P1800 Proposals for Mixed-UDN support

Author: Kevin Cameron

Background

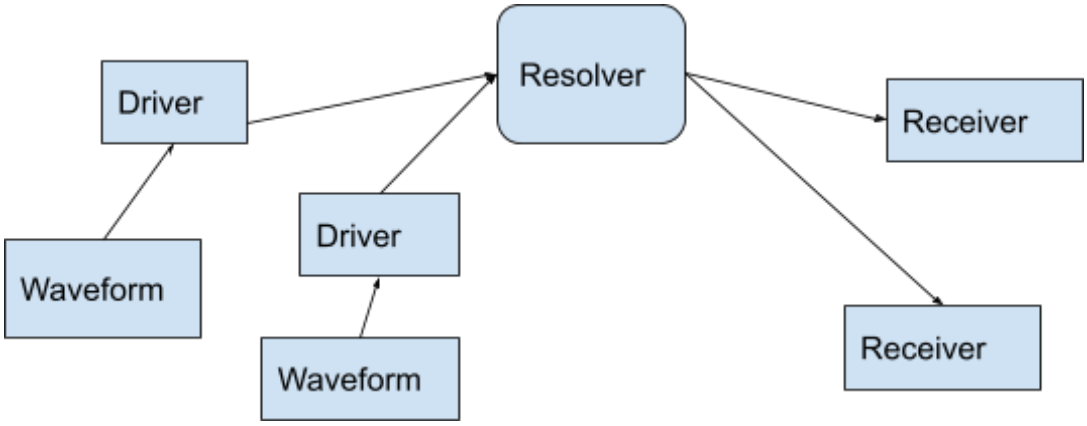
UDNs (user defined types) were added to SystemVerilog (SV) a number of years ago, giving it the same capability as VHDL, but neither language has added support for mixing different UDNs on a net, or UDNs as a replacement built-in types. However, Verilog-AMS does supply that capability across the extremes of the type system, and its approach can be generalized to UDNs.

Motivation for enhancing SV to support mixed-type nets includes being able to model power and thermal issues as well as enhanced (supply dependent) timing with more accurate back annotation. RF modeling for system level verification of complex MIMO scenarios is also desirable, and is currently not available in any HDL.

Drivers, Receivers, Natures & Disciplines

Within the simulators models are attached to nets through drivers and receivers, drivers and receivers are the things that have type (as defined in a UDN), generally the drivers and receivers have a single “nature”, e.g. Voltage or current, and a net has a single discipline (e.g. electrical, which is the combination of a potential and a flow nature). With a mixed type net, the natures and disciplines need to be compatible across all connected components, and a process of “resolution” is used to combine all the driver values to provide values to the receivers.

Fig 1. Existing net form:



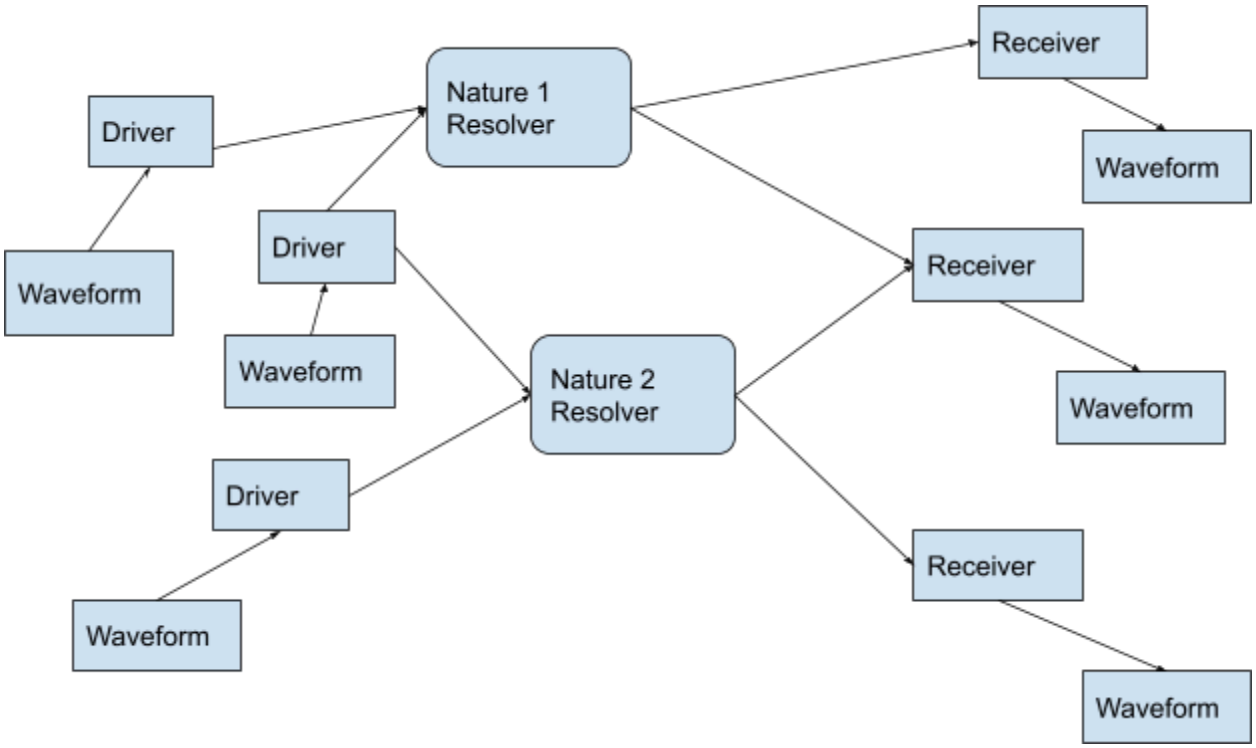
With the single type on the net the resolver yields the same value for all receivers, so it is viewed as the net’s value rather than the receivers having a value, and the receivers get

collapsed. In Verilog-AMS the (logic) receivers can have different Voltage thresholds when the resolver is converting from analog. The receivers don't have waveforms because connect-modules save the context (for future values).

