

VHDL 1076.1: The Design of a Language Extension for VHDL

Kenneth Bakalar
COMPASS Design Automation

Ernst Christen
Analogy, Inc.

Abstract—The analog extensions to VHDL 1076 developed under IEEE PAR 1076.1 will be brought to ballot in 1997. There have been a number of presentations in other venues focusing on the application of the extended language. In contrast, this paper presents the rationale for the design of the language extension itself.

Introduction

IEEE Standard VHDL 1076-1993 [1] describes a discrete, event-driven language and simulation engine that has proven useful to the designers of digital electronics. VHDL-1076.1 describes an extension to VHDL-1076 that adds syntax, semantics and a simulation model for continuous and mixed continuous/discrete systems. The extended language (VHDL 1076 including the VHDL 1076.1 extensions) is informally called VHDL-AMS.

The defining document for the extension is the VHDL 1076.1 Language Reference Manual which is in final review preparatory to balloting by IEEE members. The current paper explores some aspects of the rationale for the design of VHDL 1076.1.

This paper assumes substantial familiarity with the VHDL 1076-1993 definition. It is concerned with time-domain mixed mode simulation in VHDL-AMS. The language can be used for frequency domain simulation as well; see [2] for further discussion.

The VHDL Legacy

The requirements for VHDL 1076.1 are described in the Design Objective Document and Design Objective Rationale compiled from raw requirements submitted by a group of interested participants to the 1076.1 Working Group during 1992 and 1993 (see [3]). Paramount among these is the re-

quirement to preserve all aspects of the original language in fitting the extension.

The VHDL legacy of VHDL-AMS has both advantages and drawbacks. VHDL-AMS builds on a firm pragmatic, semantic and syntactic foundation, including structural and functional decomposition, separate compilation, well defined notions of time and concurrency, a sequential programming notation and a powerful type system. On the other hand, the base language presents many constraints on the design of the extension that should be violated only with good reason: its scope and visibility rules, the requirement for declaration before use, the structure of object names, the orderly execution of processes, the nature and timing of elaboration and the discrete simulation cycle, among others.

The approach of the Language Design Committee has been a parsimonious one. We have used VHDL-1076—its syntax, semantics, unifying principles and stylistic quirks—in satisfying as many of the design requirements as possible, adding only what is essential and still missing. Thorough exploitation of the powerful VHDL 1076 engine has led to some surprising and innovative design choices that set VHDL-AMS apart from other continuous system simulation languages.

DAEs and the Analog Solver

The behavior of the lumped systems targeted by VHDL 1076.1 can be described by systems of ordinary differential and algebraic equations (DAEs) of the form

$$\mathbf{F}(\mathbf{x}, d\mathbf{x}'/dt, t) = 0 \quad (1)$$

where \mathbf{F} is a vector of functions, \mathbf{x} is the vector of unknowns, \mathbf{x}' is a subset of \mathbf{x} and t is time. A necessary (but not sufficient) condition for the existence of a solution to the system is that \mathbf{F} and \mathbf{x} have the same dimension.

In practical models it is often the case that the functions \mathbf{F} are different in different intervals and the system is approximated by a piece wise concatenation of segments. If discontinuities in an unknown occur at a segment boundary then consistent initial values must be reestablished for the unknowns. It is important to note that the theory does not provide any method for the automatic detection of discontinuities of this sort. The subject of initialization and reinitialization is discussed further in [2].

Systems of DAEs do not generally have analytical solutions, but in recent years a variety of well-founded and serviceable numerical approximations have been devised to calculate \mathbf{x} once initial conditions have been established [4].

VHDL 1076 appeals to the "conventional mathematical meaning" of the operators on real and integer values for purposes of definition. In a similar manner, VHDL 1076.1 appeals to the theory of continuous mathematics. It is therefore unnecessary to specify a particular numerical algorithm for solution of the DAEs of a model. Indeed, it is undesirable to do so, since this would constrain implementations unnecessarily and somewhat arbitrarily.

