Verilog-AMS: Mixed-Signal Simulation and Cross Domain Connect Modules

Peter Frey and *Donald O'Riordan*
Cadence Design Systems, Inc.; 555 River Oaks Parkway; San Jose, CA 95134

Abstract

Verilog-AMS is one of the major mixed-signal hardware description languages on today's market. In addition to the extended capabilities to model analog and digital behavior, the language supports a novel approach to merge existing digital and analog designs without rewriting the individual designs. At the center of this approach is the connect module and the connection rules. These language features enable the designer to declare modules which can be automatically or manually inserted at an intersection of net segments with different disciplines. A mapping between different disciplines and therefore between the different domains, enhances the (re)usability of designs and enables a natural approach to mixed-signal design. Circuitry of interest can be modeled with high accuracy in the analog domain whereas less critical portions of the design are modeled in the faster but less accurate digital simulation domain. Since Verilog-AMS actively supports the mixed-signal approach, the interchange of digital and analog portions is straightforward and strongly encouraged.

The purpose of this paper is to introduce the semantics of Verilog-AMS connect modules in greater detail and illustrate the impacts and tradeoffs on the simulation performance. Closely related principles of driver-receiver segregation, discipline resolution and cross domain communication are discussed and evaluated to provide a thorough description of the extended Verilog-AMS mixed-signal simulation capabilities.

1 Introduction

Traditional designs are uni-domain: either digital or analog. With the advent of Mixed-Signal Languages like Verilog-AMS a new breed of designs incorporating digital and analog behavior are combined in a unified description language. While it is now possible to write a single mixed-signal module which contains both analog and digital behavior, it is also true that many designs are build on top of existing modules. Hence, the new mixed-signal designs often contain purely analog and digital library components. The usefulness of a mixed-signal language and simulator depends on the capability to utilize the existing designs, exchange and interchange different representations of individual components, and expand critical portions of the design with more detailed models.

Verilog-AMS addresses the cross domain issues specifically with new language constructs. Nets are clearly partitioned into either discrete (digital) or continuous (analog) domains. Depending on their domain, their behavior is simulated with a discrete event simulator or a differential equation solver. What remains is the modeling and the simulation of the interface between the different domains. Verilog-AMS introduces the connect module construct. The definition of a module was extended to provide translation modules, mapping values and timing behavior from one domain into the other. In general, a connect module provides two ports of different discipline and has a behavioral description stating the transformation process. These modules can then be manually or automatically inserted to provide an interface between nets of different disciplines. The behavioral description which implements the transformation process can be freely modeled at any arbitrary level, from "quick-and-dirty" to low-level and detailed. The language thus provides full customizability of the transformation from one domain to the other, which offers significant possibilities for tradeoffs related to both simulation accuracy and speed.

Since the connect module insertion depends on the disciplines of net segments on either side of a port, several features of Verilog-AMS contribute to determine the final simulation model. This paper describes in greater detail the features of Simulation Domains (Section 2), Discipline Resolution and Connect Module Insertion (Section 3), and Driver-Receiver Segregation (Section 4). The paper concludes with a discussion of connect module interface modeling and performance factors in Section 5.

2 Simulation Domains

Verilog-AMS captures two different simulation domains, namely the analog simulation domain and digital simulation domain. As stated earlier, these domains are mapped onto different simulation kernels which implement different simulation semantics.

In the digital simulation domain a discrete event simulation kernel calculates the discrete signal behavior. Discrete signals exhibit discrete signal changes at discrete points in time. On the other hand, the analog simulation domain is mapped onto a differential equation kernel, which calculates at discrete points in time a solution to a set of differential equations. The signals represented in the solution vector of a differential equation kernel are assumed to change continuously between the individual solutions (interpolation). As a result, time step selection depends on tolerance factors and model behavior and can not be predicted.

