Chapter 2

MIXED NETS, CONVERSION MODELS, AND VHDL-AMS

John Shields

Lynguent, Inc.
P.O. Box 19325
Portland, OR 97280-0325

jshields@ieee.org

Ernst Christen

Synopsys, Inc.
2025 NW Cornelius Pass Rd.
Hillsboro, OR 97124

Ernst.Christen@synopsys.com

Abstract   AMS hardware description languages like VHDL-AMS provide features for modeling at discrete and continuous domains of abstraction and communicating between them. A mixed net arises in mixed-signal design as the result of interconnecting components modeled in different domains, in particular when connecting a discrete and a continuous port. Hardware description languages do not support such connections directly. They require the insertion of an appropriate conversion model between the dissimilar ports. Using conversion models correctly, a mixed net can be successfully partitioned and modeled with the desired blend of accuracy and performance.

This paper explains mixed nets and their various configurations, setting the requirements for needed conversion models. Conversion models are explained, including criteria for what makes a good one. Strategies for partitioning a mixed net and inserting conversion models are discussed. A proposal is made for extending VHDL-AMS to handle mixed nets and automatic insertion at elaboration.

Keywords: VHDL-AMS,Verilog-AMS,mixed signal,conversion models,elaboration

1.   Introduction

Our goal is to describe and simulate a mixed-signal system design with the required accuracy. By reducing model complexity, more of the overall system can be verified. But in order to verify some aspects, the most complex implementation models are needed. Mixed-signal HDLs like VHDL-AMS were designed to support these modeling tradeoffs. The design process using VHDL-AMS still leads to structural problems when switching between two models designed to represent the same component in different domains. Solving these problems efficiently requires conversion models to be systematically inserted.

1.1   Background and Motivation

Mixed-signal modeling involves using models in the discrete domain with models in the continuous domain. With VHDL-AMS, one can create rich models of physical systems of many different energy domains. In an electrical system, for example, the discrete domain is modeled by concurrently executing processes that communicate through signals of some logic type like std_logic. The electrical nature in the continuous domain is modeled with differential-algebraic equations of voltages, currents, and other unknowns. These models may be conservative systems, i.e., a circuit obeying Kirchoff's laws. They may also be signal flow models, where a non-conservative quantity (most likely voltage or current) flows through transfer functions.

A complete set of electrical components that can be used together for modeling a mixed-signal system will include digital models and both types of analog models. One component may have a set of models in different domains designed to be equivalent. Models that are from different domains have both different implementations *and* interfaces. Nevertheless, the interfaces are closely related. There is an equivalence between corresponding ports, i.e., a given logic signal port is equivalent to its corresponding electrical pin, voltage input, etc. At the same time, the analog model interface may have additional ports that are not relevant in the discrete model, such as power/ground connections.

Any component from the mixed-signal set may be connected to others to form a system model. Component models from different domains may be used interchangeably and should support flexible and efficient design styles. It follows that suitable conversions must exist between the discrete, the signal flow, and the conservative analog domains. Indeed, such conversion behavior is at the heart of languages like VHDL-AMS. The system model can be modeled in VHDL-AMS today. It turns out that suitable domain conversions must be designed as models themselves and be part of the component set. Effective system design styles can be supported if it is possible to specify reasonable rules to insert conversion models automatically and to bind implementations of such models to each instance of a conversion model.

1.2   Design Styles

Composing mixed-signal systems from a set of components is a structural task typically done best graphically, for example in a schematic entry system. Nevertheless, portions may be written directly in the HDL or generated from other design tools. A top-down design methodology leads from a high level of design abstraction at the system or behavioral level to a lower level of abstraction going toward the physical implementation level. Moving top-down in abstraction does not necessarily mean crossing modeling domain. Some components may move from behavioral to rtl to gate level to switch level and remain discrete models. The models ultimately exist as continuous models at the analog circuit level, yet there may be no need to use them in a system model for verification. If one can safely avoid using the circuit level model in the system model, the designer asserts that the digital model is equivalent to its analog counterpart. The designer further asserts that the component is sufficiently decoupled at this level in the system such that its analog aspects are accounted for and can be ignored.