Instead, an abstract agent called the *analog solver* is given the task of solving the system (to an indeterminate but sufficient degree of accuracy; see "Tolerances", below) whenever a solution is required. The definition is completely silent on the matter of how the analog solver should go about performing this task. (The analog solver is also given certain responsibilities regarding the detection of threshold crossings; see "Time and the Simulation Cycle", below).

The Quantity

It follows from the necessity to describe systems of DAEs that VHDL 1076.1 must provide a notation for equations and a model for the unknowns in those equations. We deal with the second matter first.

The unknowns in the collection of DAEs implied by the text of a model are analytic functions of time; that is, they are piece wise continuous with a finite number of discontinuities at threshold crossings. The analog solver computes the values of all unknowns simultaneously at a sequence of times in the intervals between discontinuities. The only other writing reference occurs when the initial values are established at the beginning of an interval.

A new class of time-varying object apart from VHDL's signals and variables is required to model the unknowns of the DAEs—a class not susceptible to writing reference by an agent other than the analog solver and the initialization mechanism.

Members of the new object class *quantity* provide the required semantics. Quantities are defined on close analogy to signals, differing primarily in their novel means of assignment. The scalar subelements of quantities model the unknowns of the system (1).

Each quantity has a name, type and subtype. Quantities may be composite, and the name of a portion of a quantity is constructed using selection, indexing, and slicing. A quantity declaration may appear in any declarative part in which a signal is allowed, with the single exception of packages. Here is an example of a quantity declaration:

```
quantity Velocity: real;      (2)
```

The scalar subelements of a quantity must be of a floating point type to approximate the real numbers of the underlying mathematical formalism,

Quantities are static like signals. All quantities are created during elaboration and are neither created nor destroyed during simulation.

The name $Q \cdot$ denotes a quantity whose scalar subelements are the time derivatives of the corresponding scalar subelements of

Q . The implicit quantities Q'_{integ} , representing the integrals over time of the scalar subelements of Q , are defined in a similar way. $Q'_{delayed}(t)$ is a copy of Q with subelements delayed by t . These implicitly declared quantities are first class objects that can be used anywhere the name of a quantity is appropriate. Higher order derivatives and integrals are readily created; for example, $Q'_{dot'dot}$.

Simultaneous Statements

VHDL-AMS supplements the sequential and concurrent statements of VHDL-1076 with a new class of language statements for notating DAEs. A simultaneous statement can appear in any context in which a concurrent statement is allowed. The elementary form of simultaneous statement is the *simple simultaneous statement*.

The simple simultaneous statement contains a pair of expressions at least one of which must refer to quantities. The definition requires the analog solver to establish values for the scalar subelements of quantities such that the difference between matching scalar subelements (the *characteristic expressions*) of all such pairs of expressions are (close to) zero. The expression may include function calls and reading references to constants and signals.

The characteristic expressions of the simple simultaneous statements correspond to elements of the function vector \mathbf{F} of the underlying formalism.

Here is an example of a simple simultaneous statement. In an electrical model the constitutive equation of a resistor could be written as

$$e == i*r; \quad (3)$$

where e and i are quantities representing the voltage across and the current through the resistor, and r is a constant representing the resistance value. The element of \mathbf{F} derived from this statement is

$$e-ir \quad (4)$$

where e and i are now unknowns of the system.

Three additional forms of the simultaneous statement are defined by reduction rules to

a set of one or more simple simultaneous statements. This is analogous to the relationship between concurrent statements and processes (recall that each concurrent statement is defined in terms of an equivalent process).

The *simultaneous procedural statement* superficially resembles a VHDL process. The form provides a way to write the function f of the simple simultaneous statement

$$q == f(x,q); \quad (5)$$

"in line" (q is an aggregate of quantities and x is an arbitrary collection of other value-bearing objects). The sequential modeling style afforded by the simultaneous procedural statement is familiar and comfortable to analog designers who have experience with older notations such as the ACSL family of continuous simulation languages.

The *simultaneous case statement* and *simultaneous if statement* exhibit a family resemblance to their sequential counterparts. Each contains arbitrary lists of simultaneous statements in its statement parts, including nested simultaneous case and if statements. Only the simple simultaneous or simultaneous procedural statements selected by the case expressions and chosen by the conditional expressions are considered by the analog solver.

