Exploring New Frontiers of High-Performance Verification with UVM-AMS

Tim Pylant, Cadence Design Systems, UVM-AMS WG Vice-Chair
What Does DMS/AMS Refer To?

- **AMS**: Mixed signal with electrical (includes schematics/Spice, Verilog-AMS, Verilog-A) – Spectre and Xcelium
- **DMS**: Mixed signal with discrete real numbers (no electrical) – Xcelium only
- **Pure Digital**: Only logic (SystemVerilog/Verilog/VHDL) – Xcelium only
Traditional MS Design
Cadence View of Mixed-signal Verification Issues

• Mixed-signal verification issues in today’s flow
  • Performance vs. accuracy of analog models
    • Transistor models execute too slow for SoC verification
    • Verilog behavioral models have higher performance but do not accurately represent analog models
    • Verilog model not kept in sync with analog model
  
• Working with digital designs
  • Connectivity errors not being caught
  • Spec assumptions not verified between analog and digital
  • Lack of coverage of the analog design across MS boundary
Real-wire Coercion

- When a wire or interconnect is connected to a net of the type `wreal`, SystemVerilog `real`, or VHDL `real`, it is coerced (forced) to `wreal`.
- Coercion can occur across multiple hierarchical levels.
- The coercion process allows a seamless connection of devices without worrying about the interconnects and their types.
- Different configuration or interconnect might be used to connect electrical ports.
- It allows for different design/model abstractions without recoding interconnects.
Auto-Inserted Connect Modules (AICM)

- The concept of automatic inserted connect modules applies
  - Between discrete (logic and real) and continuous, such as electrical domain (E2L/L2E, R2E/E2R)
  - Between two discrete domains, such as real to logic connection (R2L/L2R)
- CMs allow design blocks to be seamlessly switched without needing to recode the interconnect or port types. It supports multiple power supply sensitivities used in value conversion.
Disciplines

• Continuous disciplines
  • electrical
    • Potential and flow
    • Kirchhoff’s laws KCL/KVL
  • voltage
    • Only potential

• Discrete disciplines
  • logic

• Discipline resolution algorithm
  • Incompatible boundaries have CM/IEs auto-inserted during elaboration phase
  • Types of IEs: E2R/R2E, L2R/R2L, E2L/L2E, Bidirectional RNM, Bidirectional conservative flow
AMS Xcelium Use Model

- Sign off analog IP with AVUM flow
- Schematics => Generate SPICE netlist
- Plug and play into SV UVM testbench (TB re-use)
- Verify critical path scenarios, i.e. POR, timing, dynpower, power seq, etc..

Analog/AMS Design Verification

Digital Design Verification
AMS Control File (amsd block)

The “heart of the AXUM flow” ease of use

• Create and configure with the AMS Control file(amscf.scs)
  • Spice block substitution (portmap card)
  • Cell, instance Bindings (config card)
  • Interface element declarations (ie card)
  • Multiple supplies (ie card)
  • Verilog-Spice-Verilog sandwiching(use=spice, use=hdl)
  • Includes Spice file pointers and analog control file

• Read directly by xrun:
  % xrun amscf.scs <other_files_and_options>

• amsd block settings assign cell and instance bindings using AMS xrun binding engine
If we have automated technologies that take care of coercion and inter-discipline connections, why do we need UVM-AMS?
MS-SoC: Mixed simulation env using RNMs & analog models
Classical UVM Example

uvm_agent (UVC)

- sequencer
- driver
- config
- monitor

? ? ?

top

DUT

registers
What Is UVM-MS?

• Define a way to extend UVM to AMS/DMS
  • Modular and reusable testbench components
  • Sequence-based stimulus
  • Take advantage of UVM infrastructure as much as possible

• Reuse as much UVM as possible as DUT is refined from digital to AMS
  • Use extension/factory as much as possible
  • Support UVM architecture for DMS/AMS DUT from the start

• Define standard architecture for D/AMS interaction
  • Minimize traffic across the boundary
  • Enable development of D/AMS VIP libraries and ecosystem
Generating/Driving Continuous Analog Signals

• An analog signal that is not simple DC or a slow-changing signal must be a periodic waveform like a sine wave, a sawtooth, or some composition of such sources.