There are few straightforward paths in top-down design for mixed-signal systems. Digital subsystems are decomposed with significant synthesis support. Analog subsystems have comparatively little synthesis. If your mixed-signal component set has analog components with a high level behavioral model and low level implementation model, it supports both top-down and bottom-up modeling. Designing an analog model may start with circuit topology at the implementation level and be modeled upwards, or from a top-down behavioral model and be modeled downwards. You adjust the parameterization to meet specifications. If you require an equivalent model at another abstraction level, there are two cases to consider. From the behavioral model, there is a creative leap to the implementation model and bottom-up verification to establish their equivalence. From the implementation model, there is a more organized and potentially automated transition upwards to the behavioral model. Bottom-up verification for equivalence is the same task, but here it takes the form of calibrating the behavioral model parameters to the implementation model.

Back at the system level, it may be ideal to simulate the entire system at the analog implementation level. When you choose not to, there is no substitute for bottom-up verification. As you proceed upwards in subsystem validation, component models start at the lowest level of abstraction and are swapped for equivalent models that are at higher levels of abstraction and/or cross domains from analog conservative to analog signal flow to discrete.

The common denominator of the top-down and bottom-up design styles is the need for a design composition system that is effective for hierarchical structure by providing flexible configuration of components and their underlying

simulation models. Schematic entry systems are effective, in part. They compose the structure flexibly and may be made configurable enough, but there is a difficult netlisting problem into the underlying HDL. VHDL-AMS can represent the structure provided the conversion models are explicitly included in the system model. VHDL-AMS also has the underlying features for design configuration that were intended to meet these needs. Unfortunately, they are not usable where component interfaces change across domains.

1.3   Problem Definition

Design styles suitable for mixed-signal systems need interchangeable component models from different domains, methods to efficiently switch between them, and automatic conversions between domains in order to meet the simulation accuracy and performance goals. VHDL-AMS supports the modeling needs for such components and the conversions, but lacks support to efficiently describe the hierarchical structure of the resulting systems so they can be reconfigured easily. After reviewing the mixed net modeling and conversion concepts, a solution to this problem will be proposed.

2.   Mixed Nets and Conversion Models

A net consists of a root, typically a terminal, quantity or signal declared in a block, and all ports connected to the root, including transitive connections. Its structure is a tree. A mixed net is a net whose root and connected ports belong to different object classes. In a typical mixed net, some leaves of the tree may be terminal ports, other leaves may be quantity ports, and still others may be signal ports. The root and the ports higher up in the tree typically only define the connectivity between component instances; their semantics are usually not of much concern. Nonetheless, these ports and the root must be declared to be objects of a particular class: in VHDL-AMS, a terminal, a quantity, or a signal.

During simulation, it is the simulator's task to determine a value for each net, taking into consideration the contributions from the various ports that form the net (or more precisely, the contributions from the behavioral statements in which the names of the ports appear). VHDL-AMS defines semantics for uniform nets, that is, nets whose root and ports are either all terminals, or all quantities, or all signals. Other AMS languages have similar uniformity rules for nets. Therefore, to perform a simulation, a mixed net must be split into uniform portions using some partitioning strategy, and suitable code must be inserted at the boundaries of the different portions of the net to convert between the semantics of, for example, a terminal and a signal. The preferable way to manage such conversion code is to place it in *conversion models*, which then are instantiated such that they link the different portions of the net.

2.1 Net Partitioning Strategies

There are many different ways to split a mixed net into uniform portions, each with different properties. We discuss four strategies that embody different ideas, using VHDL-AMS terminology.

User-Defined Partitioning. In this strategy, the user defines the root of the net and each port of the net to have a particular object class: terminal, quantity, or signal. The object classes represent different modeling domains. Instances of conversion models are inserted between the formal and the actual of a port association element if the formal and actual are of different object classes. The benefit of this approach is that supporting it in VHDL-AMS requires few language changes. Its drawback is that even in simple situations it may be too difficult for a user to determine how to declare the ports in different parts of the net to achieve a certain goal (performance, accuracy). For example, it is easily possible that a mixed net might have several disjoint terminal nets (or nodes), each with a different potential.

The remaining three strategies have two aspects in common: We ignore the object class of the root and the intermediate ports and only honor the object class of the ports that are leaves of the tree forming the net. We also consider the object classes to correspond to abstraction levels, with a terminal being the most detailed and a signal being the most abstract.

