Take AIM! Introducing the Analog Information Model

Chuck McClish
Staff Verification Engineer
Microchip Technology Inc.
2355 W. Chandler Blvd.
Chandler, AZ 85224
(480) 792-7737
charles.mcclish@microchip.com

***Abstract*- Developing, integrating, and debugging analog behavioral models written in SystemVerilog is very challenging due to the complex nature of the problem. In addition, today's value transport mechanisms such as User Defined Nettypes (UDNs), Unified Power Format (UPF) supply nettypes, and legacy wreals are not interoperable with each other. This paper outlines a system called the Analog Information Model to improve the current state of analog behavioral modelling in SystemVerilog by adding a level of abstraction between the value transport mechanism and the model implementation.**

I. Introduction

Real number modelling (or RNM) is a useful method to model complex analog behavior in a purely discrete time domain simulation. RNM quantizes analog signal information into one or more properties such as voltage, current, and impedance. This has many speed advantages over Verilog-A, Verilog-AMS, and cosimulation methodologies that utilize much slower continuous time domain simulators. However, RNM has been implemented in SystemVerilog using various types such as the User Defined Nettype (UDN)[1], Unified Power Format (UPF)[2] supply_net_type, and wreal type from Verilog-AMS.

These types are not directly interoperable among themselves. This can be very challenging to the model developer as it can require a certain model behavior to be written multiple times to account for these different value transport schemes. The system integrator also encounters issues when attempting to connect these different types together in a top level system and ensure consistency between various blocks.

What is needed is a level of abstraction between the RNM value transport mechanism and the model implementation itself. As a follow-up to the author's DVCon 2020 and 2021 papers [3][4], this paper documents an implementation of this abstraction layer called the Analog Information Model or AIM[1]. AIM's goal is to use the design's inherent connectivity to associate different ports, like UDNs, but have the capability to handle various port types simultaneously.

II. AIM Nodes and Connectors

AIM is based on the concept of nodes and connectors. Nodes are single terminal modules while connectors have 2 terminals:

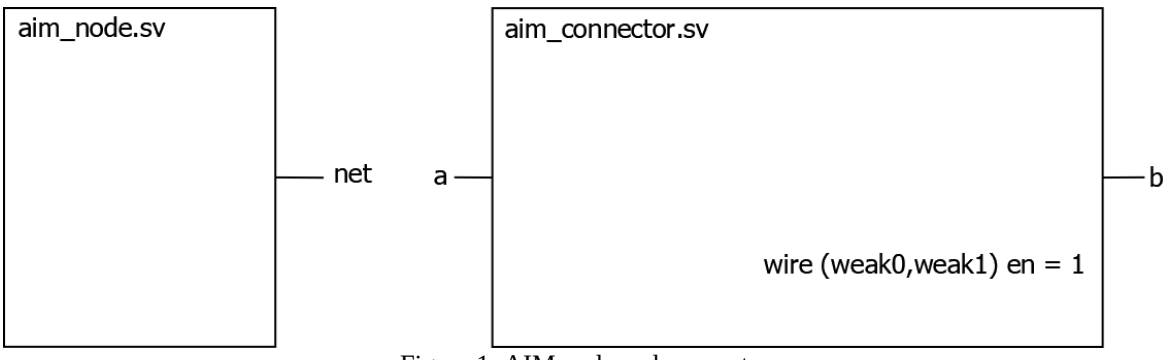

Figure 1: AIM node and connector

[1] Patent Pending.

A node is considered an endpoint. It can be either a driver, receiver, or bidirectional. Various flavors of nodes are included to connect to different data types, such as a Verilog wire, UPF::supply_net_type input and output, wreal, etc.

A connector is essentially a wrapper for 2 nodes. Different connectors exist to act as a bridge between different types, such as a simple verilog wire and UPF::supply_net_type, or to connect ports of the same type. Connectors have an internal 'en' variable driven by a weak 1 that, by default, connect the 2 sides together. Since the 'en' variable is weakly driven, a hierarchical assignment to 1'b0 can be used to disconnect the 2 sides. This enables the connector to simultaneously function as a tran or tranif gate as well as a data type converter.

