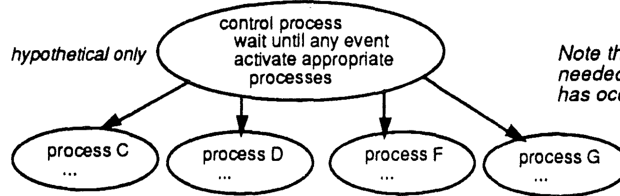
Note that F and G are executed sequentially instead of concurrently

(b) Process Model with hierarchical activation



Note that D takes one delta cycle longer to be activated than does C..

(c) Process Model with flattened activation

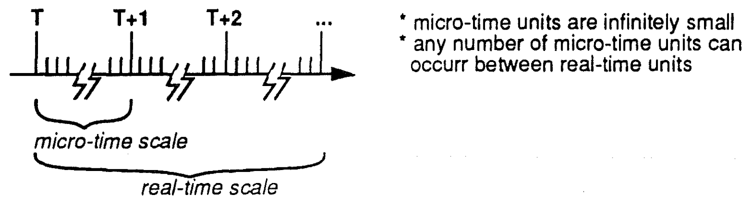


Note that there is still an extra delta cycle needed to activate a process once an event has occured.

Figure 3: Various translation schemes considered

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.



* micro-time units are infinitely small
* any number of micro-time units can occurr between real-time units

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

VHDL is based on a continuous repetition of a simulation cycle. Briefly, each cycle consists of (1) advancing time to the next 'interesting' point, (2) updating signals that should change at

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 (nor should it). 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.

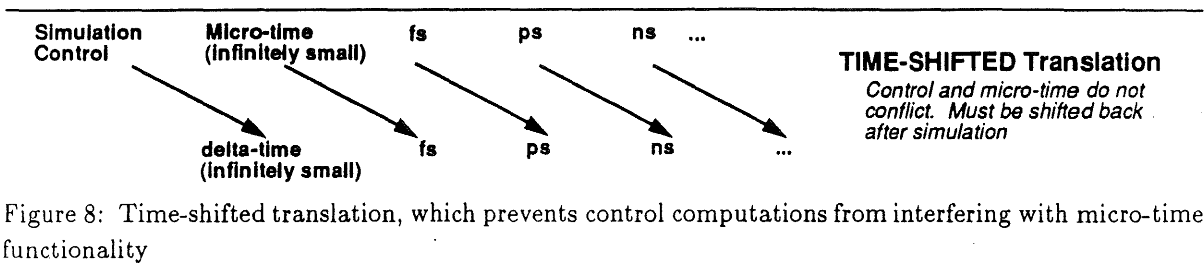


Figure 8: Time-shifted translation, which prevents control computations from interfering with micro-time functionality

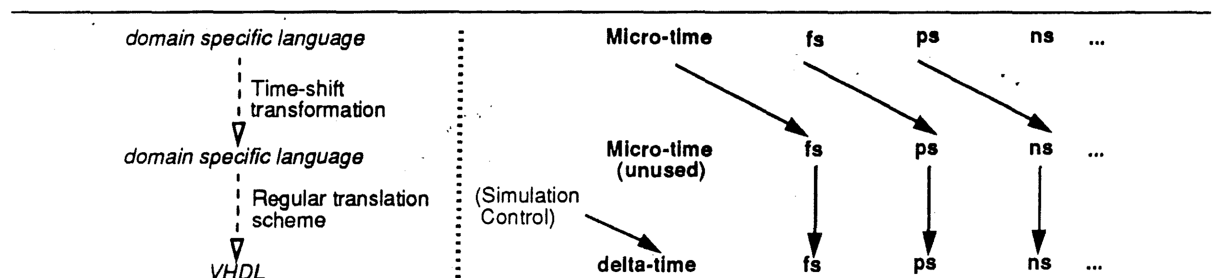


Figure 9: Implementation of time-shifted translation, showing the transformation step followed by the translation step

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 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*>. The *after* clause is optional. Omitting

it implicitly assigns the value after one micro-time unit (one delta). We first make all micro-delays explicit in assignment statements. Thus $x <= y$ becomes $x <= y$ *after 1 micro-unit* (this is an intermediate step, so is not actually a valid statement).