Partitioning Driven by Elaboration. This strategy considers, for each vertex in the tree describing the net, the object class of the vertex and its immediate children and converts this portion of the net to the most detailed of these object classes. If the result is different from the object class of the vertex, then an instance of a conversion model is inserted between the vertex and its ancestor. The benefit of this approach is its simple elaboration rules. Its drawback is the sensitivity of its result to changes of a leaf port: replacing a leaf port connected higher up in the tree has more dramatic effects on the result than replacing a leaf port lower in the tree. That is, a structural design change may lead to an unexpected change in the mixed net representation and surprising behaviour.

Partitioning for Performance. The goal of this strategy is to minimize the number of instances of conversion models. Sub strategies include: a) simulating each mixed net as two or three uniform nets, each having a subset of the topology of the mixed net, and inserting instances of conversion models between the net replicas, and b) separating the signal net into two nets, one connecting all ports with mode **in**, the other, connecting all other signal ports, and inserting instances of conversion models between the terminal net (if any) and each signal net. The advantage of this strategy is performance. Its drawbacks are the complexity inherent in having multiple representations of a single net and the difficulty of incorporating drive and load characteristics in the con-

version models without adding significant features to the language, such as the capability to determine the fanins and fanouts of a port from inside a model.

Partitioning for Accuracy. In this strategy, a mixed net is converted in its entirety to a uniform net whose object class is that of the most detailed object class of the root or any port of the net. For example, if any terminal is connected to the net, then the net is converted to a terminal. Instances of conversion models are inserted between the net and any leaf port whose object class is different from the object class of the net. The benefits of this approach are its ability to accurately model drive and load characteristics at ports of a higher abstraction level and its predictability, which makes it easy to understand. Its drawback is the potentially large number of instances of conversion models, and performance may be affected by the models of the lowest abstraction connected to the net as well as by the number of conversion model instances

2.2   Categories of Conversion Models

The discussion about partitioning strategies yields the result that there is a need for conversion models that convert between the semantics of terminals, quantities, and signals. To also satisfy the VHDL-AMS rules about port association elements, which are based on the mode of the formal port, we end up with seven categories of conversion models:

- Terminal to signal with mode **in** (conservative to event-driven, commonly called a2d)

- Signal with mode **out** to terminal (event-driven to conservative, commonly called d2a)

- Signal with mode **inout** or **buffer** and terminal (commonly called bidirectional)

- Terminal to quantity with mode **in** (conservative to signal flow, called TQ below)

- Quantity with mode **out** to terminal (signal flow to conservative, called QT below)

- Quantity to signal with mode **in** (signal flow to event-driven, called QS below)

- Signal with mode **out** to quantity (event-driven to signal flow, called SQ below)

Note that there is no possibility to have a bidirectional conversion model between quantity and signal because of the semantics of a quantity net.

It is apparent from this list that a conversion model always has a direction, even in the case of a bidirectional conversion model, where the direction changes over time, driven either by the operation of the conversion model or by control information such as switching (on the signal end of the conversion model) between high impedance (input) and driving (output) state.

Each category of conversion models is further parameterized by the type of the signal or quantity or the nature of the terminal on either end of the conversion model. Regardless of the partitioning strategy, any mechanism to bind a category of conversion models to a particular implementation of a conversion model must be rich enough to support this parameterization.

2.3 Implementation of Conversion Models

The language elements of VHDL-AMS are sufficient to implement any conversion model, regardless of the particular combination of input and output object and the corresponding types and/or natures. For an input or output object of class terminal, this includes the possibility of converting between its reference quantity (for an electrical terminal: the voltage w.r.t. ground) or its contribution quantity (for an electrical terminal: the current flowing through the terminal) and the value of the object at the other end of the conversion model.

Conversion models between terminals and quantities are straightforward to implement because of the closeness of the semantics of the two object classes. A TQ conversion model is essentially a quantity source whose value is controlled by the reference or contribution quantity of the terminal. Similarly, a QT conversion model is either a quantity controlled across source or a quantity controlled through source.

Conversion models between signals and quantities or terminals have some similarities in that they involve converting between discrete time semantics and continuous time semantics. For an a2d or a QS conversion model, this can be accomplished, in general, using a threshold based approach that involves a signal of the form Q'Above(E), where Q is a quantity and E is the threshold. For a d2a or an SQ conversion model, the general approach is that of a controlled source whose value is driven by the value of the signal. A bidirectional conversion model combines the functionality of an a2d and a d2a conversion model, possibly with some extra code to switch its direction. In the remainder of this section, we will focus on a2d and d2a conversion models; SQ and QS conversion models are essentially subsets of a2d and d2a. We further restrict the discussion to conversion models with an electrical terminal and a signal of type std_logic.

