Semantic Validation of VHDL-AMS by an Abstract State Machine

Hisashi Sasaki[1], Kazunori Mizushima[2], Takeshi Sasaki[2]
1 Analog & Mixed Signal LSI Design Dept., Toshiba Corp.
2 Dept. of Computer Science, Tokyo Institute of Technology

Abstract
This report presents a semantic analysis for VHDL-AMS, a mixed-signal extension of VHDL, based on an abstract state machine. Intended as a validation for the on-going standardization project, it faithfully reflects the view of simulation proposed. Our experiences proved practical advantages of formal approach in sharing concepts.

1. Introduction

As a language validation for VHDL-AMS [1], there are many approaches according to its focus. 1) Preparations of test examples [2] has two purposes: model description that intent is trivial for everyone is effective to check how we combine language constructs (suggesting practical usage) and to reconfirm the semantics by getting approval from language design team. 2) Test suit attentive to each language construct will be useful not only to check language semantics separately but also to verify the coverage of tool implementation and its correctness. 3) A development of a parser [3] is important to check whether proposed syntax rules is sound for implementation because that language proposal doesn't mention about a language class, such as LR(k). 4) A satisfiability check [4] how design objectives are resolved is important but not so easy to do in some cases, because that design objective is not explained enough. We cannot easily know what is expected as language constructs. 5) The formal approach in this report is most attractive in sharing operational concepts on language proposal. Another formal approach [5] is also undertaken independently. Note that each of approach has its reasons to exit. No one is fully superior to others.

We have selected an abstract state machine semantics [6] for our foundation because it has already fully described semantics for VHDL'93 and reflected LRM (language reference manual) faithfully. As far as we know, other approaches [7] provided the VHDL'87 semantics, and most of them treat a small subset of it, thus they seems for us not easy to apply for practical validation.

There are five factors to prevent understanding: 1) the mixed-signal simulation cycle itself is so complex, 2) the design objective request not to define excessively analog kernel in order to assure free-hands for future advances in equation solving. As a result, it causes a lack of necessary and sufficient formulation, 3) the writing style of LRM is based on a natural language followed in the convention with danger to permit ambiguity, 4) We cannot identify immediately a type of extension: a new concept introduced for new syntax, a new concept overloaded to conventional syntax, a combination of conventional mechanism as not new concept. 5) readers (including us) have bias and misunderstanding caused by unfamiliarity of VHDL simulation cycle.

After establishing our formal model by recognizing above factors, we intentionally formulate a wrong semantics to reconfirm why we could feel such wrong (but not so strange) interpretations admissible.

The remainder is composed as follows: Chapter 2 gives a brief introduction for an abstract state machine. In chapter 3, we extend [6] for analog and mixed-signal

1 1000-1, Kasama-cho, Sakae-ku, Yokohama 247, Japan. E-mail: sasaki @acad.eec.toshiba.co.jp.
2 2-12-1, Ookayama, Meguro-ku, Tokyo 152, Japan. E-mail: {mizkaz, sasaki}@cs.titech.ac.jp

extension. In chapter 4 , based on the developed semantic model, we review our validation experiences. This clarify how misunderstandings are taken. Chapter 5 concludes this report.

2. Abstract State Machine

We set out here the basic definition shortly and refer the readers to Gurevich's work [8] for a rigorous formalization . Abstract state machine (ASM) can be understood as "pseudo-code over abstract data".

A sequential ASM is defined by a finite set of *transition rules* of form

```
if Cond then Updates
```

where *Cond* (condition or guard) is a first-order expression, the truth of which triggers *simultaneous* execution of all update instructions in the finite set *Updates*.

We will give a simple example:

```
if Condition
then  A := B
      B := A
```

This example defines the simultaneous update of A and B. Since the assignments are performed in parallel A becomes the value of B and vice versa. These updates are performed each time *Condition* evaluates to **true**.