• For example, a signal generator for a sine wave can be controlled by four control values, determining the freq\( (1/\lambda) \), phase\( (\Phi) \), amplitude\( (A) \), and DC bias\( (v) \) of the generated signal.

• The properties of the analog signal being driven are controlled by real values generated by the sequencer.

• A UVM sequence_item contains fields for all the control parameters.

• The driver converts the transaction to a setting for the signal generator.
Requirements

- Minimal changes to UVC to add MS capabilities (driver, monitor, sequence item) that can be applied using `set_type_override_by_type`
- Define analog behavior based on a set of parameters defined in a sequence item and generate that analog signal using an analog resource (MS Bridge)
- Measure the properties of the analog signal, return them to a monitor, and package those properties into a sequence item
- Drive and monitor configurations, controlled by dedicated sequence items and support easy integration into multi-channel test sequences
- Controls can also be set by way of constraints for pre-run configurations.
- Collect/check coverage in the monitor based on property values returned from the analog resource or add checkers in the analog resource
EEnet Modeling
SystemVerilog 2012 Extended Nettype Capabilities

• Definition of “nettype” construct applicable to any datatype
  • Application to real datatype provides construct equivalent to VAMS “wreal”
    • Cadence package “cds_rnm_pkg” defines nettypes for SV identical in name & operation to VAMS wreal flavors: wreal1driver, wreal4state, wrealavg, wrealsum, etc
• User Defined Type (UDT) can use a struct of multiple values to define a net
• User Defined Resolution (UDR) functions can define how the net should merge multiple drivers of the UDT format to define the resultant net value
  • This extends the possibilities of how interfaces between blocks are defined
    • Cadence “EE_pkg” defines a nettype to define an electrical interface
    • Multiple drivers can each drive the net with voltage or current and resistance values
    • Resolution function computes resulting net voltage
How Is an EEnet Defined?

• The “EEnet” UDT specifies three fields:
  • $V$ = voltage driving the net
  • $I$ = current driving the net
  • $R$ = resistance driving the net

• This allows lots of options for how the net can be driven:
  • Specify $V$ and $R$ with $I=0$ for voltage with series resistance
  • Specify $I$ and $R$ with $V=0$ for current with parallel resistance
  • Specify $V$ with $R=0$ for ideal voltage source
  • Specify $I$ with $R=`wrealZState` for ideal current source

• Resolution with included UDR function provides:
  • $V$ = resolved node voltage, or `wrealXState` if multiple ideal voltage drivers
  • $I = 0$ normally, or current through voltage source if driven by ideal voltage source
  • $R$ = effective impedance at node (parallel combination of all connected resistances)

• Re-evaluated whenever any driver changes
Simple example of an EEnet driver code: V+R driving a node

```plaintext
import EE_pkg::*;       // access the definitions in EE_pkg
module myVRdrv(        // declare the voltage+resistance driver
    inout EEnet P,      // EEnet pin is inout to allow both read & write
    input real Vval,    // voltage value to drive to net
    input real Rval,    // resistor value to drive to net
    output real Imeas)  // measured current from pin thru V+R to ground
);

assign P = '{Vval,0.0,Rval};  // drive voltage & resistance onto net
assign Imeas = (Rval==0)? P.I : (P.V-Vval)/Rval; // measure current
endmodule
```

![Diagram of a voltage and resistor circuit driving a current](image-url)
Simple RC response using CapGeq model

- Sample rate of $\tau/4$ generates points typically within 0.1% of analog waveform
- Sample rate equal to $\tau$ still has well-controlled error considering large step size
I\textsuperscript{2}C example Using EEnet

```verilog
import EE_pkg::*;        // access the definitions in EE_pkg
module i2c_target (        // I\textsuperscript{2}C SDA pin
    inout EEnet SDL,     // I\textsuperscript{2}C SCL pin
    inout EEnet SDL
);
    assign SDA = '{Vval,0.0,Rval};  // drive voltage & resistance onto net
    assign SCL = '{Vval,0.0,Rval};  // drive voltage & resistance onto net
endmodule
```

![I\textsuperscript{2}C circuit schematic]
Why Do We Need UVM-MS for I²C?