The specific implementation of an a2d or a d2a conversion model can be rather ideal, taking into consideration only the voltage (or current) and pos-

sibly the impedance at the terminal end, or very detailed, modeling the load or driving characteristics of a particular technology such as cmos. In either case, the conversion model can be parameterized to match the properties of a particular physical device.

Ideal conversion models are the easiest to implement. As examples, we show the implementation of ideal a2d and d2a conversion models converting to or from a voltage. The a2d conversion model is based on a finite state machine that drives the output to '1' if the input voltage exceeds a threshold vhi, to '0' if the input voltage is below vlo, and to 'X' if the input voltage stays between vlo and vhi for longer than a timeout. A possible implementation of the model and the corresponding FSM are shown in Figure 2.3 and Figure 2.1 respectively [Christen, 1999].

The corresponding ideal d2a conversion model can be implemented as a voltage source with an output resistance where both the output voltage and the resistance are controlled by the signal value. A possible implementation of the model is shown in Figure 2.2 [Christen, 1999].

To better reflect the load and driving characteristics of a particular technology, a model writer can write technology specific conversion models that implement the load (for a2d) or the driving (for d2a) characteristics of the technology. For example, the driving characteristics of a conversion model for the cmos technology can be modeled by describing the channel properties of the two transistors at the output of a cmos gate, with its operation controlled by the input signal value. Conversion models with such detail typically need additional ports that provide the power supply for the model and the reference. They also have parameters that let the user parameterize an instance of the model to reflect the driving properties of a particular port of a physical device. The mechanism to bind a particular instance of a conversion model to a design unit with the necessary detail must therefore support the specification of appropriate parameter values for that instance and the connection of its power and reference terminals (and any other port that may be required, for example a port that controls the direction of a bidirectional conversion model) to suitable objects in the block in which the conversion model is instantiated.

3.   Current Approaches to Automatic Insertion of Conversion Models

Once appropriate conversion models have been designed and components that give rise to their use are available, the designer focuses on the system design task. The designer is engaged in structural composition tasks, and automatic insertion of conversion models is very desirable for improved productivity, repeatability, and correctness. There are automated solutions today, but none that is well integrated with the VHDL-AMS language.

```vhdl
library ieee;
use ieee.std_logic_1164.all;  use ieee.electrical_systems.all;
entity a2d is
   generic (vlo, vhi: REAL; – thresholds
      timeout: DELAY_LENGTH);
port ( terminal ain, ref: electrical;  signal dout: out std_logic);
end entity a2d;
architecture Hysteresis of a2d is
   type st4  is (unknown, zero, one, unstable);
   quantity vin  across ain  to ref;
   function level(vin, vlo, vhi: REAL) return st4 is
   begin
      if vin < vlo  then return zero;
      elsif vin > vhi  then return one;
      else return unknown;
      end if;
   end function level;
begin
   process
      variable state:st4 := level(vin, vlo, vhi);
   begin
      case state is
      when one =>
        dout <= '1';
         wait on vin'Above(vhi);
        state := unstable;
      when zero =>
        dout <= '0';
         wait on vin'Above(vlo);
        state := unstable;
      when unknown =>
        dout <= 'X';
         wait on vin'Above(vhi), vin'Above(vlo);
        state := level(vin, vlo, vhi);
      when unstable =>
         wait on vin'Above(vhi), vin'Above(vlo) for timeout;
        state := level(vin, vlo, vhi);
      end case;
   end process;
end architecture Hysteresis;
```