For backward compatibility we are looking to preserve the behavior within (SV) modules, not necessarily the behavior between them. For simplicity we can take the view that all wires (aka simulation nets) are of discipline electrical by default, with drivers and receivers of nature voltage. Receivers are not generally recognized in digital HDL LRMs because the type is the same across drivers and receivers, and it is essentially a pass-through component in the resolution process. In mixed-type resolution, receivers on a net may have different characteristics, Verilog-AMS recognized that logic level trip points could be different when converting from Voltage to logic, which is why it was added

In SPICE level simulation the analog solver looks at all the connected models at one go and attempts to apply Kirchoff's Current Law (KCL) in conjunction with branch voltages across all nets. KCL says that the total flow is zero. In discrete simulation (without a solver) Voltage is driven in one direction on a net, and current is driven in the other direction. For KCL to apply the Voltage driver needs to be associated with a current "sink", that absorbs the return current to get the zero total.

Fig 2. New net form:



In the new form it is best to look at the network from the driver/waveforms to receiver/waveforms as a (unidirectional) network in "resolver space", (static) resolvers are just like modules sensitive to the driver/waveform changes, but are artifacts of simulation (not part of the design). The

per-receiver waveforms are used to support functions like “cross” with PWL signals, resolvers can react to waveform changes (pending updates) on the driver side to create waveforms on the receiver side, and that allows “glitchy” behavior to be modeled.
When a signal is declared as (say) “reg x” in a module, that essentially creates an unnamed logic driver and logic receiver, with assignment being to the driver, and reads being from the receiver.

Marking up a UDN with natures implicitly indicates the UDN represents values on a wire. The mark-up can be simply labeling sections of the type:

```
`include “disciplines.vams” // same header as Verilog-AMS

// Struct to define Voltage, current, and resistance for the electrical nettype:
typedef struct {
   potential Voltage:
         real V;
         real R;
    flow Current:
         real I;
}  EEstruct;
```

That makes it easier for automatically doing conversions, and adding “sink” statements. E.g.:

```
assign vdd_out.Voltage = ‘{2.0,0.001}; // drive 2V at 10mOhm
sink     vdd_out.Current; // absorb all the current
```

SystemVerilog should ignore the “domain” directive in Verilog-AMS, since it was an unnecessary crutch for poor compilers. Likewise SystemVerilog does not need to limit itself to two natures in a UDN (see Multiple Natures below). The discipline specification includes “access functions”, which could be used instead of the *.<nature syntax>*, i.e. *V(x)* is equivalent to *x.Voltage*.
The above code snippet could be used in an LDO, with loads that just drive current back:

```
If (vdd.Voltage > 1.0) begin  // vdd is EEstruct
         vdd.Current = 1e-3;
         powered = 1’b1;
else
         vdd.Current = 0.0;   // current driver
         powered = 1’b0;
end
```

Note: only some types honor port directions, and voltage and current are going in opposite directions in the example above.

Timing with analog behavior

Analog simulations tend to deal with signals as PWL (piecewise linear), rather than discrete, but that can be handled in user code if the scheduled (driver) waveforms are available. Likewise scheduling a resolution result to a receiver rather than just assigning it enables driving PWL waveforms out of the resolution process (transport delay), see below in the Combined Resolver section.

Connect Rules

Verilog-AMS uses connect rules to indicate to a simulator how to handle resolution. The discipline scheme in Verilog-AMS is not type-specific, and was intended as a way to mark up ports in modules with useful information about how they connect; the same framework can be used in SV to direct UDN conversion.
The rule based approach was used to allow the addition of extra rules outside the design itself, to cope with mixed-signal situations as they arise, when IP is reused.

UDN Connect Rules

For handling multiple UDNs, the simple input/output methodology can be extended to having arrays of input and output (or inout) for all the distinct types handled. Viewing the driver/receiver network as just another circuit, the c-module prototype is just:

```
module <resolver  name> ((input|output|inout) <nettype> <port name>[]
                                    {,(input|output|inout) <nettype> <port name>[]});
```

The module can be added to the connectrules as with *connectmodule*, the modules to be used for resolvers would be listed in the *connectrules*. The *connectmodule* keyword in Verilog-AMS is somewhat redundant, and was mostly a crutch for lazy parsing tools.

If a module or function is to be used for automatic type conversion, it would be added to the connectrules preceded by the keywords *lossy* or *lossless*. See below.

Other Rules

More general rules to consider when designing HDLs:

1. Hierarchy and modularity are a convenience for humans, the order and placement of components should not affect resulting behavior.

Verilog-AMS violates (1) by not allowing multiple analog blocks in a module, which means you cannot coalesce multiple modules into one without modifying behavior. The idea of typed net-segments also breaks the rule.

In a similar vein, adding more receivers to a net should avoid impacting the result when the set of drivers remain the same.