• Cannot create analog behaviors from UVM class-based objects

• Need mechanism to control analog parameters
  • Parameters not part of interface
Applying UVM-MS to EEnet Model
Overall UVM-MS Methodology

- MS Bridge is the proposed layer that sits between the UVC and the (A)MS DUT
- MS Bridge is a SV module that consists of a proxy API, SV interface, and an analog resource module
- The ‘proxy’ is an API that conveys analog attributes between the UVC and the MS Bridge
- The SV ‘intf’ passes digital/discrete signal values (logic, real, nettype/RNM) between UVC and MS Bridge
- The analog resource (SV, Verilog, or Verilog-AMS)
• MS testbench may require the behavior and presence of analog components that a typical UVM-RTL testbench could not include. These could be:
  • Capacitors, Resistors, Inductors, Diodes, current/voltage sources etc. Or a complex passive network for multiple DUT pins.
  • A piece of Verilog-AMS code
  • Such components will be used to model the analog behavior of PADs, lossy transmission lines, loads/impedances, or any other voltage/current conditioning required to accurately model the signals connecting to the ports of DUT
  • Those components can be placed inside the analog resource to be controlled by proxy.
Proxy “Hook-Up”

Proxy Template (API)

```verilog
UVC package
class i2c_bridge_proxy extends uvm_ms_proxy;
...
  pure virtual function void set_capitance(...);
...
endclass
```

Implement

UVM config setting

```verilog
module top;
  ...
i2c_bridge i2c_bridge(.sda, .scl);
  ...
  initial begin
    uvm_config_db#(osc_bridge_proxy)::set(null,"*12c*","bridge_proxy",top.i2c_bridge.__uvm_ms_proxy);
    run_test();
  end
endmodule
```

Proxy instance in MS Bridge module

```verilog
module osc_bridge(...);
...
osc_bridge_core #(...) core (...); // AMS model
...
class proxy extends osc_bridge_proxy;
...
  function void set_capitance(input real cap_val);
    core.cap_val = cap_val;
  endfunction
endclass
proxy __uvm_ms_proxy = new();
...
always_comb
  __uvm_ms_proxy.delay = core.delay;
...
endmodule
```

Instance of analog resource

Passes values to analog resource to “program” waveform

Passes values to UVC component to “monitor” waveform
Push

```
class i2c_proxy ...
    function void push_Vsup(input real Vsup);
        core.Vsup = Vsup;
    endfunction
endclass
```

Pull

```
class osc_proxy ...
    function void get_measures(...);
        ampl = core.ampl_out;
        bias = core.bias_out;
        phase = core.phase_out;
        freq = core.freq_out;
    endfunction
endclass
```

Monitored

```
int delay;
endclass : osc_proxy
```

```
always_comb begin
    _uvm_ms_proxy.delay = core.delay;
end
```

```
class osc_bridge_core (...);
...
real Vsup;

analog begin
    vsin = (ampl_in * sin(M_TWO_PI * freq_in * $abstime));
    ...
end
```

```
real ampl_out;
real bias_out;
real freq_out;
real phase_out;
```

```
Vsig = V(sig);
if (Vsig > max_a)
    max_a = Vsig;
else if (Vsig < min_a)
    min_a = Vsig;
```

Interpolated value

If target is different, it’s seen as a D2A event

Analog generates update
I²C DUT

TOP

SWTOP

I²C UVC

sda_drive

scl_drive

push_V

OSC

UVC

HWTOP

I²C INTF

SCL

SDA

I²C Bridge

Analog Resource

Proxy

REGISTER

FREQ_ADPT

DUT

SCL

SDA

sda_drive

scl_drive

push_V
I²C Waveforms

EEnet signals

Logic signals
Freq Adapter Waveforms
module freq_adapter (
    output logic CLKOUT_P, CLKOUT_N; // differential output
    input logic CLK_IN;              // clock input
    input logic sdl, sda;            // I2C interface
    input logic [7:0] pw_adj, [1:0] sr_adj, ampl_adj;
);

uvm_agent (UVC)
- sequencer
- driver
- config
- monitor

Top
- intf
- DUT (digital)
Model of Frequency Adapter Ports in SV RNM

```verilog
module freq_adapter import cds_rnm_pkg::*;

output wreal4state CLKOUT_P,CLKOUT_N; // differential output
input wreal4state CLK_IN;              // clock input
input EEnet sdl, sda;                 // I2C interface
input logic [7:0] pw_adj, [1:0] sr_adj, ampl_adj;

uvm_agent (UVC)

sequencer
config

driver
monitor

Top
MS Bridge

analog resource

DUT (RNM/Elec)

RNM uses event solver so just need to convert logic to real voltage
```
Example Walkthrough