Now, we shift all the occurrences of time units in expressions throughout. Thus everywhere a *micro-unit* appears, we replace it with *fs*. Everywhere *fs* appears, we replace it with *ps*, and so on. Note that all occurrences of *micro-unit* are now gone. See Figure 10 for examples.

There are only two VHDL sequential statements that deal with time: the signal assignment statement and the wait statement. So we must examine these to see if the shift works properly in the domain-specific language.

Assignments with no after clause become assignments with a *1 micro-unit* delay which then becomes a *1 fs* delay, which is correct. If there was an after clause, the time expression's units will have been shifted, since all units throughout have been shifted. For time expressions which evaluate to a non-zero value, the correct result is obtained (e.g. 15 ns becomes 15 us). Now consider the case where the time expression evaluates to 0 ns. The time-shift will have made this 0 us. However, an after clause with a value of 0 ns (or any time unit) is identical to one *without* an after clause, implicitly meaning 1 micro-unit delay. Thus the original 0 ns (1 micro-unit) should have been shifted to 1 fs. We can account for this by using the following function:

```
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;
```

Then, the time_expression in the assignment is replaced by the call *ShiftIfZero(time_expression)* (e.g. $x <= y$ *after t* becomes $x <= y$ *after ShiftIfZero(t)* after the time-shift transformation).

The form of a wait statement is: *wait <on signal_list> <until expression> <for time_expression>*. The *on* and *until* portions are unaffected by the shift. The same discussion of *after* clauses for an assignment statement applies to the *for* clause for the wait statement. Specifically, a wait for 0 is identical to a wait for 1 micro-unit; thus we again replace the time expression by *ShiftIfZero(time_expression)*.

For each signal assignment with no after clause
   create the after clause: *after 1 micro-unit*

Replace each occurence of 'micro-unit' with 'fs', of 'fs' with 'ps', of 'ps' with 'ns', etc., in all expressions.

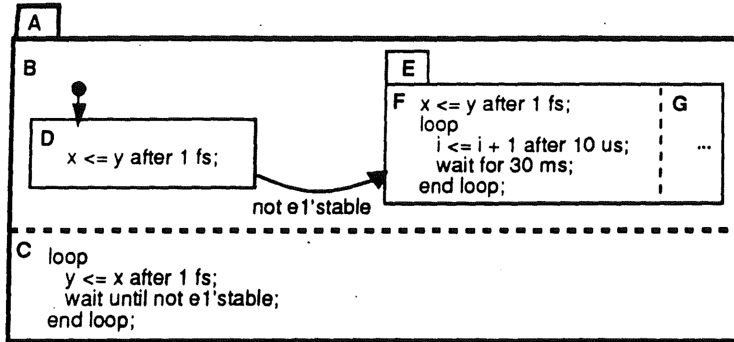
For each signal assignment and each wait statement with a for clause
   replace the statement's time expression (the after or for clause) by:
      ShiftIfZero(time_expression)   *<this routine was defined in the text>*

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

---

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.

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



(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

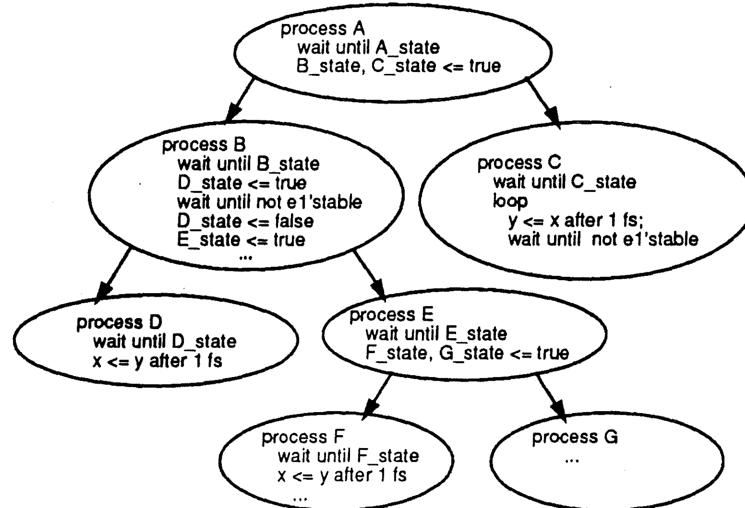


Figure 13: Sample of VHDL processes generated after time-shift; note that control is done in a different time scale than are micro-time assignments, which are now 1 fs