Finally, the *simultaneous null statement* has no behavioral semantics but serves the simultaneous regime in a way analogous to its concurrent and sequential counterparts.

Time and the Simulation Cycle

VHDL uses the physical type `Time` to represent the current simulation time. Physical types use an integer-based representation and thus have constant absolute precision over their entire range. The division of time into integer multiples of a base unit is fundamental to the VHDL simulation cycle.

In contrast, the numerical solution of DAEs is time invariant, and most calculations are performed with times near zero. A floating point representation has the appropriate characteristics for these computations: a constant relative precision, with absolute precision indefinitely high near zero, decreasing as absolute value increases.

Synchronization between the analog solver and the VHDL kernel process requires a common formulation for simulation time that encompasses both requirements. This is embodied in a new definitional type called *Universal_Time*. Values of *Universal_Time* are deemed to have sufficient precision to represent exactly each floating point value and each value of physical type *Time*. The definition provides a set of precise rules governing the conversion with truncation of *Universal_Time* to types *Time* and *Real*.

The VHDL simulation cycle is recast using values of *Universal_Time* for the kernel variables T_c , which represents the current simulation time, and T_n , which represents the next time that a driver will become active or a process will resume.

The VHDL simulation cycle is augmented to include the execution of the analog solver. The analog solver executes in each simulation cycle just before the current simulation time advances. The solver establishes a sequence of solutions to the DAEs (*analog solution points*, or ASPs) at suitable intervals between T_c and T_n .

The implicitly declared crossing signal $Q'_{Above}(C)$ is a Boolean valued attribute of the scalar quantity Q . C is a static amplitude threshold. $Q'_{Above}(C)$ is *True* when Q is greater than C and otherwise *False*.

If any scalar quantity named in a crossing signal passes the designated threshold before the sequence of ASPs is extended all the way to T_n , the analog solver will reset T_n to the current time, schedule a delta delayed event on the corresponding signal driver and terminate prematurely.

The predefined function *NOW* is redefined to return T_c , converted with truncation to the nearest value of physical type *Time*. *NOW* is overloaded with a real-valued function that returns T_c converted with truncation to the nearest value of type *Real*. The predefined attribute *Last_Value* has been similarly overloaded. A wait statement may contain a timeout clause with a real valued delay.

A process sensitive to a crossing signal or containing a wait statement with a real valued delay may resume at a Uni-

versal_Time not exactly representable as a value of type *Time* or type *Real*.

If no quantity is declared or used in a given model then the augmented simulation cycle reduces to the VHDL-1076 simulation cycle. If no signal is declared (including implicitly declared crossing signals) then the degenerate simulation cycle consists, after initialization, of a single execution of the analog solver.

The Mixed Mode Model

The essence of a mixed mode modeling language is the ability to communicate synchronized data and control information between modeling domains. The elements described in the preceding sections provide a wide and flexible communication channel.

The scope and visibility rules for signals and quantities are identical. They can be declared side by side in the same declarative part. A reading reference to a quantity is allowed anywhere it is visible, in particular within a concurrent statement; a reading reference to a signal is allowed anywhere within a simultaneous statement. Concurrent and simultaneous statements can be mixed arbitrarily in the same block statement part. The simulation cycle guarantees that all of the quantities of a model have been updated to the values appropriate to the current simulation time when any process executes, and conversely that the values of signals are current (and constant) in the interval T_c to T_n during the execution of the analog solver. The synchronous transfer of data between the discrete and continuous domains is only a matter of using the correct name.

A process can be sensitive to multiple crossing signals and it can make arbitrary investigations into the state of the system by interrogating signals, variables and quantities when it gains control. If the required state is identified, the process can execute signal assignments to notify other parts of the system. Since a signal may in turn be used in a simultaneous statement to determine the elements of \mathbf{F} in (1), the control loop is complete.