UVM digital to UVM-MS
Steps to Create a UVM-MS UVC

• Create Bridge module
  • Contains Analog Resource and Proxy

• Extend classes for Driver
  • Use set_type_override_by_type to use extended classes
module i2c_bridge import EE_pkg::*; (
  input wire scl_drive, // digital signal from driver
  input wire sda_drive, // digital signal from driver
  input EEnet SDA, SCL // EEnet signals connected to DUT
); //

parameter real Vsup_p = 1.8;
parameter real Rout = 100.0;

//UVM + MS extras
import uvm_pkg::*;
import uvm_ms_pkg::*;
'include "Uvm_macros.svh"
'include "uvm_ms.svh"

//UVM package for this component
import i2c_pkg::*;

//Create an instance of a component that can be used with uvm_info context to print
// messages from analog resource using uvm_ms_info macro/function
uvm_ms_reporter

//Class proxy extends the osc_bridge_proxy included in osc_pkg.sv
//The implementation for the config wave push function is defined here
class proxy extends i2c_proxy;
  function new(string name = "proxy");
    super.new(name);
  endfunction : new

  // implementation of function to push Vsup value to analog resource
  function void push Vsup(input real Vsup);
    core.Vsup = Vsup;
  endfunction

endclass

proxy __uvm_ms_proxy = new("__uvm_ms_proxy");

//Analog resource instantiation
i2c_analog_resource #(Vsup_p(Vsup p), Rout(Rout)) core {
  .sda_drive, .scl_drive, .SDA, .SCL
};
endmodule
i2c_driver → i2c_ms_driver

UVM

```plaintext
class i2c_driver extends uvm_driver #(i2c_packet);
`uvm_component_utils(i2c_driver)

virtual interface i2c_if vif;

real Vsup = 1.8;

function new(string name = "i2c_driver", uvm_component parent);
  super.new(name, parent);
endfunction

virtual task run_phase(uvm_phase phase);
  forever begin
    seq_item_port.get_next_item(req);
    `uvm_info(get_type_name(), $sformatf("Sending Packet :\n%%", req.sprint()), UVM_LOW)
    void'(begin tr(req, "Input I2C Packet");
      data(req.data);
      id(req.tg_addr);
      rw_(req.rw_);
    end_tr(req);
    `uvm_info(get_type_name(), "Packet sent", UVM_DEBUG)
    seq_item_port.item_done();
  end
endtask

function void connect_phase(uvm_phase phase);
if (!uvm_config_db#(virtual i2c_if).get(this, ",", "vif", vif))
  `uvm_error("NOVIF", "vif not set for ", get_full_name(), ".vif")
endfunction

function void start_of_simulation_phase(uvm_phase phase);
  `uvm_info(get_type_name(), "Call I2c_driver", UVM_HIGH);
endfunction
endclass
```

UVM-MS

```plaintext
class i2c_ms_driver extends i2c_driver;
`uvm_component_utils(i2c_ms_driver)

virtual interface i2c_if vif;

real Vsup = 1.8;

function new(string name = "i2c_ms_driver", uvm_component parent);
  super.new(name, parent);
endfunction

virtual task run_phase(uvm_phase phase);
  i2c_proxy_p.push_Vsup(Vsup);
  forever begin
    seq_item_port.get_next_item(req);
    `uvm_info(get_type_name(), $sformatf("Sending Packet :\n%%", req.sprint()), UVM_LOW)
    void'(begin tr(req, "Input I2C Packet");
      data(req.data);
      id(req.tg_addr);
      rw_(req.rw_);
    end_tr(req);
    `uvm_info(get_type_name(), "Packet sent", UVM_DEBUG)
    seq_item_port.item_done();
  end
endtask

function void connect_phase(uvm_phase phase);
if (!uvm_config_db#(i2c_proxy_base).get(this, ",", "i2c_proxy_p", i2c_proxy_p))
  `uvm_error("NOPROXY", "i2c_proxy not set for ", get_full_name(), ".i2c_proxy_p")
endfunction
```
freq_adpt_tb → freq_adpt_ms_tb