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 y to get 7 after 1 fs.*	D_state = true
50,000,000,001 fs		x = 7
x = 7	*The swap worked*	y = 6
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.*	x = 6
100,000,000,001 fs		y = 7
x = 6	*The swap worked*	
y = 7		

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

4 Improvements

One may notice that in the original domain-specific language, there were infinitely many possible micro-time units that could appear between any two real-time units. However, when we shifted micro-time to femtoseconds, we now have limited the number of micro-time units to 1000, after which they will overlap with picoseconds. In practice, this is a reasonable limitation. We can greatly reduce this limitation by noting that most models don't use more than 2 consecutive time units (e.g. seconds and milliseconds). Let's assume that a single model uses 3 consecutive time units, which is extremely rare. The highest unit can then be shifted to *seconds* in VHDL, the next lower unit to *ms*, and the next to *us*, leaving *ns, ps,* and *fs* available for micro-time when shifted. The number of micro-time units between any two real time units is then 1,000,000,000, which will likely never be exceeded.

Note that the usefulness of the time shift transformation is not limited to merely solving the delta-time interference problem. It can also be used when the domain-specific language uses units that are outside VHDL's range, e.g. kiloseconds, or even years, kiloyears, etc. These can be shifted to VHDL units as discussed above.

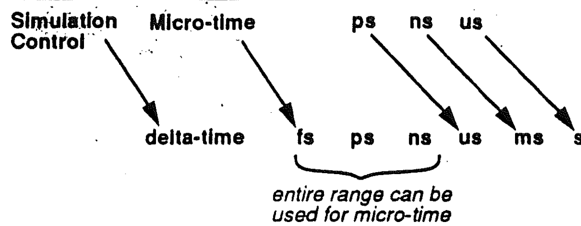


Figure 15: The time-shift can be adjusted to any VHDL units, providing more room for shifted micro-time units

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.

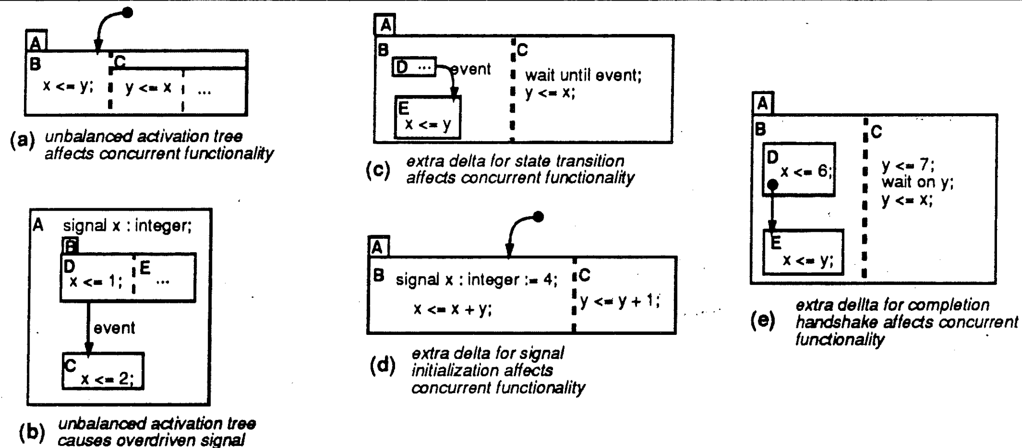


Figure 16: Examples which cause problems for translation, all solvable using the time-shift

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).

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

7 Conclusion

This paper introduced an until now unnoticed and unsurmountable obstacle that prevents translation from domain-specific languages to completely functionally equivalent VHDL. The fundamental obstacle was isolated to be that of simulation control computations necessarily being performed in VHDL's delta time, thus interfering with other delta time computations. We discussed an approach which aims simply for functionally equivalent *simulation*. A novel method of shifting time, translating to VHDL, simulating, and then reshifting the results back was introduced. By this method we can obtain completely correct simulations of the domain-specific language, while still benefitting from the advantages of VHDL simulators. This opens the door to the use of VHDL simulators as part of a truly sound and practical simulation technique.

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)
   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)
```

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 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 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.

```
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 16bd 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
```