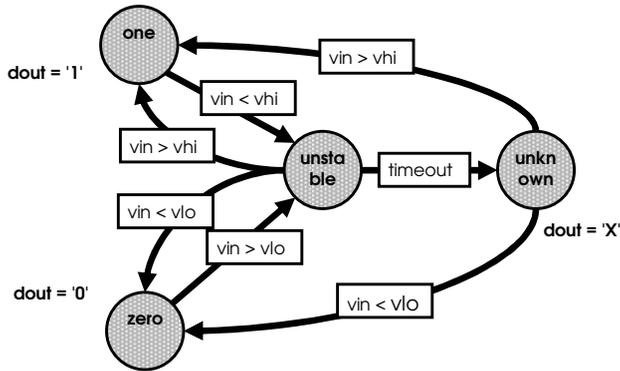*Figure 2.1.* Ideal a2d Conversion Model

Figure 2.2.   FSM For Ideal a2d Conversion Models

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.electrical_systems.all;
entity dac is
   generic (vlo : REAL := 0.2;
         vx : REAL := 2.5;
         vhi : REAL := 4.8;
         ron : REAL := 0.1;
         rwk : REAL := 1.0e4;
         rof : REAL := 1.0e9;
         tt : REAL := 1.0e-9)
   port ( signal din: in std_logic;
         terminal aout: electrical);
end entity dac;
architecture ideal of dac is
   type rt is array(std_logic) of REAL;
   constant r_table: rt := (ron, ron, ron, ron, rof, rwk, rwk, rwk, rof);
   constant v_table: rt := (vx, vx, vlo, vhi, vx, vx, vlo, vhi, vx);
   quantity vout across iout through aout;
   signal r, v: REAL;
begin
   r <= r_table(din);
   v <= v_table(din);
   vout == v'ramp(tt) + iout * r'ramp(tt);
end architecture ideal;
```

Figure 2.3.   Ideal d2a Conversion Model

3.1 Netlisting

A common approach to automatic insertion of conversion models is based on extending netlisting tools in a schematic-based design environment. A schematics-based design is one captured and maintained using a schematic entry system. The schematics database is the master representation of the design. A netlister converts the information in the schematic database to an HDL representation. The netlister may insert conversion models automatically in the generated HDL source code. Often, annotation conventions in the form of global, sheet, symbol, and wire properties may be defined to drive the netlister's choice of conversion models, actual parameters, and location of insertion. An example of a robust implementation is the Synopsys Saber® Designer product. The inherent limitation here is that the master representation of the design must be in the schematics database in its entirety. Netlisting cannot insert conversions within portions of the design described in the HDL.

3.2 Verilog-AMS

The second approach involves insertion of conversion models during the elaboration phase of an HDL simulator. It is the more general and favored approach and applies equally well to schematics-based and HDL-based design. In this approach, the identification of conversion models, definition of mixed nets that require them, and specific locations for insertion are driven by features supported by the HDL. Verilog-AMS is the first AMS HDL with such features, and the discussion uses the terminology of this paper with Verilog-AMS terminology in parenthesis.

Verilog-AMS provides a straightforward mechanism to define a conversion model (connect module). It is distinct from a normal model (module) and relies on imposing a direction to a port of a continuous nature (discipline). The selection of a conversion model is driven by the explicit declaration of connect rules, and a mechanism exists to bind parameter actuals to the model in that declaration. These rules allow specifying conversion models between continuous and discrete disciplines. Conversion between signal flow and conservative disciplines does not require a conversion model; the meaning of such connections is defined by the discipline compatibility rules. The conversion model insertion between continuous and discrete disciplines can be configured to insert one model for all connections to/from discrete ports (merged rule) or one model per port (split rule). This capability to insert one model for all connections depends on the ability of a conversion model to look outside itself at the fanin/fanout of a port.

There are some lessons to learn. One shortcoming of the Verilog-AMS approach is due to its lack of strong typing in the base language. It is essential for a user to understand the nature of every object (net) connected together in a hi-

erarchical net (signal) to understand the impact of automatic conversion model insertion. Since it is possible to declare objects which are not strongly typed with respect to their nature (discipline), Verilog-AMS provides a capability to impose (force) a nature (discipline) on such objects. However, this discipline resolution is complex to understand and allows one to coerce relationships that may not make sense. It may lead to automatic model insertion in a manner not related to domain conversion.

More significant issues exist with the definition of the discipline resolution algorithm. There are two different algorithms that may be used by an implementation, basic and detailed, each allowing the insertion of conversion models that depend on looking outside the model at the fanin/fanout of a port for reasonable accuracy. (The basic algorithm virtually requires it.) This is a powerful feature, but not required to produce an accurate model of the mixed net (signal). A tool may support either algorithm, although they produce significantly different results. Therefore, Verilog-AMS designs that employ automatic insertion of conversion models are not portable.

4.   Automatic Insertion of Conversion Models in VHDL-AMS

The following outline proposes new language features for VHDL-AMS structural modeling and automatic insertion of conversion models. Broad requirements are stated and the structural wire is introduced. Mixed nets may be constructed with wires with good semantics for all connections. User configured conversion models are inserted automatically where needed during elaboration. Open issues are noted.

4.1   Requirements

We believe that the following requirements must be met to provide robust support in the language for structural composition of designs containing instances of models at various abstraction levels.

1 Ability to configure a design with versions of components that differ in modeling domain, but represent the same device, easily.