We conclude that A/D and D/A converters of arbitrary complexity and including hysteresis and conditional thresholds can be coded directly in VHDL-AMS, and that arbi-

```

entity slipper is
  generic (mass: real := 1.0, slip: real := 0.001, stick: real := .01);
  port(quantity external_force: in real; quantity position: out real);
begin
  assert mass /= 0.0 report mass'instance_name & " must not be zero.";
end slipper;

architecture brick of slipper is
  quantity effective_force, acceleration, velocity: real;
  signal stopped: boolean := true;
begin

  process begin
    state1:
      stopped <= true;
      wait on external_force'above(stick), external_force'above(-stick);
    state2:
      stopped <= false;
      wait on velocity'above(0.0) until abs(external_force) < stick;
  end process;

  position'dot == velocity;
  velocity'dot == acceleration;
  acceleration == effective_force/mass;

  break on stopped, velocity'above(0.0);
  if stopped use
    effective_force == 0.0;
  elsif velocity>0.0 use
    effective_force == external_force - slip;
  else
    effective_force == external_force + slip;
  end use;

end brick;

```

Figure 1: Model of a block on a rough surface

trary control machines for piece wise defined continuous models are easily coded as well. The primitive power of VHDL-AMS obviates the need for special purpose language constructs.

There is one more element of control; recall that the defining theory works only when the unknowns in (1) are continuous, and that discontinuities cannot in general be detected automatically. A new sequential *break statement* (with a concurrent version defined by an equivalent process) is added so the modeler can signal the occurrence of a discontinuity.

Figure 1 illustrates some of these ideas. It represents a model for a block on a rough surface. The block is stopped at time zero. It

experiences the force `external_force`. If `abs(external_force)` becomes greater than `stick` the block begins to move. A force `slip` (due to kinetic friction) opposes the direction of motion while the block is moving. If `velocity` crosses 0.0 with `abs(external_force) < stick` then the block stops again.

The conditional simultaneous statement selects an equation based on the values of signal `stopped` and quantity `velocity`. The concurrent `break` statement tells the analog solver about the discontinuity in `effective_force` that occurs when `velocity` crosses zero or when `stopped` changes from True to False and back again.

Conservative systems

Systems that obey conservation laws—for example, electrical systems obeying Kirchhoff's laws—merit separate treatment because they are so commonly encountered. The description of conservative systems in VHDL 1076.1 uses a graph-based conceptual model. For an electrical network, the vertices of the graph represent equipotential points in the circuit and the edges represent branches of the circuit through which current flows.

Equations describing the conservative aspects of such a system need not be explicitly notated by the modeler because they can be extracted automatically from the topology of the network. Only the so-called constitutive equations remain the modeler's responsibility. The simple simultaneous statement (3) relating the voltage across and the current through two points connected by a resistor is an example of such a constitutive equation.

Electrical systems are by no means the only interesting conservative systems. The conservative style is appropriate for modeling in the thermal, hydraulic, and mechanical domains as well. A single network never connects vertices of one domain to those of another.

The automatic generation of conservation equations grants such an overwhelming advantage over manual construction that the considerable increase in language complexity it engenders is more than worthwhile.

The Terminal

Members of the object class *terminal* form the base of the required semantics. The scalar subelements of terminals model the vertices in a conservative network. Terminals are not value bearing objects and so they are not characterized by a VHDL type. A terminal can be declared in any declarative part in which a signal is allowed. The rule forbidding the interconnection of vertices from different domains is statically enforced by assigning each terminal a *nature* at declaration and then forbidding the interconnection of terminals of different natures. For example, the following declaration creates a pair of terminals of nature electrical:

```
terminal T1,T2: electrical; (6)
```

Branch quantities represent the unknowns in the constitutive equations of a conservative system description. There are two kinds of branch quantities: *across quantities* and *through quantities*. Across quantities represent effort like effects such as voltage, temperature, or pressure. Through quantities represent flow like effects such as current, heat flow rate, or fluid flow rate. Connections between terminal elements are created whenever a through quantity is declared. Here are the quantity declarations for *e* and *i* needed to support (3):

```
quantity i through T1 to T2; (7)  
quantity e across T1 to T2;
```

The order of the terminals defines the polarity of the quantity; the declaration

```
quantity e1 across T2 to T1; (8)
```

creates a quantity *e1* equal to $-e$. A branch quantity declaration contains no explicit information indicating the type of the new quantities. Rather, the type and constraints are derived from information carried by the nature of terminals T1 and T2.