The difference in the simulation domains is also reflected in the domain specific semantics of the Verilog-AMS language [2]. In the discrete portion of modules (assign statements, initial block, and always blocks), time is controlled on a per block base. If a control statement (delay, event control, etc.) is encountered the whole process is stopped and scheduled at a point where the control statement is satisfied. This is the opposite in the analog block of a Verilog-AMS model. Each module may contain zero or one analog block which states a sequential description of a set of differential equations. Therefore, each time a new solution is calculated in the differential equation kernel, all equations are evaluated which implies the complete contents of the analog blocks is executed. An example of the different semantics is provided in the following pseudo module.

```
module mixed;
    electrical in;

    always @(cross(V(in)-1,1))
       ....

    analog begin
       @cross(V(in)-1,1)
         ...
       V(in) <+ sineWave;
    end
endmodule
```

Here, the always block is suspended until the event condition (a threshold cross of net in) is satisfied. However, the corresponding event condition in the analog block is evaluated similar to an if statement at any solution point and the conditional statements are executed as soon as the cross statement evaluates to true. The contribution statement to $V(in)$ is therefore executed unconditionally.

3 Connect Module Insertion

A connect module is required to be inserted into the design whenever modules with ports of different disciplines are connected. This section provides an overview of how connect modules are inserted due to the discipline resolution. The insertion can either be performed manually or automatically. In the case where a net of one discipline (say analog) connects to several ports of the same discipline (say digital) , the Verilog-AMS language provides an option of whether to insert an individual connect module at each digital port, or whether to merge all of these into a single connect module which is then connected up to each individual digital port. This option is known as the connect mode, and offers simulation speed/accuracy tradeoffs, and is explored in detail in section 3.2.

In order to perform connect module insertion, we first need to determine the disciplines of the various nets and ports in the design.

3.1 Discipline Resolution

Verilog-AMS syntax allows a discipline to be associated with a particular net by using a net discipline declaration. For example, if we assume that the discipline electrical has been previously defined (of domain continuous) and the discipline logic has been previously defined (of domain discrete), we can declare the nets e1 and e2 to be of discipline electrical, and the nets l1 and l2 to be of discipline logic using the following syntax:

```
  wire l1, l2;
  logic l1, l2;
  electrical e1, e2;
```

The first declaration declares two wires, l1 and l2. The second declaration declares that these wires are of discipline logic, which effectively declares that they are digital objects. The third declaration declares two more nets, e1 and e2, and also declares them to be of discipline electrical. Since electrical is assumed to be previously defined as having a continuous domain, it means that these nets will have their values calculated by the analog differential equation solver.

Ports can be declared to be of certain disciplines as in:

```
module foo(in,out)
  input in;
  output out;
  electrical in;
  logic out;
  ...
endmodule
```

Here, the first port of module foo is declared as an electrical port. Similarly, the second port is declared to be of the logic discipline. It is also possible to implicitly declare nets in connectivity statements, simply by using them as port connections. In this case, the disciplines of the nets are not declared by the user, and the simulator will need to resolve the disciplines of these nets. For example:

```
module top();
  foo f1(a,b);
endmodule;
```

For the two implicitly declared nets a and b, it is not immediately obvious which disciplines they should assume, and which solver will be responsible for calculating their values. However, it would seem appropriate that net a should be of discipline electrical, and net b should be of discipline logic, in order to agree with their counterparts (in and out respectively) within module foo. Further complexity arises when we consider the case of multiple levels of hierarchy. Consider the following example:

```
module top();          module fee(p,q);
  fee f1(a,b);           foo f1(p,q);
endmodule;             endmodule;
```

Here, the module foo was replaced with an instance of module fee. Net top.a is connected to top.f1.p (where f1 is an instance of fee). However, no discipline has been declared for net top.f1.p i.e. no discipline has been declared for net p within module fee. However, looking further into the hierarchy, top.f1.p is connected to top.f1.f1.in (where top.f1.f1 is now an instance of module foo) It makes sense to assign top.f1.p the same discipline as top.f1.f1.in (electrical). Propagating this information up through the hierarchy, top.a is assigned the same discipline as top.f1.p (which was resolved as electrical). This processes propagates the disciplines of lower level nets up the design hierarchy.

Propagating disciplines up the signal hierarchy can take care of resolving certain cases. Hence, in order to determine the domain of a particular net, it is a requirement to resolve the disciplines of those nets for which disciplines are not explicitly declared. Once the domain of each net is determined, the port connections are inspected to determine where connect modules need to be inserted in order to map from one domain to the other (from continuous to discrete, and vice versa). In fact, Verilog-AMS provides two discipline resolution algorithms: non-detailed, and detailed. The introduced example algorithm represented a portion of the standard *non-detailed discipline resolution algorithm*, the basic idea of which is to propagate net disciplines up the hierarchy, starting from the leaf level, until a complete ascent of the design hierarchy is complete. Note however that after the bottom-up traversal, some net disciplines may still be unresolved. For such remaining nets, the application of a user-defined default discipline is required.