Each port on a behavioral model that conveys analog information is considered a node. Inside of the behavioral model, the port is directly connected to a node or connector. The type of node or connector used is determined by the data type of the port being connected. In the larger system, behavioral models are wired together through the digital hierarchy:

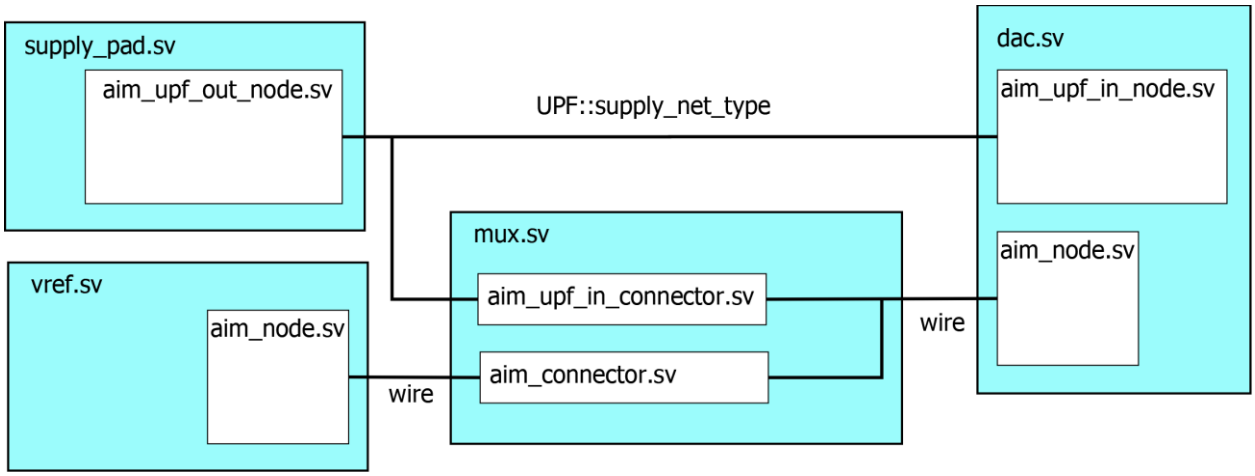

Figure 2: AIM nodes and connectors used in a system

Up to this point, we've described how AIM nodes and connectors interface to each other, but how does the user interact with these things? Each node in the design has various properties describing the state of a node as well as methods to modify these properties:

```
module aim_node import aim_pkg::*; (inout wire net);

  // Properties
  bit    is_ok;
  real   voltage;
  real   current;
  real   resistance;
  node_t node_type; //From aim_pkg

  // Methods (psuedo code prototype declarations)
  task automatic set_voltage(real v);
  task automatic set_current(real v);
  task automatic set_resistance(real v);
  task automatic set_node_type(node_t node_type);
endmodule
```

Figure 3: AIM node properties and methods

Reading and modifying node properties is as simple as monitoring the property of interest or calling the associated 'set_*()' method.

Unlike nodes, connectors do not have associated methods to modify its internal properties. This is because connectors act as simple bi-directional switches or pass gates, only transmitting values through and not modifying them. However, voltage on the connector ports and current through the connector can be accessed by viewing the relevant connector properties

But wait a minute, how does a simple Verilog wire transport various real properties to other nodes? How is information not lost transitioning through various data types in connectors? What are these node_type and is_ok properties? Let's take a look under the hood.

III. AIM Under the Hood

AIM consists of a library of SystemVerilog classes and modules. As described in the previous section, the core components of AIM are the AIM node modules and AIM connector modules. Upon instantiation, an AIM class object and configuration object are created in each AIM node module. Each AIM connector creates 2 class objects, one for each port, and a single configuration object. The properties and methods described by the module are really aliases to the properties and methods in the AIM class object:

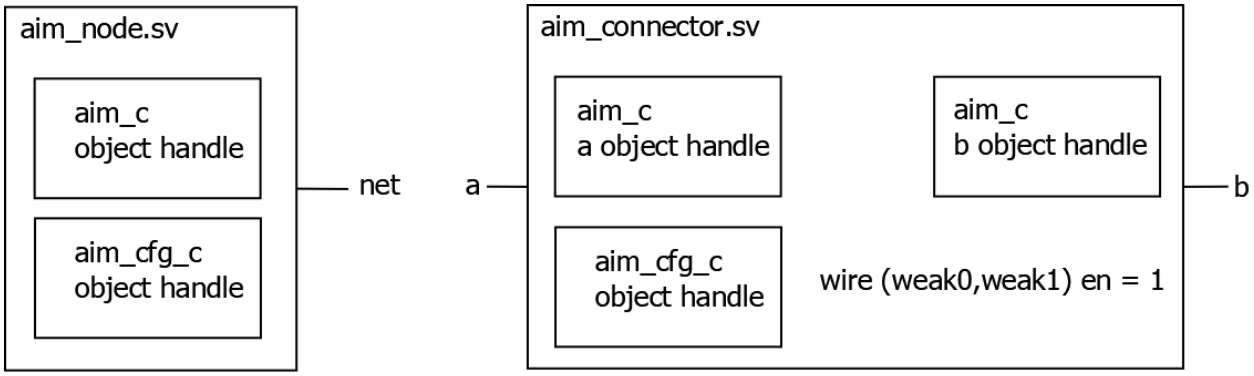

Figure 4: AIM nodes and connectors with their associated class object and configuration object

The AIM configuration object contains node configuration information, such as minimum, nominal, and maximum values for the AIM class object properties. Using the minimum and maximum ranges specified in the configuration object, the check_ok method is used to specify the AIM object 'is_ok' property. This single bit value allows any downstream code to easily and consistently check if a node is operating within valid operational parameters:

```
// Inside the aim_node module
aim_cfg_c cfg = new($sformatf("%m"));
function automatic bit check_ok();
  check_ok = 0;
  // If min and max values fall outside the specified range, return with a value of 1
  // check_ok() method called a the end of resolution and assigns return to 'is_ok' property
  if (!(voltage inside {[cfg.min_voltage:cfg.max_voltage]})) return 1;
  if (!(current inside {[cfg.min_current:cfg.max_current]})) return 1;
endfunction

// Inside of a model using the node 'is_ok' properties to enable the block
assign model_en = enable & vdd_node.is_ok & vss_node.is_ok & vref_node.is_ok;
```

Figure 5: AIM check_ok method and 'is_ok' property usage

The AIM class also contains several static properties and tasks. First is the associative array aim_nodes. This array contains all of the AIM class objects in the design. Each AIM module constructs an AIM class object and places a handle into the aim_nodes[0] slice indexed by its %m path:

```
class aim_c;
  static aim_c aim_nodes[int][string];
  function new(string node_name);
    this.node_name          = node_name;
    aim_nodes[0][node_name] = this;
  endfunction
endclass
```

Figure 6: aim_nodes associative array declaration and aim_c constructor

The static aim_init task performs the magic of associating these class objects together. Figure 7 is provided below is annotations illustrating initialization algorithm steps:

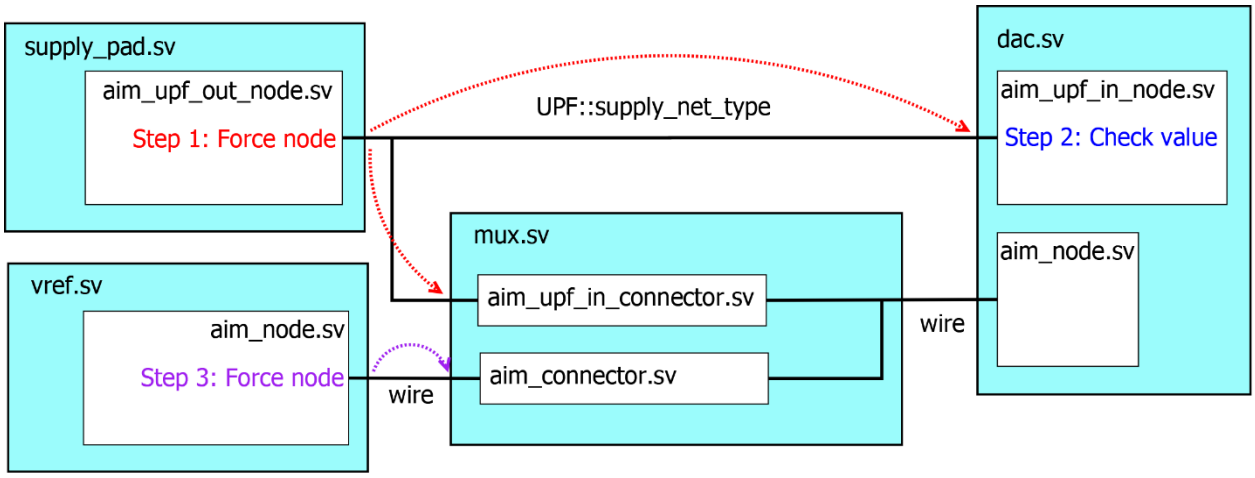

Figure 7: AIM initialization routine

When called at time 0, this task will force the port of the first aim_nodes[0] object associated AIM module and place a handle of that object in aim_nodes[1] (step 1). Note that the type of AIM node determines what is forced. For the aim_node module, a wire is used as the port connection, so the value forced is 1'b1. For the aim_upf_out_node module, the data type is UPF::supply_net_type, so the value forced on this node is {FULL_ON, 32'd100_000}. The task will then walk through all of the remaining aim_nodes[0] nodes and determine if the forced value was seen on each of the AIM modules. Any module which sees the forced value, must be connected through the design hierarchy and the associated class object handle is placed in aim_nodes[1] (step 2). After all objects have been checked, the node which forced its value in step 1 is released, and the next unconnected AIM object has its associated module port forced and its handle added to aim_nodes[2] (step 3). This process continues until all AIM objects are associated with each other.

Once initialization is complete, the aim_nodes array contains the association of all AIM class objects in a design. Objects that share the same non-zero indexed slice are on the same net. Assuming the module instance names are suffixed with an '_0', the resulting aim_nodes array can be visualized as follows:

```
aim_nodes
          [0]
             [“top.supply_pad_0.vdd”]
             [“top.dac_0.vdd”]
             [“top.mux_0.sel_0”]
             [“top.vref_0.vref_out”]
             [“top.mux_0.sel_1”]
             [“top.mux_0.out”]
             [“top.dac_0.vref”]

          [1]
             [“top.supply_pad_0.vdd”]
             [“top.dac_0.vdd”]
             [“top.mux_0.sel_0”]
          [2]
             [“top.vref_0.vref_out”]
             [“top.mux_0.sel_1”]
          [3]
             [“top.mux_0.out”]
             [“top.dac_0.vref”]
```

Figure 8: A populated aim_nodes array

After initialization, the AIM module port is no longer required by AIM itself. All properties are passed between associated AIM class objects directly. The Verilog wire based AIM node and connector leave these pins floating. In the case of UPF ports for example, there is a desire to have the UPF state and voltage propagate through the UPF supply network for verification and debug purposes. In this case, the UPF AIM node and connectors push the value out onto the UPF network:

```
module aim_upf_out_node import aim_pkg::*; import UPF::*;
  (output UPF::supply_net_type net);

  // Properties
  bit    is_ok;
  real   voltage;
  real   current;
  real   resistance;
  node_t node_type; //From aim_pkg

  // Methods (psuedo code prototype declarations)
  task automatic set_voltage(real v);
  task automatic set_current(real v);
  task automatic set_resistance(real v);
  task automatic set_node_type(node_t node_type);

  // convert from real to int scaled in microvolts for UPF
  function automatic int signed real2upf (real voltage);
    real2upf = $rtoi(voltage * (10**6));
  endfunction

  initial fork
    // aim.net_resolved is an event triggered by the resolution function after completion
    forever @(posedge aim.net_resolved) begin
      if (aim.resistance < aim_pkg::HIGHZ) //HIGHZ == 100MOhm (floating)
        net <= '{UPF::FULL_ON, real2upf(aim.voltage + aim.current*aim.resistance)};
      else
        net <= '{UPF::OFF, real2upf(aim.voltage + aim.current*aim.resistance)};
    end
  join_none

endmodule
```

Figure 9: AIM UPF output node

Figure 9 illustrates another important point of AIM, which is all AIM node modules convert the port type to the AIM Thevenin type (i.e. voltage, current, resistance, and type) for resolution. For the case of UPF, there is no reliance on VCTs or the UPF information model. All the information required for resolution is in the UPF supply connections and values assigned therein.

IV. Value Resolution Methodology

Like any RNM system, a value conflict resolution system is required. AIM class objects contain voltage, current, and resistance properties. Each time any one of these properties change in an AIM class object, new values must be resolved for all associated AIM class objects. In addition, the state of bidirectional switches on the net affect how net segments are merged or disconnected. Using the voltage, current, and resistance properties by themselves requires that Kirchhoff's Voltage Law (KVL) and Kirchhoff's Current Law (KCL) are utilized. Solving this list of linear equations is no picnic, especially in pure SystemVerilog. There must be an easier way.