2 Ability to structurally connect to ports of such components such that equivalent ports do not have to be re-connected, when models of different domains are swapped in.

3 Ability to automatically insert conversion models between domains to preserve an accurate representation of the design

4 Ability to model conversion at various abstraction levels (e.g. ideal, simple mos, detailed mos, etc.)

5 Ability to specify additional interface elements of a conversion model to be connected to model things like power supply accurately.

6 Ability to configure precisely and succinctly what conversion model is instantiated at each instance of a mixed connection in the design.

7 Preserve all VHDL-AMS semantics for strong type and nature checking.

The first requirement is outside the scope of conversion model insertion, but has a close relationship to it. With the current language definition, the instantiation statement must be rewritten when its component interface changes, which is what happens across domains.

4.2   The Structural Wire

The first feature needed for robust support of design composition provides the ability to create mixed nets while preserving the strong typing of the language. We propose to introduce a structural wire as a new object class into the language. The kinds of objects of interest are the terminal, the quantity, the signal and the wire. A wire is declared and may be used to connect to a port of a model. The corresponding port formal may be any object (i.e., signal, quantity, terminal, or another wire). While the other object classes have a specific subtype or subnature, a wire is purely structural and one is allowed to connect a wire to anything. A wire does have a shape. The concept of its shape refers to whether it is a scalar or a composite wire. A composite wire may be an array or a record. The declaration of the shape of a wire has same flexibility for arrays and records as subtypes and subnatures, as it builds on the current language features cleanly: one thinks of declaring a wire to be of the same shape as an existing subtype or subnature. The elements of a wire are named in a similar fashion as the elements of an object of the associated subtype or subnature: indexed names and selected names whose prefix is a composite wire are supported. A subelement of a wire is a wire.

The proposed language extensions to support wires include the semantics of wires and shapes, two attribute names that define the shape of a type or nature, the syntax of a wire declaration, and trivial enhancements to the object and interface declarations to include wires. The relevant new syntax elements are as follows:

T'SHAPE
   Kind: Shape.
   Prefix: Any type denoted by the static name T.
   Result: The shape of the type denoted by T.
N'SHAPE
   Kind: Shape.

Prefix: Any nature denoted by the static name N.
Result: The shape of the nature denoted by N.
wire_declaration ::=
  wire identifier_list : shape_indication ;
interface_wire_declaration ::=
  wire identifier_list : shape_indication
shape_indication ::=
  type_mark ' SHAPE | nature_mark ' SHAPE

4.3   The Behavioral View of a Wire

There is a need to reference a wire as if it were a signal, quantity, or terminal in behavioral code, but that is not allowed due to the strong typing of the language. One workaround is to re-factor the design to always isolate the behavioral code from its structural interfaces at the block interface level. But that is as onerous as manual insertion of conversion models!

We believe that a better approach is to provide such access to a wire through the concept of a view of a wire. A wire view specifies sufficient information about the class, typing, and mode of access to satisfy all strong typing rules of VHDL-AMS. In effect, one is saying this wire is viewed as an object of the desired type or nature. Of course, this may ultimately imply automatic insertion of an appropriate conversion model in the block where the wire view is referenced. The proposed definitions for wire views are as follows:

W'TERMINAL(N)
  Kind: Terminal.
  Prefix: Any wire denoted by the static name W.
  Parameter: A nature mark denoted by the name N.
  Result nature: The nature defined by the nature mark N.
  Result: A terminal whose nature is N.
  Restrictions: N'SHAPE must match the shape of W
W'QUANTITY(T, mode)
  Kind: Quantity.
  Prefix: Any wire denoted by the static name W.
  Parameters: T: A type mark denoted by the name T.
       mode: The mode specifying how the quantity defined
       by the wire view is used. Must be either **in** or **out**.
  Result type: The type defined by the type mark T.
  Result: A quantity whose type is T and whose mode is
       as specified.
  Restrictions: T'SHAPE must match the shape of W
W'SIGNAL(T, mode)

Kind: Signal.
Prefix: Any wire denoted by the static name W.
Parameters: T: A type mark denoted by the name T.
                 mode: The mode specifying how the signal defined by
                 the wire view is used. Must be **in**, **out**, **inout**, or **buffer**.
Result type: The type defined by the type mark T.
Result: A signal whose type is T and whose mode is as specified
Restrictions: T'SHAPE must match the shape of W

4.4   The Elaborated Model of the Mixed Net

The wire object forms a part of a mixed net. When a VHDL-AMS design is elaborated, the mixed net is elaborated. After semantic checks, a simulatable model will be produced, complete with automatically inserted conversion models wherever needed.

The elaboration of a mixed net involves:

1 Insertion of wire views at each port association element where either the formal or the actual, but not both, is a wire

2 Overall classification of the mixed net, which determines how it is modeled.

3 Determining the specific type or nature of each of its wires.

4 Mode propagation and semantic checks of each connection to determine validity