The standard *detailed discipline resolution algorithm* is somewhat more complex, and can be summarized as follows. The idea behind the detailed discipline resolution algorithm is to propagate analog disciplines through as much of the design as possible, for those nets that cannot be easily be determined to be of the discrete domain. In effect, a bottom-up traversal is performed, where analog disciplines are propagated upwards. Then a top-down traversal is performed, where the disciplines of port high-connections (of known analog discipline) are propagated downward to any port low-connections (whose disciplines are still unresolved), effectively *pushing analog down*.

Since analog disciplines are pushed both up and down, the detailed discipline resolution algorithm will typically result in a design which has a higher portion of its nets resolved to analog. This results in higher accuracy (detailed) simulations, though at the likely expense of simulation speed.

It should be noted here that the user can always create a design in which the discipline of every net is predetermined, and discipline resolution is not necessary. This can be achieved by declaring the discipline of every net in the design, either by local discipline declarations, or by out-of-context discipline declarations such as

```
electrical top.f1.a;
```

Hence, the user has complete control over the final simulation model from a discipline resolution (and hence domain resolution) perspective.

3.2 Automatic Connect Module Insertion

It was previously stated that a connect module is required to be inserted into the design whenever modules with ports of different disciplines are connected in order to map the signal values from one domain to the other. The insertion can either be performed manually or automatically. This section provides an overview of how connect modules are automatically inserted due to the discipline resolution, and user-supplied connect rules.

In the case where a net of one discipline (and of one domain) connects to several ports, each of the same discipline, but of the other domain, the Verilog-AMS language provides an option of whether to insert an individual connect module at each digital port, or whether to merge all of these into a single connect module which is then connected up to each individual digital port. This option is known as the connect mode, and offers simulation speed/accuracy tradeoffs.

Consider the following example (Figure 1), in which a net W1 of discipline electrical connects to two ports of discipline logic: The dashed line represents a module instance boundary. A connect mode of *merged* results in a single connect module being inserted, which is then connected to multiple digital ports. A connect module of *split* results in multiple connect modules being inserted, one for each digital port. Hence, the merged mode results in less connect module instances than the split mode. This has several effects:

1. By having one connect module per digital port, the load of each digital port (receiver) on the analog side can be accurately represented.

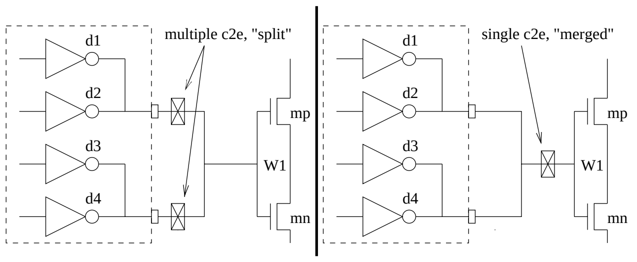

Figure 1. Merged/Split connect mode attribute

A connect mode of *split* achieves this high level of accuracy. Conversely, a connect mode of *merged* will cause only a single connect module to be inserted, so that only a single load is presented to the analog solver, thus losing accuracy in modeling fanout effects.

2. A single connect module instance may contain some locally declared analog nets, depending on what algorithm is used to map from one domain to another. (Examples will be presented later). The split mode, which results in a higher count of connect module instances, will thus result in a higher count of local analog nets. Hence, use of the split mode may result in larger systems of equations and may degrade simulation performance.

The following examples show how connect rules can be specified.

```
connectrules AMSconnect;
  connect elect_to_logic split; // split mode
  connect logic_to_elect; // defaults to merged mode
endconnectrules

connectmodule elect_to_logic(aVal, dVal);
  input aVal; electrical aVal;
  output dVal; logic dVal;
...  // body of connect module goes here
endmodule

connectmodule logic_to_elect(dVal, aVal);
  input dVal; logic dVal;
  output aVal; electrical aVal;
...  // body of connect module goes here
endmodule
```

Initially, a set of connect rules (delimited by connectrules and endconnectrules) specify that modules elect_to_logic and logic_to_elect are to be used as connect modules where appropriate. In addition, the connect mode where module elect_to_logic is used is split. The default case is merged, which applies to module logic_to_elect.