Besides simultaneous execution of multiple update instruction guarded by a condition, there is another form of parallelism. A characteristic example which we will use later has the form

```
if $S \in SIGNAL \wedge condition(S)$ then updates(S)
```

where *condition(S)* is a condition and *updates(S)* is a set of update instructions in which S does appear. The meaning of this rule is to simultaneously execute *updates(S)* for each signal S which satisfies *condition(S).*

We also introduce a sequential execution by a delimiter ";" in order to simply express it in analog kernel, which seems to be against the original ASM idea. For example, *A:=B* **;** *B:=C* means that *B:=C* is executed after *A:=B*.

3. The Formal Model

In this section, we first redefine the basic concept for defining the mixed-signal kernel in order to update the original work [6]. Thereafter, we add the formal definition of break statement and simultaneous statements newly invented for mixed signal extension. Finally, we give a definition of the mixed-signal simulation kernel process by adding analog kernel.

3.1 Basic Concept

3.1.1 Mixed Signal Simulation Cycle

Given the underlying digital time model, the domain *TIME* is linearly ordered and contains the distinguished element T_c for *current time*. Assignment to signals are performed by the user defined processes and may cause events at specified point in time. Before reaching the *next simulation time* T_n or before interruption by a2d event, analog kernel process is invoked. After suspending analog kernel process, digital kernel resumes. Each user defined process is executed until digital kernel process suspends. A process becomes suspended upon reaching a wait statement, which then delays the process execution until the timeout expires, or one of the associated signals is updated, or a given expression becomes **true** if one of the corresponding signals is updated.

If all user defined process are suspended, the kernel process executes: digital kernel**:**1) determines the value for the next time point T_n, analog kernel**:** 2) apply break set , 3) find analog solution at T_c, 4) find analog solution at each T_i until a detection of a2d event or reach to next time T_n, digital kernel**:**5) sets the new current simulation time T_c, if required; 6) updates the current values of the relevant signals, and 7) resume the suspended process which are sensitive to the signal changes or timeouts.

3.1.2 Quantities, Equations, Basic set, Augmentation set

Now, we prepare definitions about analog solver and mixed-signal simulation cycle.

Let *QUANTITY* be the set of quantities, classified by its kinds:

$$QUANTITY \equiv QSOURCE \cup QFREE \cup QTERMINAL \cup QDOT \cup QINTEG \cup QDELAYED \cup QZOH \cup QLTF \cup QZTF \cup QRAMP \cup QSLEW$$