Nature

A terminal is declared to be of some *simple nature* or of a composite nature with scalar subelements that are of a simple nature. Each simple nature represents a distinct physical discipline—electrical, thermal, hydraulic, and so on. A simple nature includes the names of two subtypes for the scalar subelements of branch quantities; one for across quantities and one for through quantities. The simple nature electrical might be declared:

```
nature electrical is (9)  
  real across  
  real through;
```

A composite nature has elements of a simple nature. The system of natures and subnatures in VHDL 1076.1 closely parallels the system of types and subtypes in VHDL 1076; for example, it is possible to declare an unconstrained array nature and subsequently to specify an index constraint in the subnature indication of a terminal declaration.

A branch quantity inherits the "shape" of its terminals; for example, a branch quantity with terminals of an array nature is also an array. If one terminal is scalar and the other is composite, then the branch quantity inherits the shape of the composite terminal. The type and constraints of the scalar subelements of a branch quantity are determined by the simple nature of its terminals.

The scalar terminals of the scalar subelements of a composite quantity are the matching scalar subelements of the composite terminals of the composite quantity (figure 2a.). If one of the terminals is scalar and the other composite, then the scalar terminal matches each element of the composite terminal (figure 2b.).

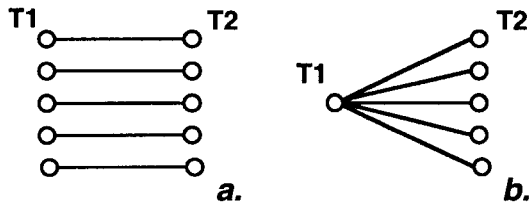


figure 2: Matching elements of terminals.

The declaration of a simple nature implicitly creates a reference terminal which is shared by all terminals with elements of that simple nature. In an electrical system, the reference terminal corresponds to ground. The reference terminal of the simple nature of nature N is designated N'Reference.

The declaration of a terminal creates two quantities in addition to the terminal itself. The *reference quantity* T'Reference is an across quantity with T and N'Reference as its terminals. Each scalar subelement of the *contribution quantity* T'Contribution is equal to the sum of all through quantities incident to its matching elements (with appropriate sign) and the through quantities incident to the matching elements of each terminal with which T is associated (see "Modularization", below).

The reference quantity is a convenient way to refer to the voltage (temperature, pressure, etc.) at a terminal. The contribution quantity is useful for calculating power dissipation.

Modularization

The modular structure and binding mechanism of VHDL 1076 remains unchanged in VHDL 1076.1. However, new interface elements have been added to the block header and entity declaration port clause and port map aspect to accommodate the declaration and association of *interface quantities* and *interface terminals*.

The association of terminals builds netlists from models of parts. The most important use of quantity association is to build data flow models from abstract operational blocks.

Interface quantities and terminals are first class objects with all the characteristics of their standard counterparts.

The semantics of the association of terminals is quite simple. An interface terminal is modeless. It is completely accurate to think of two terminals that are associated as if they were one terminal with two different names. Each scalar subelement of the actual is associated with the matching scalar subelement of the formal.

The association of formal and actual quantities has the same effect as the simple simultaneous statement

$$\text{formal_quantity} == \text{actual_quantity}; \quad (10)$$

If conversion functions are present, they are applied as appropriate to the formal and actual quantities in the equivalent statement.

An interface quantity can have the modes **in** or **out** (the default is **in**). Here is the entity declaration of a data flow multiplier:

```
entity multiplier is (11)
  generic (Gain: real := 1.0);
  port (
    quantity A,B: in real;
    quantity C: out real);
end multiplier;
```

A locally declared quantity used as an actual may be associated with a formal of mode **in** or **out**. If the mode of the actual is **in**, then the corresponding formal must be of mode **in**.

Each scalar subelement of an actual of mode **out** should be associated exactly once with a formal of mode **out**, or it should appear in exactly one simultaneous statement (not both). The element may also be associated with formals of mode **in**.