5 Binding of conversion models to each wire view

In a first step to classify a wire implementing a mixed net, each wire that is associated with an actual or formal that is not a wire is replaced by a wire view. The object class, type or nature, and mode are obtained from the object associated with the wire. After this replacement has been made, each connection has a formal and an actual that match in object class and type or nature, thereby satisfying the strong typing of the language.

In the second step, each wire is converted to a terminal, a quantity, or a signal. If any wire view anywhere in the mixed net is a terminal, the entire mixed net will be classified as a node. If not, but if there is a wire view that is a quantity, the mixed will be classified as a quantity net. There are rules governing the formulation of quantity nets to account for solvability. Finally, if all wire views are signals, the entire mixed net will be classified as a signal net. In either case, the nature of the node or the type of the quantity or signal net is obtained from the appropriate wire views. The result of following these precedence rules

is to create a net that preserves the accuracy of the connected objects. Wire conversion fails if a mixed net has incompatible wire views, for example two wire views that are terminals of different natures, or two wire views that are quantities of different types. In other words, automatically inserted conversion models may only serve to convert between different domains. They are not a back door to subvert strong typing.

When a mixed net has been classified to be of a particular class and type or nature, it is elaborated as if it were a net of that class and type or nature. If it is a quantity net or a signal net, each formal port that was converted from a wire must be given a mode. The mode is determined using the modes of all wire views of this port and of all formal ports with which this port is associated as an actual. The rules guarantee that the language rules about modes at a connection are satisfied.

The result of these steps is a consistent implementation of each mixed net at the accuracy requested, and clearly identified locations where conversion models must be inserted. These locations are the locations of the wire views. Since the properties of the converted wire are known as well as the properties of the other end of each wire view, we have enough information to bind one representative of a collection of conversion models to each wire view. Of course, if the two ends of a wire view are type or nature compatible, no conversion is needed.

4.5   Automatic Conversion Models and Wire Configuration Rules

The remaining issue is the specification of a particular entity/architecture for each instance of a conversion model and its proper instantiation. A *wire configuration specification* identifies a collection or a class of wires and associates binding information with the wire views of these wires. The wires may appear in the port association list or the declarative region of the block in which the wire configuration specification appears and any block nested within the block.

wire_configuration_specification ::=
    for *wire*_object_specification
        { conversion_specification }
    end for ;
object_specification ::=
    terminal name_list : nature_mark
    | **quantity** name_list : [**in** | **out**] type_mark
    | **signal** name_list : [mode] type_mark
name_list ::=
    simple_name { , simple_name }
    | **others**
    | **all**

A *conversion specification* associates binding information with wire views of the wires identified by the enclosing wire configuration specification.

conversion_specification ::=
    for object_specification binding_indication ;

The binding indication of a conversion specification supports binding any entity/architecture pair with a wire view specified by the combination of its object specification and wire object specification of the enclosing wire configuration specification. For an a2d, d2a or bidirectional conversion model, this includes architectures converting between the reference quantity or the contribution quantity of the terminal involved and the object at the other end of the conversion model. Additionally, the generic map and port map of the binding indication provide the means to associate the formal arguments and ports of the conversion model specified by the binding indication with actual arguments and ports. Semantically, the existing definition of a binding indication must be extended slightly.

Wire configuration specifications may appear anywhere a configuration specification may appear, and additionally in the declarative region of an entity. They may also be separately specified in a configuration declaration. A wire configuration specification applies to the elaboration of the region in which it has been declared and in the sub hierarchy of the design rooted at that region, unless it is superseded by a more specific rule. (It is a detail to state carefully that there is a similar mapping of a rule declared in a block of configuration declaration to sub hierarchies of the design.). A wire configuration specification supersedes a prior rule if it specifies the same wire view, but appears lower in the design hierarchy. There are other possible ways to map rules to the design hierarchy, but that is a usability issue we don't discuss here.

4.6   Examples

Digital and analog implementation of inverter model using explicit wire views:

    library ieee;
    use ieee.electrical_systems.**all**;
    entity inverter **is**
        port (**wire** input, output: REAL'SHAPE;
          terminal supply: electrical);
    end entity inverter;
    architecture digital **of** inverter **is**
    begin
        output'SIGNAL(BIT, **out**) <= **not** input'SIGNAL(BIT,**in**);