Rewrite Rules

Constructs like “assign” in SystemVerilog are shorthand for creating a process, for analysis purpose these constructs can be translated into an equivalent “always@” version, e.g.:

```
assign #1 a = b; // a is a wire
```

Becomes something like:

```
reg a; always@(b) a #1 = b;
```

In a similar vein Verilog-AMS branch functions can be interpreted with a rewrite rule; if you see something like *V(a,b)*, then it can be rewritten as *a.V(b)*, which makes it a method of *a*. The rewrite rules can include labels to be used in accessing the involved drivers and receivers.

Implicit vs Explicit Drivers

In the original version of Verilog the *reg* statement created a driver in the simulator, this was later misinterpreted as a type because Verilog only had one type, i.e. a *reg* statement declares a driver of type *logic* and a corresponding wire, and assignments within the module are to the driver, not any attached port or wire.
In a similar vein Verilog has implicit receivers, and because they are all type *logic* they are merged into one for resolution.
If one wants to be explicit about drivers, one can extend the reg syntax to use nettypes, e.g. rather wreal (a non-standard wire with real drivers/receivers), you could say:

```
reg(real) my_wreal; // assignments through this driver are of type real
```

The old *reg* is equivalent to *reg(logic)*. This is useful when constructing rewrite rules in the standard, even if it isn’t part of the official syntax people implement, however there is no particular (logical) reason you can have multiple drivers (of different types) for the same net within a module. For the *assign* rewrite above you might get:

```
begin reg(typeof(a)) a_driver; alias a = a_driver;
        always@(b) a_diver #1 = b; end
```

Built-in (Primitive Module) Replacement

A lot of tools have been built to use Verilog, and we would like to avoid unnecessary rewriting of software, so it's desirable that netlists become polymorphic. Verilog was invented in the 1980s, and it focused on compact representations in order to have a low memory footprint at runtime, which it achieved with its 0,1,X,Z (& strength model). Unfortunately, that that representation mashes three concepts together (value, certainty, and strength), which are easier to deal with unbundled when converting between types, it also leads to "X-propagation" since having an X clobbers the 1/0 and puts all downstream logic into an unknown state.
Replacing the standard "logic" type with a different type (UDN) is not particularly difficult, but we also want to replace the built-in "primitives" that go with it, since it would cost in performance and accuracy if we have to down-convert and up-convert on the way through gates between the user-defined pieces of a design. The difficult components to model are the bi-directional transistor devices, but one can consider them as being like Verilog-AMS connect-modules which only do logic operations (and don't drive analog), since Verilog-AMS has a set of "driver access" functions that allow you to write the needed code - additional ports can be added for control.

Type Priority

When dealing with resolving multiple UDNs it is desirable to know which are more accurate, e.g. strength-logic is better than bool. This can be done by external rules (as with Verilog-AMS connect rules), or by marking conversion functions as lossless. E.g.*bool->logic->bool* is a lossless conversion, *logic->bool->logic* is lossy since strength and certainty will be lost. For simple cases lossy/lossless can be determined by the compilers.

Automatic resolution

Given the type priority, UDN/AMS resolution can be performed by doing:

1. Convert drivers losslessly to a level where there is a resolution function
2. Resolve
3. Convert resolved value back to receivers (with lossy conversion)

Conversions can be functions, or modules (as in Verilog-AMS), and a stack of conversions can be used. For AMS the resolution level is in the analog domain, but if all the drivers and receivers are discrete an overarching UDN can be used. The stack of conversions may include resolution itself (combining multiple drivers to one, see below on lossiness).

The MAR (most accurate representation) is the type resulting at (2). The MAR could be declared explicitly if the resolution stack is ambiguous, but the goal here is not to constrain the tools/users unnecessarily. As with connect-modules, new scenarios that pop up as a design

comes together can be dealt with by simply adding extra conversion functions. There is no need to predeclare what resolvers will be used on any particular net.

Resolution function enhancements

Allowing a resolution function to return different types from the drivers being resolved is helpful, e.g. a resolver for bool -> logic. Syntactically that's easier to do as a resolution function with different inputs, i.e. the bool->logic resolver would be attached to the logic type. It's also useful to be able to declare them independently of the resolved type.

Resolution functions that return the same type as their input are probably intrinsically lossy, since you are compressing the information from multiple drivers to one.
Resolution is applied per-nature, since drivers/contributions can be located anywhere, and don't necessarily interact, e.g. Voltage and current travel in opposite directions, and DC operating point is separate from small-signal RF behavior.

Examples:

```
// resolver for EEnet  (ignoring Current)
function automatic EEstruct res_EE (input EEstruct.Voltage driver[]);

// resolver for EEnet  working from logic
function automatic EEstruct res_EE (input logic driver[]);
```

Connect-Modules in SystemVerilog

The philosophy for the *connectmodule* in Verilog-AMS, is that it should really just be the same as a regular module in terms of what you can put in it, but the input and output signals are essentially drivers and receivers rather than actual nets. Here are examples copied from the Verilog-AMS LRM:

```
connectmodule elect_to_logic(el,cm);
 input el;
 output cm;
 reg cm;
 electrical el;
 ddiscrete cm;
 always @(cross(V(el) - 2.5, 1))
   cm = 1;
 always @(cross(V(el) - 2.5, -1))
   cm = 0;
endmodule
```

```
connectmodule logic_to_elect(cm,el);
 input cm;
 output el;
 ddiscrete cm;
 electrical el;
 analog V(el) <+ transition((cm == 1) ? 5.0 : 0.0);
endmodule
```