By examining the port interface of connect module elect_to_logic, we see that it has an input port aVal which is of the electrical discipline, and an output port dVal which is of the logic discipline.

Whenever the simulator detects an electrical net forming the high-connection of an input port which has a net of discipline logic as the low connection, it now knows that it needs to insert module elect_to_logic at the interface. It can determine this by looking at the port disciplines and directions for the connect modules, and comparing against the port disciplines and directions for our example in which the net of discipline electrical forms the high connection to a port of discipline logic. The only match in our example would be module elect_to_logic. Similarly, if a net of discipline logic forms the high connection to a port which has a (low connection of) the electrical discipline, and direction input, then the module logic_to_elect will be inserted.

This section explored how a *split* connect mode can achieve higher accuracy in modeling fanout/fanin effects, whereas it can also have a degrading effect on simulation performance due to introducing extra equations that need solution by the analog solver. Overall, the performance of the simulator will be a tradeoff of accuracy versus simulation speed, and will be highly influenced by the choice of discipline resolution algorithm, by the connect mode used for connect module insertion at the various port boundaries, and by the detail of modeling performed within the connect modules themselves. In addition, the user has control by specifying statements within the connect rules over what particular connect module classes get inserted at particular net/port boundaries, and thus can choose to insert connect modules of varying modeling accuracy.

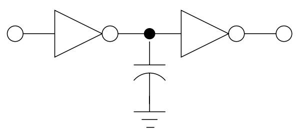

Figure 2. Capacitively loaded inverter chain

Verilog-AMS allows the user to declare the disciplines of various nets in the design using local discipline declarations, and also using out-of-context discipline declarations. By using such declarations, writing corresponding connect rules, and choosing the discipline resolution algorithm appropriately, a great deal of control and flexibility is thus available to the user in determining how many connect modules are inserted, and where. Additionally, the contents of the connect modules themselves and the level of modeling performed therein is also completely user customizable. This is due to the fact that they are user written/customized modules, and not some hardcoded inaccessible primitives. The Verilog-AMS user can thus make intelligent and wide-ranging tradeoffs in simulation speed versus accuracy, provided he/she is aware of tradeoffs involved.

4 Driver-Receiver Segregation

Figure 2 shows two inverters in sequence, which are modeled in the digital domain, and a capacitor C1, which is modeled in the analog domain, loading the digital net joining the inverters. The motivation behind driver receiver segregation is to allow the presence of the analog capacitor to influence the delay from the first inverter to the second. Since the inverters are modeled in the digital domain, they essentially know nothing about the presence of the analog domain, and the digital simulation kernel will faithfully propagate events from the first inverter (d1) to the second inverter (d2). Enter the concept of driver receiver segregation, which simply states that if there is a mixed signal present (i.e digital and analog net segments), then the digital receivers are *segregated* from the digital drivers for that mixed signal. As far as the simulator is concerned, this means that digital signal n1 is now *split internally* into two net segments, the driver side (the side closest to d1 in Figure 2) and the receiver side (that closest to d2 in Figure 2), and there is *no direct link* between them. The concept of driver receiver segregation intends for the *connect module* to provide that link. The connect module will be placed between the digital side of n1 and the analog side (terminal of the capacitor). The connect module is responsible for propagating the driver side of n1 (i.e. the output of inverter d1) to the receiver side of n1 (i.e. the input to inverter d2) with the appropriate delay.

Depending on the implementation of this connect module, it may be written to directly propagate the driver side values to the receiver sides without any delay, or it may be written to introduce some delay, either a purely digital delay, or one that is controlled by the analog solver

Of course, the connect module must also do the necessary domain translation i.e. convert digital to analog or vice versa. The Verilog-AMS language provides a lot of freedom in customizing these tasks. While the concept of driver receiver segregation allows a large degree of freedom and accuracy in modeling the delay on mixed nets, it also introduces:

1. requirements on the connect modules themselves (see Section 5).
2. requirements on the designer to think in a different manner, regarding the concepts of drivers and receivers, which are no longer directly related. However, well written connect modules will reduce the requirements on the designer.