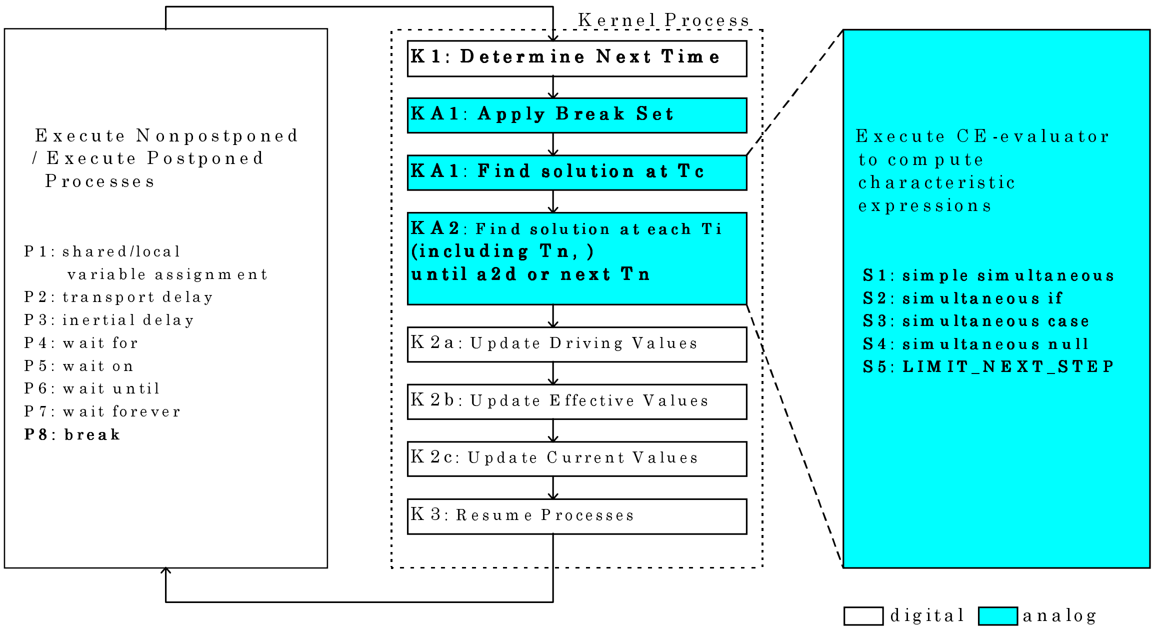

Fig. 1 : mixed-signal simulation cycle

where each set has its name after declaration: for example, *QSOURCE* is a set of quantity declared as *source quantity*.

Let *EQUATION* be the set of maximal execution unit textually described, that is the set union of explicit equations (by simultaneous statements) and implicit equations (by block statements). This concepts corresponds to the user-defined *Process*. It may contain the nested form by simultaneous if/case statements.

For simple speaking, let equation *eq* be given in form of simple simultaneous statement, and its quantities be of scalar type. Then a *characteristic expression char_expr(eq)* of *eq* is the difference between left-hand-side and right-hand-side:

char_expr(eq) = lhs - rls

where

lhs: left-hand-side of equation *eq*
rhs: right-hand-side of equation *eq*

A *basic set* is the set of instantiated characteristic expressions of all equations which is the base for analyses, and to be modified further into *an augmentation set* for a specific analysis.

Basic set = { *char_expr(eq)* | *eq* ∈ *EQUATION* and *char_expr(eq)* is instantiation by *eq* }

Example: Note that *eq* (∈*EQUATION*) is a single execution unit of description, it is not a set of characteristic expressions.

```
If vinput > dvmax use          -- (1)
  gain_current == imax;        -- (2)
elsif vinput < -dvmax use      -- (3)
  gain_current == -imax;       -- (4)
else                           -- (5)
  gain_current == gmnom * vinput; -- (6)
end use;                       -- (7)
```

The statements (2), (4), and (6) dynamically determine a single characteristic expression according to the value of if-conditional in (1) and (3). That is, *EQUATION* is determined at compile time (elaboration), but a *characteristic expression* such as *basic set* is determined at run time (simulation).

An augmentation set is the set of scalar characteristic expressions. It is determined by *QSOURCE, QDOT, QINTEG* and *QDELAYED(T)* because other predefined *QZOH, QLTF, QZTF, QRAMP, QSLEW* are derived from the former four basic kinds of quantity. Let *scalar_subelem(q)* be a set of scalar quantity involved in the composite quantity *q*. Now we choose to define *time domain augmentation set* as a representative example (others are defined by the same way):

time domain augmentation set:
$\{ sq \mid sq \in scalar\_subelem(q) \wedge q \in QSOURCE \} \cup$
$\{ sq\,'DOT - time\_derivative() \mid$
$sq \in scalar\_subelem(q) \wedge q \in QDOT \} \cup$

$\{ sq\ \text{'INTEG} - time\_integral() \mid$
$\quad sq \in scalar\_subelem(q) \wedge q \in QINTEG \} \cup$
$\{ sq\ \text{'DELAYED(T)} - delay\_value\_sq \mid$
$\quad sq \in scalar\_subelem(q) \wedge q \in QDELAYED(T) \}$

where the *time_derivative()* denotes a returned value by an implemented numerical analysis routine as a primitive function. The *time_integral()* is same as *time_derivative().* The *delay_value_sq* is defined as follows:

$$delay\_value\_sq = \begin{cases} sq & \text{when } T = 0.0 \\ sq0 & \text{when } T \leq T_C \\ sq(T_C - T) & \text{when } T_C < T \end{cases}$$

where sq0 is the value of sq at the time when the time domain augmentation set was determined. $sq(T_C - T)$ is the value of sq at time $T_C - T$.