Looking at the various circuits employed in our microcontrollers, it was apparent that nearly all analog nodes in the design could be modelled as voltage sources, current sources, loads, and switches. All these components are connected in parallel to each other.:

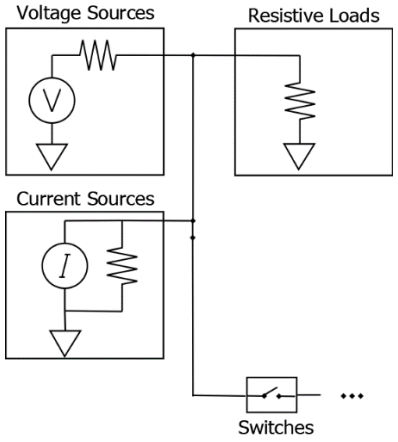

Figure 10: Analog component topology

In addition, all analog nodes reference the "chip" ground. In practice, ground is not one single net, nor is it equipotential across all nodes on a net, however, for the accuracy we required, this assumption greatly simplified our circuit description. The resulting schematic for any collection of AIM class objects looks like this:

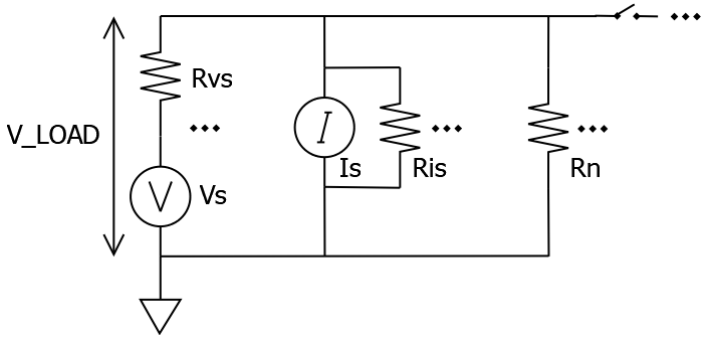

Figure 11: Analog component schematic

During the first phase of resolution, all AIM switches connected to the net are resolved. If enabled, the net segments on either side of the switch are effectively merged for the resolution. If disabled, all nodes on the other side of the switch are ignored. In other words, switches have no effect on the mathematical calculation, they are simply used to connect or disconnect AIM net segments together during resolution. This leaves only source and load components.

With all parallel components, we can leverage Millman's Theorem[5] to simplify the equation even further. This theorem simplifies this topology to a single Thevenin voltage and resistance along with a load resistance:

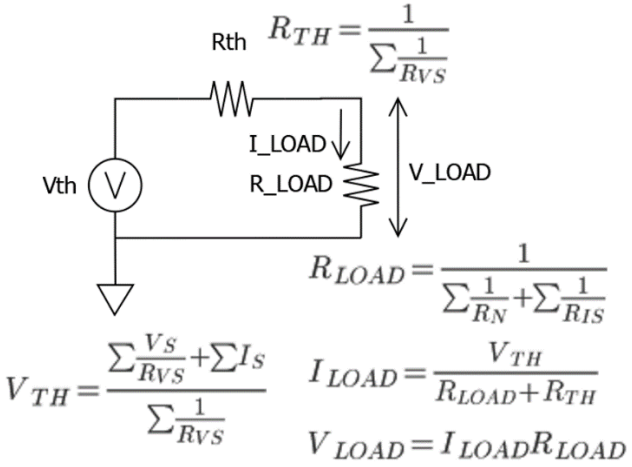

Figure 12: Millman's theorem applied

Once the load voltage is calculated, this is applied to the voltage property in all associated AIM class objects, as they are all in parallel. Then the individual branch currents can be calculated and applied to the associated AIM class objects:

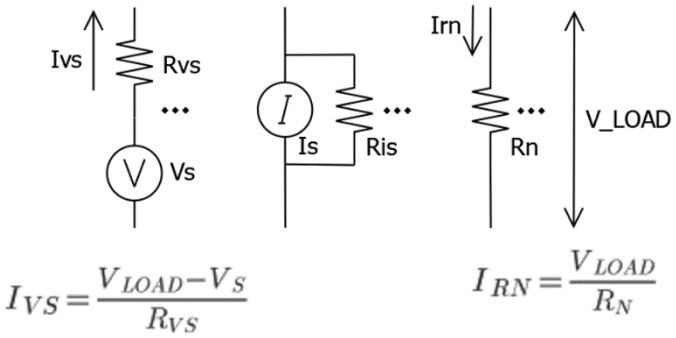

Figure 13: Solving branch currents

Now to model in SystemVerilog. Up to this point, only circuit properties have been described in the AIM class objects. To use Millman's theorem, the component node_type is required so that each node is plugged into the equation properly. AIM class objects have an enum property called node_type which specifies a node as a V_SOURCE, I_SOURCE, LOAD, or SWITCH.

This methodology works well for systems with nodes connected in parallel, referencing a common ground tied to 0V, and with no feedback. This covers many of the scenarios encountered. However, AIM has some flexibility here by implementing the resolution as a class. If Millman's theorem is found to be insufficient, the resolution class can be extended and the resolution method overridden for the project's needs.

V. Truly Mixed Signal Ports

Verilog wire based ports are the preferred datatype to use for AIM. The main reason is that a 'wire' is easy for an integration engineer to understand. Simply connect the points together and you're done. In addition, because AIM only uses the port during initialization, the rest of simulation time is available for purely digital purposes. This is extremely useful in things like mixed signal pads where a single connection point must be capable of transmitting both analog information and purely digital information bidirectionally:

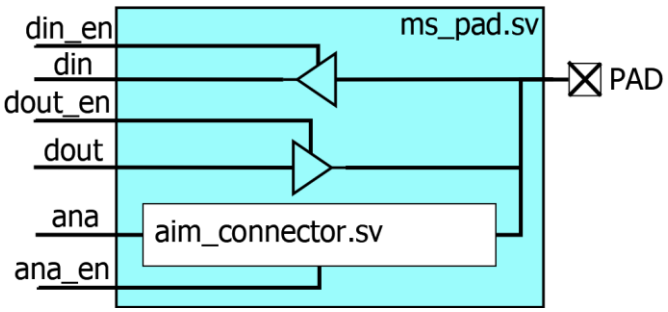

Figure 14: Simple mixed signal pad setup using AIM

Using this setup, digital signals pass through the pad into the system without any extra work. As a bonus, the specify block functionality is undisturbed, so all digital timing information through the pad is preserved.

If this were to be done purely using UDNs, several layers of converters are necessary to handle this operation because UDNs require a single type to be used. So for the digital functionality, a wire-to-UDN and UDN-to-wire conversion needs to take place somewhere. Based on how this is performed affects how a specify block is handled, which depending on the implementation, may no longer be possible.

VI. Enhanced UPF Simulation

Much like mixed signal ports, UPF also benefits from AIM ignoring the pin after initialization.[2] In the case of UPF, the initialization process forces values onto the UPF supply network and uses that connectivity information to associate AIM nodes. Using connectors, it is possible to connect UPF ports and wires together seamlessly.

Since AIM has current information and connects UPF:supply_net_type ports to other AIM nets, it is possible to perform additional functional checks that were not possible before. For example, a common failure that is hard to detect with classic UPF simulations is overcurrent checks. If there is a regulator driving a power domain, each load on that net draws a certain amount of current depending on its operational state. With AIM, if the current consumed by the loads is too high, the 'is_ok' property in the regulator node will drop, which the model developer can use to trigger assertions, kill the regulator output, etc. This enables the system integrator to perform a power sanity check for these types of failures very early on in the design process.

AIM can also be used to model load of digital hierarchies. To do this, a module is created and bound to the DUT hierarchy to be modelled. Inside this module, an AIM UPF node is created. The UPF file then connects the pin of this node to the appropriate UPF supply. Since the module is bound to the digital hierarchy, it is very easy to hierarchically reference signals in the DUT to modify the impedance based on device operation. By creating companion load models for each of the digital hierarchies in question, it is easy for the user to model power consumption of a larger system in a modular, flexible way at the desired level of accuracy:

dig.sv

```
always @(en)
  if (!en)
    // in reset
  else
    // running
```

dig_load.sv

```
UPF::supply_net_type vdd;
aim_upf_in_node node(vdd);
always @(dig.en)
  if (!en) node.resistance = 10e6;
  else     node.resistance = 1e6;

bind dig dig_load load();
```

top.upf

```
connect_supply_net VDD -ports top/dig_0/load/vdd
connect_supply_net VDD -ports top/ana_0/vreg/vdd_out
```

Figure 15: Digital load model

[2] To clarify, values calculated in AIM are pushed onto the UPF port, but values are not read back from the UPF network. To AIM, whatever is on the UPF network is of no consequence.