In the context of the first c-module *el* is the resolved value of the net, and it is being converted and assigned to the logic receiver represented by *cm*. In the second c-module *cm* is a (discrete) logic driver, and is being converted into a contribution to *el*. The *input* and *output* keywords indicate the direction of the conversion.
Fairly obviously one could replace *cm's* type (*logic*) with a UDN, for full analog conversion, but one can also drop the *electrical* for a UDN and use an *always* block instead of the *analog* block. Note: the "electrical" typing is redundant since everything has to be of the same nature/discipline anyway, and the actual nature will be defined in regular code, it's really the "V(el)" that tells you what's happening. The function "cross(V(el)..." could be implemented in SystemVerilog by looking at the current value of a signal and the next scheduled value, and returning a time (*cross()* being a method of the UDN). As with driver-access (below) the next value for a signal can be made available as an implicit signal, and the next value is created when a non-blocking assign with a delay is used to update a receiver.
C-modules would be instantiated as children of the module where the driver or receiver is located, so that upward cross-module references can be used to locate power supplies (Vdd, Vss etc.).

No-MAR Resolution

Although the idea of a MAR is appealing it isn't actually necessary, resolvers can be applied in parallel to a set of drivers to get individual results for different receivers. E.g. if you have a set Thevenin drivers going to logic receivers each receiver might have its own Vt characteristics, and going to a PWL MAR to work out crossing points would be more complicated than needed. See "Arena Resolution" below about resolving RF in free space, other scenarios include separating small signal (e.g. audio) from the DC bias in the electronics.

The no-MAR or per-receiver approach can have performance advantages because it decouples the different natures on the net, which avoids doing a MAR calculation on any driver change, and allows specific driver/nature changes to propagate to interested receivers without activating disinterested ones. In RF modeling of OFDM modems, this also means you can evaluate the separate frequencies in parallel.

For debug it may be desirable to generate a MAR view to present to DV/design engineers, but that can be done in the viewing tools.

Multiple Natures

Verilog-AMS limited wires to carrying two natures because that's how analog simulators work (just voltage and current), but extra natures can be added to represent other things about what is attached to a net. An example of that is (grounded) capacitance, where each attached module drives a capacitance value, another is multiple carrier RF (one nature per carrier).

Multiple Nettypes in a Module

As can be seen in the no-MAR cases it's desirable to split drivers and receivers to deal with natures individually, and that should be possible within a module as well as across modules. To support that within a module the "alias" statement can be used to say local nets declared as different UDNs are the same net or some subsection (of the MAR), but the driver/receiver type is taken from the aliased item (see "Implicit vs Explicit Drivers"). The following is an example where we ignore one nature and alias the others.

```
typedef struct {
    potential Voltage:
        real V;
    flow Current: // could be marked as a sink (receiver only)
        real I;
    potential ripple:
        real Vr;    // supply ripple pk-pk
}  PowerNetStruct;

inout PowerNet Vout;
real  I,V;
alias V  = Vout.Voltage;
alias I    = Vout.Current;
sink       Vout.Current; // receiver for all the current
always @(enable) V = enable * 1.2; // driver
Always @(*) power = I * V;
```

If you want to indicate a particular type for a nature section, it can be optionally added into the flow/potential declaration, e.g.:

```
(flow|potential) [<type>] name:
```

Nature definitions within a UDN can be viewed as advisory and can be guarded by *`ifdef*. An objective of marking up the UDNs with natures is to let the tools know what you are doing for automation purposes.

Bridging to Analog

The existing case of Verilog-AMS has a MAR of discipline electrical and real types for the natures. The main difference in an analog (SPICE level) simulator is that there is a mathematical relationship between the drivers and receivers that needs to be satisfied as part of the resolution process, otherwise the same structure of drivers and receivers applies, for clarity we can call drivers that have a common mathematical relationship “contributions” for consistency with Verilog-AMS.

Branches

Verilog-AMS uses “branches” to assign currents and Voltages; a branch involves two nets. While this is a more complicated scenario, it can be pushed to user space if the adapter stacks for the two nets are mutually aware of each other. A branch can be considered as a pair of drivers of opposite polarity (and matching flow). Analog solvers work by solving constraint equations across multiple branches at the same time using matrix math, similar functionality can be achieved in user space code in SystemVerilog by instantiating more structures in resolver-space that connect groups of drivers together.

Overloading Built-in Instances

While it is possible to just do blanket replacement of all primitive instances with a single UDN equivalent, or picking particular instances through configuration, there are advantages to automating the process. A primitive that matches its surrounding components in type will be more efficient in simulation since type conversions can be avoided. This is particularly true if a UDN based wiring model is used to back-annotate a netlist using standard Verilog (from legacy synthesis tools).

Back-Annotation

The simplest approach to back-annotation is to allow the breaking of ports and insertion of more circuitry, which also works for defect insertion in fault analysis. Back-annotation with (say) SPEF would be inserted after initial elaboration, and before choosing which module version when overloaded.
The intention with Verilog-AMS was to enable accurate timing simulation for digital sign-off by being able to use fast behavioral models for logic cells, with analog level wiring models. Accurate D2A/A2D connect-modules would be selected by marking the logic model ports with sub-classed disciplines (through pre-characterization), erroneous placement of the connect-modules and lack of back-annotation in the final standard precluded doing that. In the proposal below the former problem is fixed by giving users access to the driver context, so the location of the resolution code does not affect the process.