An application of an augmentation set to basic set is defined as a set union:

$apply\ (basic\_set, augmentation\_set) \equiv$
$\quad basic\_set \cup augmentation\_set$

An application of break set to an augmentation set is defined by both a type of augmentation (discontinuity / quiescent) and a form of selector quantity (Q / Q'Integ):

```
Apply-break-set(break_set, augmentation set) ≡
 if discontinuity augmentation set is active
   and selector quantity is in form of Q
 then
 discontinuity augmentation set :=
   discontinuity augmentation set
   – { sq | sq ∈ scalar_subelem(selector quantity) }
  ∪ { sq - sq_value |
    (sq, sq_value) ∈ scalar_subelem(q, q_value),
    (q, q_value) ∈ break_set, q is the selector quantity }
 ...
```

where other omitted cases are defined by the same way.

3.2 User Defined Processes
3.2.1 Processing Statements

The control flow of each iterative *process* (user-defined process) is determined by the environment which provides the dynamic changes of values for the *program_counter*. The *program_counter* of each process is initialized by pointing to the first statement of that process. After having processed the last statement it returns to the first statement again.

In order to express the execution of statement by *Process*, we use the following abbreviation:

Process does < statement >

3.2.2 Break Statements

A break statement sets the flag **true** in order that the user-defined process announces the discontinuity and its treatment (as *break set,* a union of *break-set(Process)*) to the analog kernel. As the flag *break-indicated(Process)* is attached to each process, user-defined processes are executed concurrently. It only prepares the data for analog kernel to treat discontinuity.

P8: SEQUENTIAL BREAK STATEMENT

```
if Process does  < break break_list >
then if  break_list is empty
     then  announce-discontinuity(Process)
     else  announce-discontinuity(Process)
           register-break-set( break_list )
announce-discontinuity(Process) ≡
                     break-indicated(Process) := true
register-break-set( break_list ) ≡
if  break-element ∈ break_list
then add-break-set( break-set(Process),
                     quantity-name, value( expression ) )
```

The procedure *add-break-set* add the pair of the name of quantity and its associated value (value is computed in the execution of this break statement) to *break-set(Process).*

A break with "**when** <condition>" option and a concurrent break statement are interpreted based on this rule.

3.3 CE-evaluator
3.3.1 Processing Simultaneous Statements

CE-evaluator (characteristic expression evaluator) is defined for each eq ∈ *EQUATION.* At first, we give an outline for the equation solving by *SolveEquationsByOracle*. Its argument *set_of_char_expr* is the augmented set to be solved finally, that is the *basic set* augmented by various set such as *discontinuity set* , *time domain augmentation set* etc. *SolveEquationsByOracle* is called recursively until all of characteristic expression are within tolerance, and its body is sequentially executed.

```
SolveEquationsByOracle(set_of_char_expr) $\equiv$
    guess Q's;
    calculate CE;
    check convergence;
    if found-solution = false
    then SolveEquationsByOracle(set_of_char_expr);
```

```
Guess Q's $\equiv$
    if Q $\in$ QUANTITY
    then guess Q by an oracle;
```

The design objective DO37 requests that the language doesn't assume any specific algorithm in advance. So we adopt a keyword *an oracle* to explicitly express it.

```
Calculate CE $\equiv$
if eq $\in$ EQUATION
then CE-Evaluator does < simultaneous _statement(eq) >
```

where arguments of *CE-Evaluator*, simultaneous statements, are simultaneously evaluated with all eq $\in$ *EQUATION*.

```
Check Convergence $\equiv$
    if ($\forall$ expr $\in$ set_of_char_expr
        [ | value( expr ) | $\leq$ value(tolerance( expr )) ]
    then found-solution := true ;
```

tolerance(expr) is the tolerance associated with a characteristic expression *expr*. A *found-solution* flag is always reset before invoking *SolveEquationsByOracle*.

3.3.2 Simultaneous Statements

S1: SIMPLE SIMULTANEOUS STATEMENT

```
if CE-evaluator does < expr1 == expr2 >
then if scalar_expr $\in$ scalar_subelem(char_expr)
        value(scalar_expr) :=
        value(expr1(char_expr)) - value(expr2(char_expr) )
```

where *char_expr* is the characteristic expression for the statement *expr1 == expr2*. *scalar_subelem(char_expr)* gives the set of scalar characteristic expressions *scalar_expr*. *expr1(char_expr)* is a corresponding scalar sub-expressions of *scalar_expr* for *expr1* . *expr2(char_expr)* is a corresponding scalar sub-expressions of *scalar_expr* for *expr2* . Thus, equations using composite type of quantity is converted into scalar one.