UVM

class freq_adpt_tb extends uvm_env;

// component macro
5  uvm_component_utils(freq_adpt_tb)
6
7  registers_env registers;
8  osc_env freq_generator;
9  osc_env freq_detector;
10
11  freq_adpt_tb #freq_adpt_ms_tb freq_adpt_sb;
12
13  // Constructor
14  function new (string name, uvm_component parent=null);
15      super.new(name, parent);
16  endfunction : new
17
18  // UVM build() phase
19  function void build_phase(uvm_phase phase);
20      uvm_info("MSG", "In the build phase", UVM_MEDIUM);
21      // set up virtual interfaces for UVMs and scoreboard
22      uvm_config_db#(virtual_if)#set(this, "freq_generator", "vif", top_generator_if);
23      uvm_config_db#(virtual_if)#set(this, "freq_detector", "vif", top_detector_if);
24      uvm_config_db#(virtual_if)#set(this, "registers", "vif", top_reg_if);
25      uvm_config_db#(virtual_if)#set(this, "freq_generator", "diff_set", 0);
26      uvm_config_db#(virtual_if)#set(this, "freq_detector", "diff_set", 1);
27      super.build_phase(phase);
28
29      // create the envs for the generator, detector, registers and scoreboard
30      freq_generator = osc_env::type_id::create("freq_generator", this);
31      freq_detector = osc_env::type_id::create("freq_detector", this);
32      registers = registers_env::type_id::create("registers", this);
33      freq_adpt_tb = freq_adpt_ms_tb::type_id::create("freq_adpt_tb", this);
34  endfunction : build_phase
35
36  // UVM connect phase
37  function void connect_phase(uvm_phase phase);
38      // Connect the TLM ports from the UVMs to the scoreboard
39      freq_generator->monitor->item_collected_port.connect(freq_adpt_tb, registers_in);
40      freq_generator->monitor->item_collected_port.connect(freq_adpt_sb, registers_i);
41      registers_env->monitor->item_collected_port.connect(freq_adpt_tb, registers_out);
42      registers_env->monitor->item_collected_port.connect(freq_adpt_sb, registers_o);
43
44  endfunction : connect_phase

UVM-MS

class freq_adpt_ms_tb extends freq_adpt_tb;

// component macro
59  uvm_component_utils(freq_adpt_ms_tb)
60
61  //freq_adpt_ms_scoreboard freq_adpt_sb;
62
63  // Constructor
64  function new (string name, uvm_component parent=null);
65      super.new(name, parent);
66  endfunction : new
67
68  // UVM build() phase
69  function void build_phase(uvm_phase phase);
70      // if UVM MS
71      // set up bridge proxy pointer references to generator and detector UVMS
72      uvm_config_db#(osc_bridge_proxy)#set(this, "freq_generator_agent.", ","bridge_proxy", top_generator_bridge._uvm_ms_proxy);
73      uvm_config_db#(osc_bridge_proxy)#set(this, "freq_detector_agent.", ","bridge_proxy", top_detector_bridge._uvm_ms_proxy);
74      // endif UVM MS
75
76      // if UVM AMS
77      uvm_config_db#(uvm_ms_proxy)#set(this, ","12c_agent.", ","12c_proxy", top.i2c_bridge._uvm_ms_proxy);
78      // endif UVM AMS
79
80      // override driver, monitor, and scoreboard with UVM-AMS versions
81      set_type_override_by_type(freq_generator_driver::get_type());
82      set_type_override_by_type(freq_detector_driver::get_type());
83      set_type_override_by_type(freq_monitor::get_type());
84      set_type_override_by_type(freq_source::get_type());
85      set_type_override_by_type(freq_source_monitor::get_type());
86      set_type_override_by_type(freq_adpt_scoreboard::get_type());
87      super.build_phase(phase);
88
89  endfunction : build_phase
90
91 endclass : freq_adpt_ms_tb
Demo
UVM Messaging
Messages for Debug and Error Reporting

• Debugging activity inside a large environment with many UVCs is critical
• Need to report:
  • Errors
  • Debug
  • Progress
• Messages need to be categorized via severity:
  • Fatal, Error, Warning, Info
• Need to link actions with messages
  • Stop simulation on fatal or after four errors
  • Summarize number of messages reported
• Need a different mechanism than simulator messages to avoid filtering effects
UVM Messaging System
UVM Messaging from Analog Resource