Driver Access by Implicit Signal

In Verilog-AMS the driver-access is mostly through system functions, however the driver/receiver network is static (in most cases) and the drivers and receivers can be viewed as implicit signals (VHDL parlance). Doing so limits the scope of language extensions to the (new) driver/receiver constructs, rather than having the globally available system functions (of Verilog-AMS).
Using this approach makes it easier to apply synthesis tools to resolution for emulator code generation, and avoids namespace/keyword issues.

Proposed implicit signals and methods for use in resolution functions:

- <signal>.driver - an array of driver references
- <signal>.receiver - an array of receiver references
- <signal>.drivers - count of drivers
- <signal>.receivers - count of receivers
- <signal>.driver[<index>].value - current value of driver
- <signal>.driver[<index>].last_value - last value of driver (for convenience)
- <signal>.driver[<index>].mine - boolean 1/0, true if process owns driver
- <signal>.driver[<index>].waveform - scheduled values (time,value pairs)
- <signal>.driver[<index>].update - bool(?) signal conversion modules can be sensitive to
- <signal>.driver[<index>].find(...) - look for something in the context of the driver
- <signal>.receiver[<index>].find(...) - look for something in the context of the receiver
- <signal>.receiver[<index>].sinks.<flow nature> - boolean for receiver being a sink
- <signal>.driver[<index>].sinks.<flow nature> - boolean for driver being a sink

The return value and arguments of the *find()* method should map to VPI calls, and will return handles to objects like the instance where the driver/receiver sits (so you can go look for Vdd/Vss). An alternative is to add a pseudo-scope in the path that takes you to the parent, e.g.:

- <signal>.driver[<index>].parent.<object name> - handle for item in driver's instance

This approach needs the right name to elaborate, which can be set by picking a particular resolver through connect rules. In the ".find()" approach you would probably want to query the driver discipline/UPF to disambiguate in the case of power-domains.

In regular code (for modeling bidirectional components) it would be necessary to specify the nature of interest (if there is more than one), so the syntax would be *<signal>.<nature>.driver…* rather than just *<signal>.driver…* Where a nature is not specified, it would default to potential/voltage.

The *.find(...)* methods would be (optionally) translated to DPI calls when not supported by the simulator, which allows access to information in other environments like UPF. In the case of a

translation stack, *find* calls are passed through to the base drivers and receivers when not satisfied in the stack.

PWL Handling

PWL signals can be handled by user interpretation in SV, for a UDN representing a PWL signal that means being able to schedule the end-point of the current value, and the following value, which can be done as transport-delay non-blocking assignments to the receiver -

```
<signal>.receiver[0] #1 = <signal>.receiver[0]; // maintain for 1t
<signal>.receiver[0] #2 = 1.0;                  // ramp to 1.0
```

Combined Resolver

The resolution mechanism described above could end up using a cumbersome stack of conversions, an alternative approach is to use a resolution function that can handle a heterogeneous array of drivers and receivers and adapt. Using the implicit signals above SV code can ask for the type name of a driver or receiver through *$typename* e.g. *$typename(<signal>.driver[<index>].value)*. To avoid repeated evaluation it is best if such a resolution function can be unrolled and treated as a module, with any static data being created per-net.
For the benefit of the compiler it is necessary to list the input types and natures accepted by this kind of resolver.

If the combined resolver is a static context (like a connectmodule) it can be triggered by external signals through DPI/VPI, which makes it the ideal connection point for co-simulation with analog simulators or testbenches (e.g.SystemC-AMS).

Here is an example of the code structure for resolving multiple UDNs in one go:

```
connectmodule (input in, output out); // treat as separate, but might be the same net
    EEnet mar; // MAR (local to this module)
    genvar d;
    generate
        for (d = ; d < length(in.driver); d++) begin
            always @ (in.driver[d].update)
                case (typeof(in.driver[d]))
                logic: begin
                    // use implicits to get Vdd/Vss and resistance
                    real Vl = in.driver[d].find(`DRIVER_VDD); // vendor defined
                    real Vh = in.driver[d].find(`DRIVER_VSS);
                    real R = in.driver[d].find(`DRIVER_R);
```

```
                        assign mar = {in.driver[d].value ? Vh : Vl,R,0.0};
                    end
                    Thevinin: begin // V/R drive
                        // copy through
                        assign mar =  {in.driver[d].value.V,
                                       in.driver[d].value.R,0.0};
                    end
                    endcase
                    real: begin // assume current
                        assign mar =  {0.0,`INFINITE,,in.driver[d].value};
                    end
                end
    endgenerate

    genvar r;
    generate
        for (r = ; r < length(out.receiver); r++) begin
            always @ (mar.V,mar.I)
                case (typeof(in.receiver[r]))
                logic: ....
                    // use implicits to get Vdd/Vss threshold
                    real Vl = in.driver[d].find(`RECEIVER_VDD); // vendor …
                    real Vh = in.driver[d].find(`RECEIVER_VSS);
                    real Vt = in.driver[d].find(`RECEIVER_THRESHOLD);
                    out.receiver[r] <= mar.V > Vt ? 1'b1 : 1'b0;
                Thevinin:
                    out.receiver[r] <= `{mar.V,mar.R};
                real: // total current
                    out.receiver[r] <= mar.I;
                endcase
        end
    endgenerate

endmodule
```

The net in the example is assumed to have one main Thevenin power driver and loads which drive Thevenin (resistance) or currents, The main driver would have a real-number receiver to get the current total. The location of the connectmodule instance is at the top of the net in the hierarchy, if it is the only top resolver, if it is in a stack of resolvers, it would be placed as low as possible above its drivers and receivers, only one resolver will be on the default hierarchical path (*<instance path>.<net>.resolver*), if natures are resolved separately, the path is *<instance path>.<net>.<nature>.resolver*. The fixed path allows the use of defparams to modify behavior at elaboration or OOMR during simulation (see Extra Functionalty). Syntactically,

connectmodule is just a synonym for module, and matches what is done in bidirectional component modeling, the keyword is just an indication of purpose to tools.
Assignment to receivers is limited to non-blocking, other semantic restrictions may be applied in initial implementation, but otherwise the full language should be available. The assignment to the receivers is done at the end of resolution, such that blocking assigns in regular code work as expected, i.e. there is no propagation of execution into regular code until the resolution is complete, but resolution consumes no time.
Since the connectmodule is a static context it can instantiate other modules, to create stacks of conversion if needed.