3.3.3 LIMIT_NEXT_STEP

S5: LIMIT_NEXT_STEP

```
if CE-evaluator does < LIMIT_NEXT_STEP(Expr) >
then    bound-$T_i$(EQ) := value(Expr)
```

The *value(Expr)* gives the argument evaluation of *LIMIT_NEXT_STEP* at T_i in [T_c. T_n]. Its value will be referred in analog kernel.

3.4 The Kernel Process

Though VHDL-AMS treats various kind of analysis, we will focus only on the time domain analysis here, because it is the core behavior of mixed simulation kernel.

Mixed-signal kernel actions are defined by the rules KA1, KA2 and K1-K3. After determination of next time point by K1, analog kernel KA1 and KA2 will start. The rule KA1 treats the break set and find the solution for the (digital) current time T_c. The rule KA2 find the solutions for each (analog) current time T_i including the (digital) next time T_n. The rule K2 and K3 are to be modified (not mentioned here, see [6]).

3.4.1 Determine Next Time Point

Analog kernel will be invoked only when *cycle* is *time_cycle* and *phase* is *update_driving_value*.

K1: DETERMINE NEXT TIME POINT

```
if AllProcessesSuspended
then /* step X corresponds to the step of simulation cycle in the draft 12.6.4 */
  if $T_n = T_c$
  then cycle := delta_cycle
         phase := update_driving_values   ... (for step c)
  elsif cycle = delta_cycle
          then cycle := postponed_cycle
                 phase := execute_postponed   ... (for step j)
          else   cycle := time_cycle
                 phase := update_driving_values   ... (for steps c, d, e )
                AdvanceTime;   ... (step b)   $T_c$ is updated for next sim cycle

                 apply_break_set; .... by analog kernel KA1
                 find_solution_for_Tc;   ... by analog kernel KA1

                 find_solution_for_each_Ti ;   ... by analog kernel KA2_
```

```
$T_n$ :=
    if DOMAIN=TIME_DOMAIN    ... (step i)
    then universal_to_physical_time(0.0)
    else min { time-high, mindriver, mintimeout }
```

```
AdvanceTime $\equiv$
if $T_n$ $\leq$ TIME'HIGH then $T_c$ := $T_n$ else phase := undef
```

```
time-high = TIME'HIGH,
mindriver = min { time(t) |
              $\exists$ d $\in$ DRIVER: t = $t^i$, active(d) = I }
              where
              $t^{true}$ = first(d) and $t^{false}$ = second(d)
mintimeout = min { timeout(p) |
   p $\in$ PROCESS $\wedge$ timeout(p) $\neq$ undef $\wedge$ timeout(p) $\geq$ $T_c$ }
```

```
UpdateDrivers(Time) $\equiv$
```

```
if $d \in DRIVER \wedge tail(d) = <> \wedge timeout(second(d)) = Time$
then d := tail(d)
    active(d) := true
```

3.4.2 Analog Kernel

At first, time domain augmentation set is determined, then applied to basic set. To process break, *merge-break-set* and *break-indicated*, *apply_break_set* are used. After fixing *set_of_char_expr_Tc*, solution is found by *solveEuationsByOracle*:

KA1: APPLY BREAK SET & FIND SOULTION FOR T_c

```
determine time_domain_augmentation_set;
set_of_char_expr := apply(basic_set, time_domain_augmentation_set);
merge-break-sets;
if  break-indicated then
        set_of_char_expr :=
                apply(set_of_char_expr, discontinuity_augmentation_set ) ;
end if;
set_of_char_expr_Tc := apply_break_set ( set_of_char_expr, break_set );

found-solution := false ; /* set the flag false for SolveEquationsByOracle */
SolveEquationsByOracle(set_of_char_expr_Tc); --- for time $T_c$
clear-break-indication;
solve-For-Each-analog-cycle;   --- solve for time $T_i$ in $[T_c , T_n]$
```

```
break-indicated ≡
        ($\exists P \in PROCESS$  [ break-indicated(P)=true ] )
```