From a driver-receiver segregation standpoint, it helps to be aware of a set of rules/guidelines when writing and/or using connect modules:

Rule 1 *A mixed net is always subject to driver receiver segregation.*

The connect module that gets inserted in this net must also take care of propagating the digital driver values to the digital receivers. How it does this is at the discretion of the person who writes or implements the connect module itself. The easiest way to perform this is via a continuous assignment statement such as 'assign d = d;' within the connect module, as in the following example. This module serves two purposes:

1. It reads the value of d (i.e. the resolved value of the ordinary module drivers, in this case the output of inverter d1), and based on this value, it drives the corresponding analog node either high or low, using a transition statement. This performs the domain translation.
2. It reads the value of d (i.e. the resolved value of the ordinary module drivers, in this case the output of inverter d1), and directly propagates it to the ordinary module receivers (in this case the input to inverter d2) by assigning to d in the continuous assignment statement.

In other words, this module both performs the desired domain conversion, and propagates the digital driver values to the digital receivers, compensating for the "missing link" or segregation that was introduced by the simulator.

```
connectmodule c2e(d,a);
logic d; input d;
electrical a; output a;

assign d = d; // d on left side goes to receivers,
              // d on right side is resolved drivers.
analog
   V(a) <+ transistion( d == 1 ? 5.0 : 0.0 );
endmodule
```

Rule 2 *When the digital port of a connect module is read from, it is the resolved value of all the ordinary modules drivers that are associated with the corresponding digital net that is being read, and when it is assigned to, the value being assigned becomes a driver for the ordinary module receivers associated with that digital net.*

This rule holds *regardless* of the declared directions of the digital port, input, output, or inout.

Rule 3 *If the digital port of a connect module is not driven, then the ordinary module receivers associated with the digital net connected to that port may never receive any values from the ordinary drivers.*

Rule 4 *Driving a value to the digital port of a connect module does not automatically write this value to the corresponding digital receivers, instead it becomes a contribution to the state of these receivers.*

If there are multiple connect modules attached to the same mixed signal, then the values driven to each module's digital port are collected, and the resolved value of these drivers is what actually drives the ordinary receivers.

Rule 5 *Connect modules are not aware of each others existence.*

Any connect module should thus assume that it may be the only one associated with the mixed signal, and must perform driver-receiver propagation.

Rule 6 *Any connect module will only see the ordinary module drivers and receivers for the digital island that it is isolating.*

Rule 7 *The use of a split connect_mode attribute not only causes multiple connect module instances to be inserted, it also has the effect of splitting the ordinary module drivers and receivers that can be read from or contributed to by a connect module.*

The previous two rules are illustrated with the following Figure 3. Here, inverters d1, d2 and d3 are on the same digital *island*. They are not connected to any other digital drivers/receivers, by any other digital net, and only by a purely analog net to inverters d4, d5 and d6. Similarly, inverters d4 and d5 exist on their own digital island, and can only communicate with the other inverters via the analog wire W1, hence through the analog simulator. In terms of driver receiver segregation, this is important. The connect module c2e1, when reading from its digital port, sees only the ordinary module drivers d1 and d3. It does not see drivers d4 or d5, since there is no digital connection to these drivers. In addition, when connect module c2e1 writes to its digital port to perform driver-receiver propagation, it only propagates the values of d1,d3 to receiver d2. It does not contribute to receiver d6.

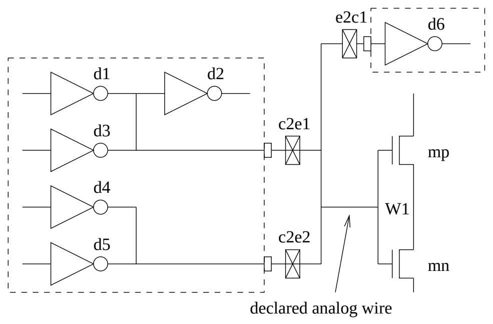

Figure 3. Digital Islands

Rule 8 *Connect modules inserted in a net, regardless of whether they are intended as analog-to-digital converters, digital-to-analog converters, or bidirectional converters, cause driver-receiver segregation of that mixed net.*

Hence they should all perform driver-receiver propagation, regardless of their port directions.