Typed C-Module Insertion

In the case where a typeof() function is not available, or the combined resolver can't handle all types (it would), then the mechanism mentioned in "Connect-Modules in SystemVerilog" above can be extended to have multiple input/output arrays for each type seen on the net. A simple case would be EEnet and logic, where the c-module would be declared something like:

```
module my_EE_logic_adapter(input logic lin[], input EEnet Ein[],
                                    output logic lout[],output EEnet Eout[]);

    genvar gl;
    generate
        for (gl = ; r < length(lin); gl++) begin // attach logic drivers
            …

    genvar gE;
    generate
        for (gE= ; r < length(Ein); gE++) begin // attach EEnet drivers
            …

   // reactive code, receiver drives
   ….
endmodule
```

The ordering of the ports is not significant, and the individual arrays may be empty. Port direction (input/output) implies driver/receiver, and *inout* implies a driver which you can write back to.

In the event no matching c-module is found for a net, the simulator/compiler will look for lossless type conversions (other c-modules) that can be inserted to create a usable set of driver and receiver signals. E.g. if one has a Voltage type (scalar real) it can be converted to EEnet for the above c-module with a simple c-module -

```
module V2EE (input real Vin[0:0], output EEnet Ein[0:0]); // limited to per-driver
```

```
        assign Ein[0] =  '{Vin[0],0.0,0.0}; // pass-through

    endmodule
```

And down-conversion -

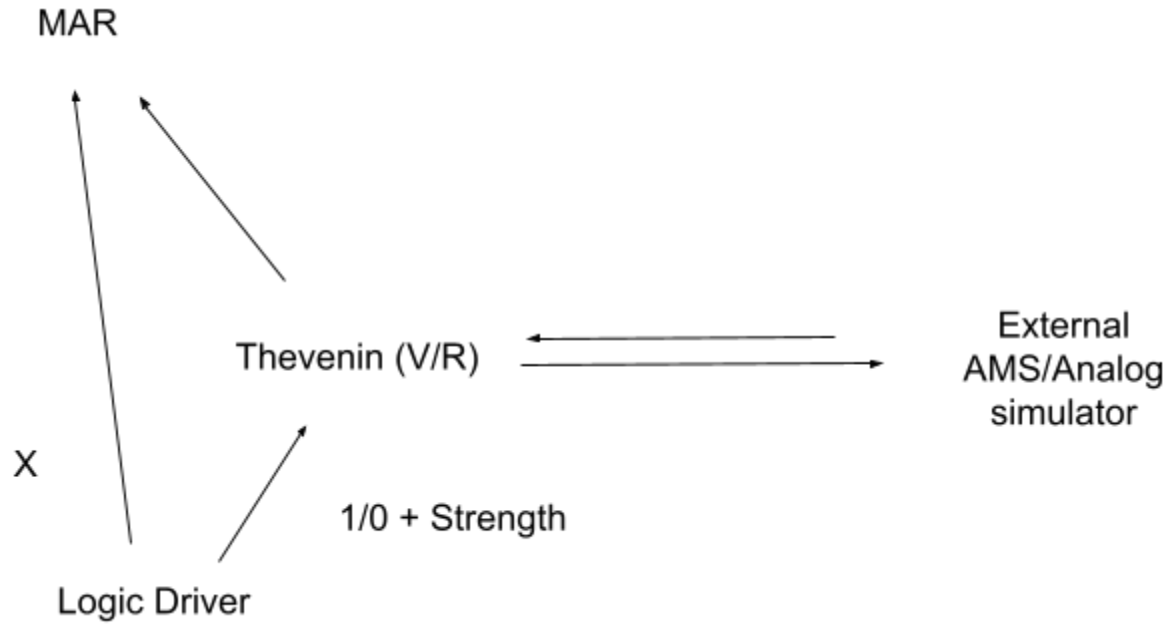
```
    module EE2V (output real Vin[1], input EEnet Ein[1]); // limited to per-receiver

        assign Vin[0] =  Ein[0].V; // pass-through

    endmodule
```

Notes on Thevenin Types

In analog simulation Thevenin drivers and receivers are resolved using matrix math, which allows components to be floating between the fixed Voltages. In discrete simulation Thevenin types driving a net need a common floating node and all other nodes tied to fixed Voltages, e.g. simple power supply models and simple loads. If a network of the latter type is broken at a port boundary to insert wiring components for back-annotation, it is unlikely it can be resolved simply. A super-adapter can make decisions about resolving such nets, e.g. making it work by pegging a floating node (e.g. shared LDO outputs) at a likely Voltage, and using impedance to calculate the current split.