If there exists a process *P* such that *break-Indicated(P)* = ***true***, then break-indication *break-indicated* is announced to kernel.

```
merge-break-sets ≡
  if  P ∈ PROCESS
  then   if (q, q-value) ∈ break-set(P)
            then add-break-set(break-set, (q, q-value))
```

merge-break-sets gathers the *(q, q-val)* pairs as *break set*: *add-break-set* is the procedure to add its pair to *break set*.

```
clear-break-indication ≡
        break-set := $\phi$
        break-Indicated := false
        if P ∈ PROCESS
        then  break-Indicated(P ) := false
                break-set(P) := $\phi$
```

clear-break-indication immediately reset the effects of break statement.

By the following behavior KA2, it is easy to see what is the difference between Q'Above(E) and an evaluation of the expression Q-E > 0 which causes no suspension of analog solver.

KA2: FIND SOLUTION FOR EACH T_i

```
solve-for-each-analog-cycle ≡
    guess next $T_i$ ;     /* $T_i$ in the interval $[T_c , T_n]$ , $1 \leq i$ */
    if  $T_i \geq T_n$          --- $1 \leq i$
    then
            /* no a2d event occurred in this interval $[T_c, T_n]$ */
            /* and reached the time next digital event to be occurred */
            $T_i := T_n$ ;
             /* let last $T_i$ be $T_n$ in order to include $T_n$ ; $T_i$ must includes $T_n$' */
            determine time_domain_aug_set;
            set_of_char_expr := apply(basic_set, time_domain_aug_set);
            found-solution := false ;
            SolveEquationsByOracle(set_of_char_expr);
            if  a2dEventQAbove $\neq \phi$   /* detect a2d event for $T_n$*/
            then  setDriverQAbove;
                    /* analog solver suspended here ... */
    else
            /* set the flag false for SolveEquationsByOracle */
            determine time_domain_aug_set;
            set_of_char_expr := apply(basic_set, time_domain_aug_set);
            found-solution := false ;
            SolveEquationsByOracle(set_of_char_expr);
            if  a2dEventQAbove $\neq \phi$   /* detect a2d event for $T_i$ */
            then  setDriverQAbove;
                    /* analog solver suspended here ... */
            else
                    solve-for-each-analog-cycle;
            end if;
    end if;
```

```
guess next $T_i$ ≡
if.  $\exists eq \in EQUATION$ [bound-$T_i$ (eq)  is defined]
then  guess next $T_i$ by using bound-$T_i$(eq)
     /* such that $\forall$ eq. [ $|T_i - T_{i-1}| \leq$ bound-$T_i$(eq) ] */
else  guess next $T_i$  without limit;
```

The value of *bound-T_i(EQ)* gives maximum limit of time step. By using *Guess next T_i* , it does not specify any algorithmic details.

```
setDriverQAbove ≡
if Q'Above(E) ∈ a2dEventQAbove
then
   $T_n$ := universal_to_physical_time($T_i$ )
   driver(Process,Q'Above(E)):=<notvalue(Q'Above(E)), $T_n$>
   active(driver(Process, Q'Above(E))) := true
```

```
a2dEventQAbove ≡ {Q'Above(E) ∈ QAboveSignal |
                        contradictory( Q'Above(E) ) }
contradictory(Q'Above(E)) ≡
    (value(Q - E) > 0.0  $\wedge$ value(Q'Above(E)) = false )
  $\vee$ (value(Q - E) < 0.0  $\wedge$ value(Q'Above(E)) = true)
```

If there exits a contradictory implicit Q'Above(E), the driver of implicit Q'Above(E) signal is assigned a waveform <***not*** *(value(Q'Above(E))),T_n*> by *setDriverQAbove*.

4. Experiences in Language Validation

Here we will show our experiences how we had made wrong interpretations. The reasons we dare to expose our failures and struggles is to illustrate examples such that the formal methods not only gives the theoretical foundations but it is practically useful.