```
        end architecture digital;
        architecture analog of inverter is
              quantity vin across input'TERMINAL(electrical);
              quantity vout across iout through output'TERMINAL(electrical);
              quantity vcc across supply;
        begin
              vout == vcc – vin;
        end architecture analog;
    Wire configurations:
        for terminal w1, w2: electrical
              – named wires converted to terminals with nature electrical
              for signal all: in std_logic use entity work.mosa2d
                 port map (a=>wire, d=>signal);
        end for;
        for terminal others: electrical
              – other wires converted to terminals with nature electrical
              for signal all: in std_logic use entity work.a2d
                 port map (a=>wire, d=>signal);
              for signal all: out std_logic use entity work.d2a
                 port map (d=>signal, a=>wire);
              for quantity all: in REAL use entity work.tq
                 port map (a=>wire, q=>quantity);
        end for;
```

4.7   Open Issues

There are open issues at several levels. Conceptually, we believe there is insufficient information if a wire as a formal is assiciated with a quantity or signal as an actual and the wire is converted to a node. In this situation, no information is available as to what the mode of the corresponding wire view should be. It is unresolved whether it is possible to support a connection association element whose formal is a wire and whose actual is not a wire. Definitionally, we have not worked out the semantics to make a wire configuration specification applicable across a sub hierarchy of a design. The definition of the elaboration semantics that involve the steps after a wire has been classified as either a terminal, or a quantity, or a signal, are incomplete. Initialization of a net needs further analysis. Many other places in the LRM need minor changes to support the described functionality; these places have not been identified.

We chose a mixed net partitioning model for accuracy and rejected additional requirements that may improve performance by reducing conversion model count. The complexity in language definition as well as for user, model writer, and simulator implementor is judged not to be worth the potential gain.

Perhaps there are some important use cases that we are overlooking. In any case, adding support for such a partitioning strategy is not likely to invalidate any of the proposed language changes, only to extend them.

5. Conclusion

There is need to support structural design methodologies in mixed-signal modeling that lead to the creation of mixed nets. VHDL-AMS is effective at describing a wide range of mixed systems, but structural decomposition or bottom-up composition of mixed-signal components is not well supported. In particular, connections across domains, that is, mixed nets, are not allowed. The proposed language extensions provide the needed support for these methodologies using the concept of a wire and automatically inserted conversion models. It is possible to design good conversion models to effectively balance accuracy and performance of the mixed net. The wire object class adds important structural flexibility to the language while, through the wire resolution rules, preserving the strong semantics of the language type and nature system. Rule-based automatic conversion model insertion supports accuracy with very fine discrimination of what conversion model to use in any mixed connection, yet makes it very simple to generalize about model choice. Overall, there is a good opportunity to improve VHDL-AMS to support the re-configuration of mixed-signal systems effectively. The authors are working through a formal language change proposal for VHDL-AMS and welcome feedback.

References

P. Ashenden, G. Peterson, D. Teegarden: *The System Designer's Guide to VHDL-AMS.* Morgan-Kaufman Publishers; 2003.

E. Christen, K.Bakalar, A.M. Dewey, E. Moser: *Analog and Mixed-Signal Modeling Using the VHDL-AMS Language*; Tutorial at 36th Design Automation Conference, 1999

IEEE Std. 1076.1 - 1999 IEEE Standard VHDL Analog and Mixed-Signal Extensions

Verilog-AMS language Reference Manual, Version 2.0. Open Verilog International; February, 2000.

Springer