Nature Splitting

The blurring of certainty and value causes some problems, so it is helpful to split a driver for combined value into multiple drivers in different natures so that something like certainty can bypass levels of resolution and be recombined later. I.e. an analog simulator cannot handle Xs so you want to give it clean Voltage and current values and reattach any X info to the result. I.e. the “MAR” is a level above analog resolution where the X component is recombined.



Other scenarios include splitting and recombination of multiple carrier RFsignals, as below.

X-Propagation

When the X (certainty) component is not being fed into a component, but you would like to propagate the information, extra components can be added in parallel to do that. A technique to achieve that is to instantiate two versions of the component - an accurate (behavioral analog) one and the legacy logic version, but only use the certainty (X) from the legacy version. Simulators can do that automatically given enough information, if already capable of overloading instances.

Noise

As with certainty, noise is a signal value that can be handled orthogonally to operating point or transient simulation evaluation, it can also be evaluated by instantiating a parallel model of an instance.

For both cases the simulator can be told that a model can be used for modeling particular natures, and the module would only be instantiated if drivers/receivers of the relevant type are seen on the nets.

Model State Sharing

In some cases parallel models may want to query each other, e.g. outgoing noise may depend on the circuit state, that requires an extension to the out-of-module-reference syntax to allow going sideways.

Other cases where this is useful include initialization of analog simulation from the digital state of a circuit, since cold starting an analog simulator often gets stuck trying to get convergence for the entire design. Also, verification modules that are the same regardless of the simulation model level being used.

Arena Resolution

An application of the combined-resolver is to handle resolving free-space signals like radio waves. We refer to this as “arena” resolution rather than wire/net resolution, where a bunch of drivers and receivers exist in a contained space, rather than being directly connected as with the KCL potential/flow view.

Resolution in a multiple-antenna scenario requires attaching the resolution to receivers (each representing an antenna) individually, the implicit signal driver-access allows access to a location and orientation which can be used to calculate attenuation due to distance and phase. That would be combined with the multiple carrier/nature mentioned above, noting that each carrier is a discrete signal with frequency, amplitude and phase, the event rate is considerably lower than the frequency, and carriers are orthogonal.

This just requires tagging the receiver type as arena-resolved, tagging the net as an arena, or using a super-adapter that is aware that is what is needed.

Bidirectional Components

Building components for bidirectional signal flow is similar to making connect-modules, except that you are resolving on one net to get the driver for another net, and resolving on that net to get a driver back into the first net. The key functionality needed is the ability to spot which drivers in the array of drivers belong to the component model itself. In some simulators this is done by having a “resolve others” function which excludes the component’s own driver(s), in Verilog-AMS you could ask for the index in the array of that driver. In the first case you have duplicate data since the component has to keep a copy of its own driver’s data, and the latter doesn’t handle multiple local drivers, so here we propose the implicit signal (or method) *driver[i].mine* to give user code that information.

The code below is an example implementation of the tranif1 built-in in user code using the driver implicit signals.

```
module tranif1 (Inout1, Inout2, Control);

        logic res1,res2;

        assign Inout2 = Control ? res1 : 1'bz;
        assign Inout1 = Control ? res2 : 1'bz;

        … // initialization and local variables go here, skipped for brevity

        genvar r1;
        generate
                for (r1 = ; r1 < length(Inout1.driver); r1++) begin
                    always @ (Inout1.driver[r1].update) begin
                        if  (false == Inout1.driver[r1].mine
                            && Inout1.driver[r1].value != Inout1.driver[r1].last_value) begin
                          case (Inout1.driver[r1].last_value)
                                1'bz: Zs–; 'bx: Xs–; 1'b0: Os–; 1'b1: Is–; endcase
                          case (Inout1.driver[r1].value)
                                1'bz: Zs++ 'bx: Xs++; 1'b0: O++; 1'b1: I++; endcase
                          if (0 != Xs)  res1 = 1'bx;
                          else if (0 == Os) res1 = 1'b1;
                          else res1 = 1'b0;
                        end
                    end
                end
        endgenerate

        genvar r2; // repeat above going the other way
        generate
                for (r2 = ; r2 < length(Inout2.driver); r2++) begin
                    always @ (Inout2.driver[r2].update) begin
                        if  (false == Inout2.driver[r2].mine
                            && Inout2.driver[r2].value != Inout2.driver[r2].last_value) begin
                          case (Inout2.driver[r2].last_value)
                                1'bz: Zs–; 'bx: Xs–; 1'b0: Os–; 1'b1: Is–; endcase
                          case (Inout2.driver[r2].value)
                                1'bz: Zs++ 'bx: Xs++; 1'b0: O++; 1'b1: I++; endcase
                          if (0 != Xs)  res2 = 1'bx;
                          else if (0 == Os) res2 = 1'b1;
                          else res2 = 1'b0;
                        end
                    end
                end
        endgenerate
```

endmodule

The use of generate statements allows the compilers to flatten the module out so it looks like static logic that can be synthesized for use on emulators, i.e. the path from a driver to a receiver looks the same as regular simulation code as far as the compilers are concerned.

Adapter Access

As with drivers & receivers, implicit names or “virtual hierarchy” can be used to access the resolver stack; a syntax like *<net>.resolver* anywhere on the net will take you into the resolution network. So, although the location of the adapters might change with different schemes, the access points remain the same. E.g., if you know logic is being converted to Voltage, you can refer to the Voltage as <net>.resolver.<UDN name>.<field>, maybe: lsig.resolver.EEnet.V. If the resolution involves a stack of converters, the compiler can search upward in the stack looking for the matching type. That allows for the addition of assertions about how the conversion is being managed.