Rule 9 *Any digital nets that are connected to the ports of manually inserted connect module (may be inputs, outputs or inouts) are treated as mixed nets, in exactly the same fashion that they would be if the connect module was automatically inserted by the simulator.*

A module with a two ports of different domains can be instantiated to map from one domain to the other. However, if that module is declared as a *connectmodule* (rather than as a module), then it *will* cause driver-receiver segregation to occur. Hence, the module should perform driver-receiver propagation.

5 Connect Module Contents

5.1 Cross Domain Access

There are various ways in which values of objects which are calculated in the analog domain can be made accessible in the digital domain, and vice versa.

5.1.1 Demand sensitivity

The value of an analog variable or net is available within the digital context (always or initial block), simply by accessing it in an expression as follows:

```
analog
    aVar = laplace_zp(....);
    // aVar is result of laplace function

always begin
   dVar = aVar;   // read variable aVar whose value is
                  // calculated by analog,
   if ( V(anet) > 2.5 ) ) // access voltage on anet...
      clk = ~clk; // ... and wiggle clock if necessary
   ...
end
```

Similarly, the value of a digital variable or net is available within the analog context (analog block), by accessing it directly within an expression as follows:

```
always ( @posedge(clk) )
   dVar = ~dVar; // wiggle a digital variable...

analog begin
   if ( dVar == 1 ) // ... and read it in analog block
      out_v = 5.0;
   V(out) <+ transition(out_v,delay,rise,fall);
end
```

Here, the analog output voltage V(out) is driven by an analog variable out_v, which is conditionally set whenever the digital variable dVar is equal to one. Demand sensitivity means that the analog kernel will "demand" the value of digital objects from the digital simulation kernel only when it next executes an analog block statement that references the digital object in an expression, and vice versa. A change in the object in one kernel does not necessarily result in an immediate recognition of this change in the other kernel, and a delay can exist between the object changing value in one kernel, and the other kernel realizing it. Immediate notification of changes in values from one kernel to the other are served by what is known as transition sensitivity.

5.1.2 Transition Sensitivity

Transition sensitivity comes with the notion of events and event notification. The digital solver can be immediately notified of a change in an analog net voltage by referencing a cross event in an event detection statement:

```
always @ ( cross(V(a_net) - 2.5, 0) )
   clk = ~clk;
   // clock is toggled IMMEDIATELY whenever a_net > 2.5
```

Similarly, the analog simulator kernel can be immediately notified of a change in a digital object by referencing it in an event detection statement, such as:

```
analog begin
    always @ ( posedge(clk) )
        a_out = 5.0;
    V(out) <+ transition(a_out,delay,rise,fall);
end
```

In this case, the analog simulator will detect the positive edge on digital signal clk as soon as it happens, and will immediately begin to transition the analog output to 5.0 volts.

5.2 Performance/Accuracy Tradeoff

There are two ways in which the digital domain can determine if a threshold crossing has been reached in the analog domain, which is often required when performing analog-to-digital conversion at the appropriate interface. In this section we will discuss efficiency aspects of cross event detection vs. threshold crossing (polling). Cross event detection is implemented using the syntax shown in the first example in Section 5.1.2. In this case, the analog solver determines exactly (i.e. within tight tolerances) when the voltage of a_net crosses the 2.5 volt transition point. It will arrange for an analog solution point to be placed such that the timing of this crossing is accurately resolved. It will then inform the digital kernel that the cross has happened, and the digital process (the always statement) which was previously blocking, will resume. In order to accurately resolve the timing of the cross event, the analog solver may have to solve its system of differential equations at many different time points, until it finds one sufficiently close to the actual time at which the cross event happens. Such a process can be computationally expensive, as solving the system of equations in the analog simulator is a CPU intensive process. However, it does result in the digital always process (which was blocking waiting for the event to happen) to resume at the correct time, leading to accurate simulation results at the expense of simulation time.