Modes on interface quantities make it possible to enforce the rule requiring an equal number of equations and unknowns locally within each external block. This has an enormous practical advantage; in a hierarchical model with hundreds of equations and unknowns the module containing a missing or superfluous equation can be quickly identified.

Tolerance

A real implementation of the abstract analog solver can calculate only an approximation to the true solution of the system (1). The accuracy of the approximation is inversely related to the time spent on the calculation. The language must provide a mechanism that allows the user to trade off accuracy against computation time.

A review of feasible implementations leads to the conclusion that, in the general case, tolerance parameters are needed for each unknown and for each function in (1). Because different implementations of the analog solver will characterize tolerance in different ways, it is not possible to specify the parameters in the language definition. Instead, the tolerance is indicated with an uninterpreted string that must be given meaning by a particular implementation.

The subtype of a quantity provides a convenient vehicle for tolerance constraints as it does for range and index constraints. The VHDL 1076 subtype indication is augmented to include a *tolerance aspect* to propagate the necessary information to quantities. For example,

```
subtype mvolts is (12)
  real tolerance "low_voltage";
```

creates a subtype more suitable in practice for the across type mark of the simple nature in (9). The tolerance codes associated with the type marks of a simple nature can be overridden in a subnature indication or a branch quantity declaration.

A tolerance code is associated with the characteristic expressions derived from a simple simultaneous statement in one of two ways. If one of the expressions is the name of a quantity then the tolerance code of each characteristic expression is the tolerance code of the matching scalar subelement of the quantity. If neither expression is in the required form or if the modeler wishes to override the default, an explicit tolerance aspect can be added to the statement.

Completing the Vector **F**

As we have shown, some of the functions **F** in (1) are derived from the simultaneous statements of the user's code. The remainder are implied by the topological structure of the model, the laws of conservative systems and the definitions of implicitly declared quantities. The result is a system that meets a necessary condition for solvability; the number of equations is equal to the number of unknowns. The additional functions are:

- For each implicit quantity $Q \cdot$, $Q \cdot \text{integ}$ and $Q \cdot \text{delayed}$, a function constraining the quantity to its mathematical definition,
- The difference between the reference quantities of the formal and actual terminals of each scalar terminal association,
- The difference between the formal quantity and actual quantity or expression of each scalar quantity association, and
- For each scalar subelement of a locally declared terminal, the sum of the scalar through quantities incident to that terminal and any terminal with which it is associated directly or indirectly (with attention to sign).

Impure References

A practical implementation of the analog solver may evaluate the characteristic expressions of a model many times. When it does so it must interpret them according to the ordinary expression evaluation semantics of VHDL. Iterative algorithms assume that only the values of the unknowns change from iteration to iteration. If a simultaneous statement includes references

to shared variables or calls to impure functions there is a danger of side effects that violate this condition.

Simultaneous statements by default cannot reference shared variables or impure functions. But there are many benign uses for impure code; for example, to write messages as an aid to debugging. VHDL 1076.1 allows a simple simultaneous statement or simultaneous procedural to be marked "impure", in which case the references otherwise prohibited are allowed. It then falls on the modeler to guarantee that side effects are harmless.

ACKNOWLEDGEMENT

The authors wish to acknowledge their colleagues on the 1076.1 Language Design Committee for their considerable contribution to the design of VHDL-AMS.

REFERENCES

- [1] *IEEE Standard VHDL Language Reference Manual*, ANSI/IEEE Std 1076-1993.
- [2] E. Christen and K. Bakalar. "VHDL 1076.1: Analog and Mixed-Signal Extensions to VHDL". To be published in J.-M. Bergé, O. Levia, J. Rouillard (ed.). *Current Issues in Electronic Modeling, Vol. 10: Analog and Mixed-Signal Hardware Description Languages*. Kluwer, 1997.
- [3] C.-J. R. Shi and A. Vachoux. "VHDL -A Design Objectives Rationale" In J.-M. Bergé, O. Levia, J. Rouillard (ed.). *Current Issues in Electronic Modeling, Vol. 2: Modeling in Analog Design*. Kluwer, 1995.
- [4] K. E.. Brenan, S. L.. Campbell, and L. R.. Petzold. *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*. North-Holland, 1989.