Nature Extensions

Certainty

The “X” in the standard logic type represents either unset logic, bad resolution or undriven nets. That’s one bit representing three things, using multiple bits to represent bad behavior individually improves the tools understanding of the state of the design, and having the information orthogonal to values stops verification being blocked by those issues. E.g. if a resolver that needs Vdd/Vss can’t find those it can set a (un)certainty bit and use the UPF power domain as a default, that allows verification to proceed with partially wired designs rather than being blocked by signals just going to “X”.

CDC/RDC

Models that handle Silicon/timing variability can be mixed with regular models, either substituting for standard models or as a parallel model. Possible timing failures can be communicated as (real number) failure probabilities, rather than single bit flags. Similarly information about clock jitter for BER calculations, i.e. with a SERDES block driving a channel, the channel model can add to the jitter given crosstalk analysis information.
A nature indicating timing spread/jitter can be driven at the same time as a logic value from (new) models, and downstream models that don’t need that information would be able to ignore it, but for verification purposes the adapter could randomly vary delivery to receivers as part of constrained-random testing. E.g. given min/typ/max timing information the adapter can randomly pick min/max timing instead of typ.

Implementation/Prototyping

The driver-access functionality can be implemented by generating the driver/receiver network as regular SytemVerilog code, i.e. you just treat the drivers and receivers as regular signals that are aliased to the actual driver/receiver objects. If you wanted to do a user-space version you could rewrite the source where drivers and receivers are declared explicitly, and then connected to the adapter by cross module reference. Access to UPF/C++ can be achieved by translating the “find” functions to DPI based on their arguments.

In rewriting the code assignment with a delay would be split into a two-part assignment with an undelayed signal that can be used for the *waveform* signal.
In a similar vein, the proposal can be restated in rewrite rules from new syntax to existing language.

Extra Functionality

Defect Injection

The *find* function on drivers and receivers can be used to indicate a component is in a defective state and have it disabled by a (super) adapter, which would support IEEE P2427 defect injection in a standard way.

Discipline Methodology

Disciplines were added to Verilog-AMS to indicate the physical nature of the nets in simulation, *electrical* being the default. It is orthogonal information to driver types, and is common to all drivers and receivers on a net, i.e. if a net is electrical then only various forms of voltage and current drivers are allowed, trying to add a thermal driver would be disallowed. This got syntactically confused in the implementation because Cadence wanted to use electrical only for analog nets, rather than extend it to *wreal* or recognize it in digital models. Obviously, *electrical* can apply to UDNs like Thevenin types, without changing the operation of anything.

During the development of Verilog-AMS, as with this effort for UDNs, it was necessary to find a place to hang information for things like port/driver impedance and logic thresholds, Vdd/Vss names etc., so the discipline mechanism was extended to be a general purpose attribute scheme with inheritance so that a single discipline declaration will give you all the attributes you need to guide the connect-rules for insertion of the best *connectmodule*, it can also include simulator directives for precision and initial conditions.

Automation

The intended methodology was to create disciplines for individual ports in library cells during cell characterization, some information will be common to all cells, and some ports will have the same attributes within the cell library, so port disciplines are derived from library level disciplines, and are expected to be limited in number. Tagging cell ports with things like Vdd/Vss information allows tools to check for misuse (outside simulation), as well as finding the best connect-modules.

Application to UDNs

Given libraries characterized and tagged with disciplines, the driver/receiver *.find(...)* methods would be used to access the information, e.g. *in.driver[0].find(`DISCIPLINE)* could return the string name of the driver's discipline, and a secondary function can look up attributes based on that name. The secondary lookup would give you (say) the Vdd or Vss net name for that cell/port, and you can do another call, e.g. *.find(`DR_CONTEXT_NET,<net name>),* to get a handle for it.

[Alternative] API Approach

An alternative to language extensions is to use a C/C++ API to external software that does resolution in a language independent manner. In this case drivers and receivers are registered with the external resolver, with callbacks being used to get information and drive receivers. Having such an API is useful in addition to the language extensions when mixing multiple languages, i.e. an extended SystemVerilog simulator can let co-simulators connect extra drivers and receivers into a net.
An external resolver has more flexibility in handling runtime changes like the addition of debug or back-annotation, since it doesn't have the static construction limitations of SystemVerilog. The API approach can run into limitations if the driver/receiver user-defined data is complex structures rather than scalar types, although the individual nature segments being scalar in a UDN mitigates that somewhat, and using common compiler/debug information can fill the gaps.

From within a SystemVerilog simulator the API is fairly simple, the UDN just needs to be marked as resolving externally (with the external resolving registration function).

Dynamic Resolvers

A nice-to-have feature is the ability to handle hardware being reconfigured during runtime, that's somewhat easier with an API that allows dynamic registration/de-registration, and helps support scenarios with multiple simulators. If there is no state held by the resolution stack then the module version can be re-instantiated as needed, a function based stack would allow that

automatically. The state of the resolution would be held in the future waveforms on the receivers, or in the driver/receiver contexts.

DIY Resolution by Translation

As mentioned above, you can view drivers and receivers as being like signals themselves, and you can just treat them that way literally. In net-listing from (say) OpenAccess you can use a generic Voltage/logic nettype on all nets, and dissociate the drivers and receivers so that they are just local signals in the module. The connection from the drivers and receivers is then handled by instantiating a (top-level) module to do resolution that connects with all of those by cross-module reference, or by using the API approach. With the appropriate glue, the resolver/adapter modules will be the same as those that work with the official version of mixed-nettype resolution when it arrives.