4.1 Difficulties in the Guideline of LRM Design

There are two concepts of sequential statement relating to user-defined processes and simultaneous procedural statements. But proposal provide single non-terminal symbol *sequential_statement*. In fact, the draft defines BNF as:

```
Simultaneous_procedural_statement ::=
   [ procedural_label : ] [ pure | impure ]
      procedural [ is ]
          procedural_declarative_part
      begin
          procedural_statement_part
      end procedural [ procedural_label ];

 procedural_statement_part ::=
     { sequential_statement }

 Sequential_statement ::=
          wait_statement
        | assertion_statement
        | report_statement
        | signal_assigment_statement
         ...
        | null_statement
        | break_statement
```

But also LRM draft says:
It is also an error if a wait statement or a signal assignment statement occurs in the procedural statement part.

Note that break_statement is not mentioned.

4.2 How we made misunderstandings
4.2.1 Break Statement

Following was the typical misunderstanding caused by the above decision on language design.

```
begin
    -- use break to set the phase initial condition
    break Phase => 0.0;
    -- another break statement keeps the phase within 0.. 2pi
    break Phase => Phase mod TwoPi when Phase > TwoPi;
    -- phase equation
    Phase'dot == max(0.5 MHz, fc+(Vin-Vc)*df);
       -- output voltage source equation
       Vout == 2.5*(1.0+sin(Phase));
end architecture PhaseIntegrator;
```

The underlined statement should be corrected by "**on** Phase' Above(TwoPi)".

4.2.2 Expiration of Break Effects

Next we will formulate the wrong interpretation excluded by LDC to give clue to nobreak/unbreak misunderstanding. The wrong break statement could be defined related to CE-evaluator:

```
S6: (SIMULTANEOUS)   BREAK STATEMENT
if CE-evaluator does   < break break_list >
then
      if   break_list is empty
      then   announce-discontinuity2(eq, Break);
      else    announce-discontinuity2(eq, Break)
              register-break-set2( break_list );

announce-discontinuity2(eq)  ≡
      Break-indicated2(eq) := true;
     /* the variable Break-indicated2(eq) will
          be referred for each $T_i$ */
register-break-set2( break_list )  ≡
 if break-element  ∈  break_list
 then
 /* assume break-element is in form of the pair
 < quantity-name , expression > */
      add-break-set2( Break-Set2(eq),
          quantity-name, value( expression ) )
break-indicated2  ≡
 ( ∃ eq ∈ EQUATION [ break-Indicated2(eq)=true ] )
```

We can define nobreak statement:

```
S7: (SIMULTANEOUS)   NO BREAK STATEMENT
if CE-evaluator does   < nobreak >
then   clear-break-indication2

clear-break-indication2  ≡
       break-set2 :=  $\phi$
       break-Indicated2 := false
       if eq  ∈ EQUATION
       then break-Indicated2(eq ) := false
             break-set2(eq) :=  $\phi$
```

By this definition, we give some possible event models which show when break indication is set and reset. Note that there are two kind of break-indication in our models: break-indication relating to process and break-indication2 relating to equation. Fig. 2 shows the various candidates for effective lifetime for break. In (a) and (c), as there is some duration of time point which break effect is ON, it is natural to want to cancel the effects of break, because that we afraid redundant break operations will make simulation speed slow. In (b) and (d), as lifetime is instantaneous, no such an anxiety.

Further, even worse, we could have other twisted interpretation as (e) and (f): All above assumed that there are two independent clear-break-indication, but (e) and (f) share the single clear-break-indication for two different kinds of break.