Additionally, the model developer doesn't need to care if the power connection comes from UPF or from a simple wire. AIM obfuscates the data type used at the connection point so that the developer only needs to check the node's properties and uses the common methods. This is easier for the model developer (since they only need to model behavior once) but also give the integration engineer more flexibility. Some power tests can be performed without a UPF file at all by connecting the power pins in a separate file or with power pins in the design itself. Since UPF simulations typically run much slower than normal digital simulations and require additional licenses, there are cases where this approach is very beneficial from a development standpoint.

VII. SPICE COSIMULATION SUPPORT

Analog cosimulation is required at both the system and IP level to verify that the SystemVerilog behavioral models are accurately representing the behavior of the physical device. Analog cosimulation requires the use of electrical-to-RNM and RNM-to-electrical elements to handle the transition between discrete and continuous time domain solvers. In AIM, the cross domain aspect is handled automatically through the use of a Verilog-AMS module:

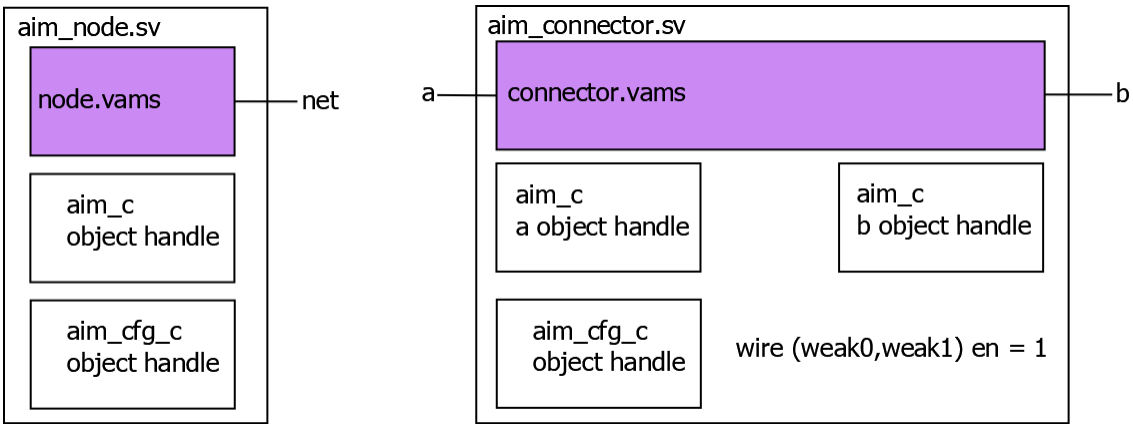

Figure 16: AIM node and connector module supporting spice cosimulation

In the aim_node, the electrical port is connected to the port of the node VAMS block itself. The aim_c class object interfaces directly with the variables inside the node VAMS instance to drive and monitor values on the electrical port. The aim_connector.sv connector VAMS module instance is simply coded as a switch in the analog block. This either connects or disconnects the two electrical ports from each other depending on the state of the 'en' variable.

The node VAMS module leverages the node_type property in the analog block to build the device. For example, if the node_type is a voltage source, the analog block models the connection as an ideal voltage source in series with a resistor. The other types are modelled as described in the following figure:

```
// 'select' assigned from the aim_node.sv module and mapped
// to the node_type enum property
wire [1:0] select;
analog begin
  case (select)
    2'b00: V(hdl_con) <+ I(hdl_con)*r_tran + voltage; // Voltage Source
    2'b01: I(hdl_con) <+ V(hdl_con)/r_tran + current; // Current Source
    2'b10: V(hdl_con) <+ I(hdl_con)*r_tran;           // Load
    2'b11: I(hdl_con) <+ 0;                           // Switch
  endcase
end
```

Figure 17: Verilog-AMS module node types

Using hierarchical references, the encapsulating AIM node module has the ability to monitor and drive the voltage, current, and resistance properties inside the Verilog-AMS module. From the user's perspective, all this happens transparently. In a behavioral model, the voltage property still reads the voltage, the set_voltage() method still sets the voltage.