- UVM Reporting macros not supported in Verilog-AMS modules
  - Take advantage of up-scoping to access SV bridge
- `include “uvm_ms.vamsh” in Verilog-AMS analog resource or
  `include “uvm_ms.vdmsh” in SystemVerilog analog resource
  - localparams to define UVM Verbosity levels as integers to match UVM enum
  - Macros to wrap the uvm_ms_* reporting function calls defined in uvm_ms.svh
- `include “uvm_ms.svh” in MS Bridge (SV)
  - Definitions of the functions called by analog resource
  - Provides macros for `uvm_ms_[info|warning|error|fatal](...)”
  - Utilizes the “__uvm_ms_proxy” declaration as the originating path for analog resource UVM messages
UVM Messaging Example for Verilog-AMS Resource

• Use analog domain to detect the issue and toggle a flag
• Flag is detected by absdelta to then report the message via the digital engine
• Example

```verilog
analog begin
  if((I_PLUS > 1.0) && !IThr_triggered) IThr_triggered = 1;
  else if(I_PLUS < 0.9) IThr_triggered = 0;
end

// Convert the detection in the analog block to a UVM report.
string message;
always@($(absdelta(I_thr_triggered,1,0,0,1))) begin
  $sformat(message,"The Current is above the thresholds @ %e",I_PLUS);
  if(I_thr_triggered) `uvm_ms_error(P__TYPE,message)
end
```

Up-scope function call
UVM Message – Analog block

“uvm_ms.vamsh” uvm_ms_info function is found via up-scope and executed from SV bridge

```vhdl
`define uvm_ms_info(id,message,uvm_verbosity) \
  uvm_ms_info(id,message,uvm_verbosity,$sformatf(“%m”),__FILE__,__LINE__);
```

**osc_core.vams**

```vhdl
`include “uvm_ms.vamsh”
```

```
`define uvm_ms_info(id,message,uvm_verbosity,$sformatf("freq=%e Hz period=%e ns", freq_in, out_period),\UVM_MEDIUM)
```

“uvm_ms.svh”

```vhdl
function void uvm_ms_info(id,message,uvm_verbosity,uvm_path,__FILE__,__LINE__);
  uvm_component CTXT;
  CTXT=uvm_ms_get_bridge_path(uvm_path); // get path to uvm_component in top.bridge
  CTXT.uvm_report_info(id,message,uvm_verbosity'(verbosity_level),file,line);
endfunction
```

UVM_INFO ../uvc_lib/osc/vams/osc_bridge_core.vams(98) @ 52001.098068ns: top.detector_bridge

[FREQ_UPDATE] The Current is above the threshold @ 1.178812e+00A

**osc_bridge.sv**

```vhdl
`include “uvm_ms.svh”
```

```
`include “uvm_ms.svh”
```

```
`define uvm_ms_info(id,message,uvm_verbosity,$sformatf("freq=%e Hz period=%e ns", freq_in, out_period),\UVM_MEDIUM)
```

```
function void uvm_ms_info(id,message,uvm_verbosity,uvm_path,__FILE__,__LINE__);
  uvm_component CTXT;
  CTXT=uvm_ms_get_bridge_path(uvm_path); // get path to uvm_component in top.bridge
  CTXT.uvm_report_info(id,message,uvm_verbosity'(verbosity_level),file,line);
endfunction
```

UVM_INFO ../uvc_lib/osc/vams/osc_bridge_core.vams(98) @ 52001.098068ns: top.detector_bridge

[FREQ_UPDATE] The Current is above the threshold @ 1.178812e+00A
Conclusion

• There is a need for more advanced, standard methodologies for scalable, reusable and metric-driven mixed-signal (AMS/DMS) verification

• The UVM-AMS WG proposal addresses the gaps in current verification methodology standards

• Extend UVM class-based approach to seamlessly support the module-based approach (MS Bridge) needed for mixed-signal verification
  • Targeting analog/mixed-signal contents (RNM, electrical/SPICE)
  • Application and extension of existing UVM concepts and components
    • Sequencer, Driver, Monitor
    • MS Bridge / Analog resources
    • UVM Messaging System
Questions?