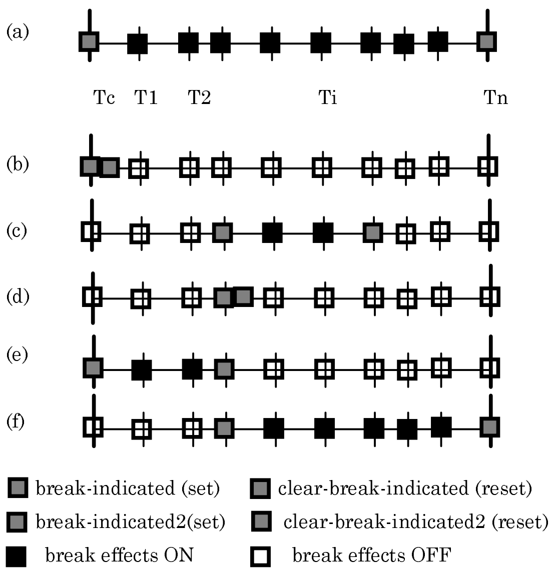

Fig. 2 : effective lifetimes of break

The point here is the fact that we can ask LDC what is their intent based on the accurate model.

4.2.3 Discussion on the *Implicit Break Signal*

Next is more philosophical discussion. You may feel no difference on actual behavior by the both formulation. But it is so critical in concept that it will affect the difficulty / complexity in understanding.

For the execution of a break statement the condition, if present, is first evaluated. A break is indicated if the value of the condition is TRUE or if there is no condition. If a break is indicated, the driver of the implicit BREAK signal is assigned the waveform TRUE after 0 sec and then each break element is evaluated in the order in which the elements appear.

A faithful model to above LRM is:

```
announce-discontinuity(Process, Break) ≡
    driver(Process, Break) := <true, $T_C$>;
    active(Process, Break) := true;
```

In our understanding, flag set is better to show straight meaning: A Flag is expressed as the predicate *break-indicated(Process)* explained by **P8** in the previous chapter.

By stating that *implicit break signal* is not signal (which has a specific meaning in view of digital kernel), we can avoid the undesirable confusion that break effects would continue over the several simulation cycles. Without ASM, we cannot make delicate discussions to consider the lifetime of break effects.

5. Conclusion

We had provided the formal foundation on VHDL-AMS, and verified its practical usefulness. We wish that such an approach will be popular in our standardization activity in future and the writing style of LRM will be improved.

Acknowledgment

We would like to thank members of analog-task group EIAJ (Toshiyuki Saito, Nokuya Emoto, Jun-ichiro Toyozumi, Goichi Yokomizo, Seitaro Shinbara, Prof. Hidetoshi Onodera), and LDC and VAL team of 1076.1 WG, for discussing on validation works. Especially, Ernst Christen and Kenneth Bakalar for helpful e-mail advises, and Prof. Alain Vachoux for his help to analog-task group. We would also like to thank Prof. Naoki Yonezaki, Prof. Takuya Katayama and VHDL-project EIAJ (Satoshi Kojima, Prof. Masaharu Imai, Kenji Yoshida) for their miscellaneous supports, and Prof. Egon Boeger, Prof. Wolfgang Muller for their e-mail encouragement.

References

[1] IEEE/DASC 1076.1 Working Group, "1076.1 Working Document Definition of Analog Extensions to IEEE Standard VHDL", June 1996, May 1, and July 1 1997.

[2] Tom Kazmierski, Mark Zwolinski, Southampton VHDL-AMS validation Suits, accessible by http: //www. syssim. ecs. soton. ac.uk/index.html

[3] Vasu Shanmugasundaram, Hal Carter, the analyzer for VHDL-AMS, accessible by http://www.ececs.uc.edu/~vasu

[4] Hisashi Sasaki, et. al., 1076.1 validation by EIAJ team, accessible by http: //www. tamaru. kuee.kyoto-u. ac.jp /1076.1 /index. htm.

[5] M. Madrid, P. T. Breuer, C. D. Kloos, "A semantic model for VHDL-AMS", CHARME'97, October 1997.

[6] Egon Boerger et.al., "A Formal Definition of an Abstract VHDL'93 Simulator by EA-Machines," in [7]

[7] Carlos D. Kloos, Peter, T. Breuer (editors), Formal Semantics for VHDL, 1995, Kluwer Academic Publishers.

[8] Yuri Gurevich, "Evolving Algebras 1993: Lipari Guide", in Specification and Validation Methods, Ed. E. Boerger, 1994, Oxford University Press.