Back to the AIM node configuration, the AIM node port is connected directly to the Verilog-AMS module. AIM leverages the ability of cosimulation tools to transparently connect electrical ports together through wire connectivity

in the design. Since all AIM nodes and connectors are connected to each other, all connections are electrical to electrical, requiring no interfacing elements to be inserted by the cosimulation tool. In this configuration, AIM *does* use the connectivity of the design directly. In doing so, AIM can now leverage the resolution engine in the analog simulation tool without having to resolve values itself:

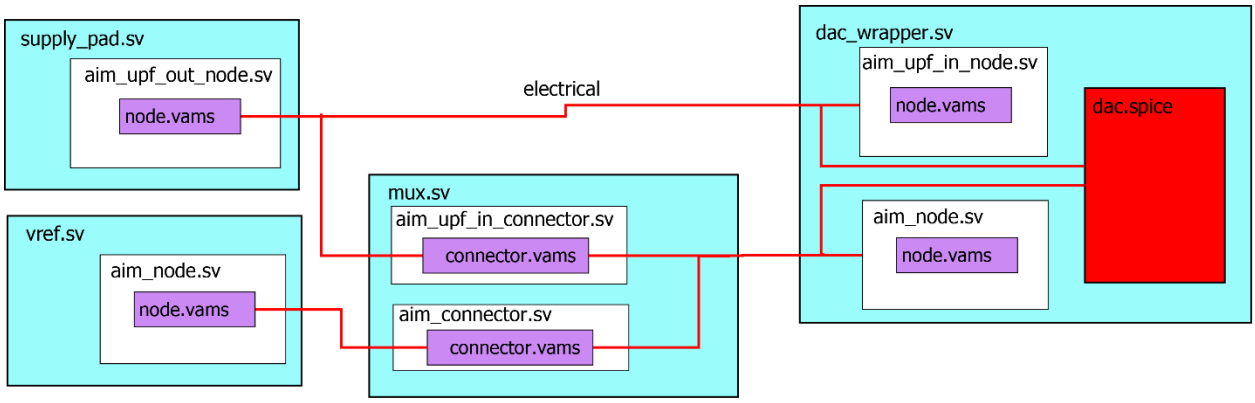

Figure 18: AIM spice cosimulation topology

When a model is swapped out with a spice block, a wrapper level is recommended to place all of the AIM node module instances. These are used to monitor the values coming out of the spice block and the methods are never called. While the resolution object does not perform any value calculations, it does record the values coming from the spice and Verilog-AMS modules and sends them to AIM. This is required so that any of the behavioral models that are not spice react appropriately to changes from any spice blocks instantiated in the design:

Using the analog solver directly has several key advantages. Firstly, there is no digital-analog-digital type handoffs, which can quickly cause convergence and feedback loop issues. Secondly, the model developer doesn't care if they are talking to a spice block or a purely digital AIM node. More importantly, the testbench doesn't care either. This enables the same digital sequences to be used to stimulate the design and the same scoreboards and assertions be used to ensure correct operation. That said, the real analog block won't *exactly* match the behavior of the digital model, but it will be close enough for a fuzzy match. That is a topic for another day (or another paper!).

VIII. SUMMARY

AIM enables a useful level of abstraction between the various RNM value transport mechanisms and the underlying model functionality. The value resolution method is contained in a class object to enable efficient value resolution of voltages, currents, and resistances. UPF and analog cosimulation support are available out of the box in a form that is easier to use by the engineer.

IX. References

[1] IEEE Std 1800-2012, "Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language"

[2] IEEE Std 1801-2018, "Standard for Design and Verification of Low-Power, Energy-Aware Electronic Systems"

[3] C. McClish, "It Should Just Work! Tips and Tricks for Creating Flexible, Vendor Agnostic Analog Behavioral Models", DVCon San Jose, March 2020

[4] C. McClish, "Bi-Directional UVM Agents and Complex Stimulus Generation for UDN and UPF Pins", DVCon San Jose, March 2021

[5] J. Millman, "A Useful Network Theorem", Proceedings of the IRE. 28 (9): 413–417, 1940