Polling a threshold crossing is accomplished using the following syntax (i.e. without the event detection "@" operator, and using a simple "if" statement instead)

```
always begin
   if ( V(a_net) > 2.5 ) ) // access voltage on anet...
      clk = ~clk; // ... and wiggle clock if necessary
   #1; // delay: very important to break infinite loop!!
end
```

Here, the always block will continually loop, checking to see if the voltage on a_net has become greater than 2.5. If the important delayed NULL statement were omitted (the #1; statement in the above example), then the digital solver would immediately resume the always process, and digital simulation time would never advance. This corresponds to regular digital Verilog [1] semantics. In addition, because this always processes never suspends pending on event notification, the effect is that it will demand the value of V(a_net) from the analog solver every digital time tick (assuming a delay of 1 time-tick #1 as in the example). If the delay was set to 5 time-ticks, the process would suspend and resume every five time ticks, inquiring the value of V(a_net) from the analog solver, and then comparing to see if it has become greater than 2.5 volts. At one of these digital time points, it will eventually discover that at some point during the intervening 5 time ticks, the value of a_net finally increased above 2.5 volts. It will not know exactly at what time this happened, as it only discovers it at one of the time tick multiples governed by the value of the delay. However, since there is no cross statement that needs accurate resolution by the analog solver now however, the analog solver will thus perform less solutions, thus resulting in a significant increase in simulation speed from the analog solver's perspective. There is a trade-off to be made here, in which the digital solvers simulation time can be increased (thus performance decreased) by having a smaller delay statement within the always process, while the accuracy (with regard to clock edge timing) of the simulation is increased. If accurate timing is critical, the cross event detection mechanism is recommended. However, if it is not so critical, then the threshold detection (polling) can provide significant performance benefits, with further performance/accuracy tradeoff possible by varying the delay value in the always process. Note, the use of a delay polling mechanism might also result in missed thresholds, in case the polled value crosses the threshold an even number of times within the delay interval.

5.3 Driver Access Functions

When performing a digital-to-analog conversion, it is useful to know exactly which digital drivers are contributing to the state of a net. By knowing exactly which digital drivers are contributing, which value each driver contributes, and by making certain technology assumptions about such drivers based on their disciplines, a particular connect module can construct a detailed analog equivalent circuit of these drivers. This equivalent circuit is then presented to the analog simulation kernel. The driver access functions supported by Verilog-AMS provide this information for a given net.

The following connect module uses the driver access feature of Verilog-AMS to examine the individual digital drivers of the digital signal to which it is connected. It uses a number of approximations and assumptions about the analog characteristics of a "cmos1" driver to present an accurate equivalent circuit of the digital signal to port a. The voltage at port a is, in turn, used to determine the effective logic state seen by receivers of the digital signal. The following assumptions about "cmos1" are embodied in the module:

1. The equivalent circuit of an output in this technology is a function of the rail to ground supply voltage "supply"
2. When a gate output in "cmos1" is driven high, its equivalent circuit can be approximated by a resistance "r1" between the output and the power rail.
3. When a gate output in "cmos1" is driven low, its equivalent circuit can be approximated by a resistance "r0" between the output and ground.
4. The impedance between rail and output when the output is driven low, or the impedance between output and ground when the output is driven high is a few orders of magnitude higher than "r1" and "r0" so that its effect is not very important

This module effectively adds another parallel resistor from output to ground whenever a digital output connected to the net goes low, and another parallel resistor from output to rail (supply) whenever a digital output connected to the net goes high. Thus if this were used as the c2e in Figure 1, then not only would the delay from digital outputs to the digital input be a function of the value of the capacitor, but, for a given capacitance it would be approximately half the time with two gates driving the signal as with one (split/merge).

```
connectmodule c2e(d,a);
cmos1 d; input d;
electrical a; inout a;

electrical rail;
integer num_ones, num_zeros;
reg tmp;
branch pull_up(rail,a);
branch pull_down(a,ground);
branch power(rail,ground);
parameter real r0=120.0, r1=100.0, roff=1e6;
parameter real vt_hi = 3.5, vt_lo = 1.5;
parameter real supply = 5.0;

always @(driver_update(d)) begin
    num_ones = 0;
    num_zeros = 0;
    for ( i = 0; i < driver_count(d); i++ )
        if ( driver_state(i) == 1 )
            num_ones = num_ones + 1;
        else
            num_zeros = num_zeros + 1;
end

assign d = tmp;   // bind d to a reg

always @(cross(V(a) - vt_hi, -1) or
         cross(V(a) - vt_lo, +1))
    tmp = 1'bx;

always @(cross(V(a) - vt_hi, +1))
    tmp = 1'b1;

always @(cross(V(a) - vt_lo, -1))
    tmp = 1'b0;

analog begin
    // Approximately one r1 resistor to rail
    // per high output connected to the digital net
    V(pull_up) <+
        1/((1/r1)*num_ones+(1/roff)) * I(pull_up);
    // Approxately one r0 resistor to ground
    // per low output connected to the digital net
    V(pull_down) <+
        1/((1/r0)*num_zeros+(1/roff)) * I(pull_down);
    V(power) <+ supply; // specify power value
end
endmodule
```

The module above is able to model the equivalent output impedance of the digital drivers, with only the addition of a single analog node, and two extra branch equations. Traditional approaches to model such impedances include the placement of transistor models (primitives) within the connect element. While this approach provides even more accuracy (and is still possible using Verilog-AMS), the complexity of the underlying model equations, and the extra unknowns that need to be solved by the analog solver (introduced by internal nodes within the transistor models themselves), can degrade the performance of the analog solver. Hence, by judicious use of driver access functions and the appropriate level of behavioral modeling, significant increases in simulation speed can be achieved with small losses in simulation accuracy. This can be a very compelling reason to use driver update functions within connect modules.

5.4 Finite Output Impedance

Since multiple connect modules might drive the same analog node, the question of output impedance is crucial for Verilog-AMS models. Well written connect modules should be written with finite input and output impedances, regardless of whether they are intended to be used as digital-to-analog converters, or analog-to-digital converters. At first glance it may seem that a connect module written to map from the digital domain to the analog domain need not contain any output impedance modeling step, and that it can simply "write" the corresponding logic high/low voltage value to the analog net. However, cases (see Figure 4) can arise where multiple connect module instances are simultaneously trying to force values on the same analog net. If implemented as purely behavioral voltage sources

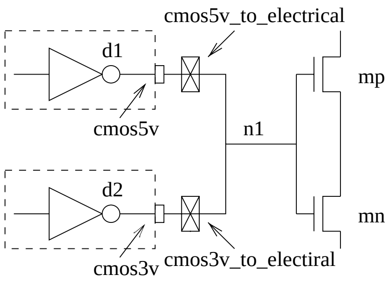

Figure 4. Different logic connect modules

(contribution statements), this situation will lead to a singular matrix in the analog simulation kernel. However, in a real world situation, the hardware implementation of the logic gates d1 and d2 would have some finite analog output impedance, and a real voltage (probably somewhere between 3v and 5v) would actually result on net n1.

The following module cmos5v_to_electrical illustrates how a finite output-impedance can be modeled by use of an intermediate node (n), and a resistive contribution statement *I(a,n) <+ V(n) / rout*.

```
connectmodule cmos5v_to_electrical(d,a);
cmos1 d; inout d;
electrical a; output a;
parameter rout=50; // output impedance, 50 ohms
reg tmp;
electrical n;      // intermediate node

assign d = tmp;    // bind d to a reg
...

analog begin
    V(n) <+ transition( d == 1 ? 5.0 : 0.0, 3n, 3n);
    I(a,n) <+ V(n) / rout; // models output impedance
end
endmodule
```

6 Conclusion

Since Verilog-AMS actively supports the mixed-signal approach, the interchange of digital and analog portions is straightforward and strongly encouraged. Connect modules are essential to this, and this paper has illustrated several salient features of the Verilog-AMS language. Included were differences in simulation semantics between the analog and digital contexts, discipline resolution, auto-insertion of connect modules, and the significant impacts of driver receiver segregation. Speed/performance tradeoffs in modeling and implementation of connect modules were investigated, and a set of rules/guidelines was established for use when coding connect modules. These tradeoffs, rules and guidelines are particularly applicable to an end-user when simulating a mixed-signal design using Verilog-AMS.

References

[1] IEEE. *IEEE Standard Hardware Description Language based on Verilog Hardware Description Language*, Oct. 1996.

[2] Open Verilog International. *Verilog-AMS Language Reference Manual 2.0*, Jan. 2000.