Migrating from UVM to UVM-MS

Accellera UVM-AMS Working Group

Tim Pylant, Cadence Design Systems, UVM-AMS WG Vice-Chair
UVM-AMS WG Member Companies

- Renesas
- Cadence
- Siemens EDA
- Qualcomm
- NXP
- Synopsys
- Texas Instruments
Why UVM-AMS

• Same reason as UVM – explosion of verification needs
  • Verifying analog functionality/connectivity under large set of digital configurations
  • Digital control system transitions interacting with analog functions
  • Dynamic control between analog & digital circuits under wide range of conditions
  • Finding problems with A/D interaction in unexpected corner cases

• Standard methodology
  • Plug & play reuse of existing UVM components
  • Rich debug and messaging scheme integrated with simulator
What Are We Trying to Do?

• Define a way to extend UVM to AMS/DMS
  • Modular, 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 boundary
  • Enable development of D/AMS VIP libraries & ecosystem
Classical UVM Example
Terminology

• Analog Mixed-Signal (AMS) simulation and verification refers to systems that can simulate/verify analog/mixed-signal designs as a co-simulation of digital + analog (electrical) solvers

• Digital Mixed-Signal (DMS) simulation and verification refers to systems that can simulate/verify analog/mixed-signal designs within a discrete event-driven solver as digital (logic) and real number models
Requirements

• Minimal changes to agent 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 analog resource or add checkers in analog resource
What is needed to move from UVM to UVM-MS
Generating/Driving Discrete Analog Signals

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

• Classical RNM would drive real numbers from UVC sequence/driver within the agent

• In AMS this would generate too many D2A events or not give enough finesse to the signal
Generating/Driving Continuous Analog Signals

• A signal generator for a continuously changing signal can be controlled by four properties determining the \( \text{freq}(1/\lambda) \), phase(\( \Phi \)), amplitude(\( A \)), and DC bias(\( \nu \)) 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 passes the fields of the sequence_item to the controls for the signal generator.
• MS Bridge is the proposed layer that sits between the agent and the (A)MS DUT and consists of a proxy API, SV interface, and an analog resource module.

• The ‘proxy’ is an API that conveys analog attributes between the agent and the MS Bridge.

• The SV ‘intf’ passes digital/discrete signal values (logic, real, nettype/RNM) between agent and MS Bridge – can be left in top or moved to Bridge.
Verilog-AMS Simulator DC OP

• DC Op – Steady State operating point of all the nodes/branch currents
• Understanding of UVM-MS DC OP is important
  • Need to make sure initial conditions (caps, supplies, timesteps) start with valid values for proper convergence
  • Enable UVM DUT configuration before analog circuit initialization (DC Op).
    • E.g. make a cap open for a particular test before DC op
  • Using #0 is not good practice as it shows poor coding and understanding of the simulator(s) scheduler
• Must raise a UVM objection before DC OP otherwise the simulation finishes

```verilog
virtual task my_ams_test::run_phase(uvm_phase phase);
  ...
  phase.raise_object(this);
  if ($realtime <= 0.0) #1step;
  `uvm_info("TEST", "AMS DC-OP finished", UVM_MEDIUM)
  my_seq.start(my_seqr); // Launch sequence(s)
  ...
  phase.drop_object(this); // Test termination
endtask: my_ams_test
```

Ensures time is consumed
Verilog-AMS Simulator Scheduling - Transient

- Analog engine always leads
- Digital to analog events cause matrix re-evaluation and timestep backtrack
- Most simulators see any digital var in the analog block as a D2A to monitor

@cross, above, timer or internal time-step

Some digital to analog event
Verilog-AMS Best Practices

• Variables are ‘owned’ by one engine but can be read by another

• AMS can’t access digital variables that are dynamic (everything in the matrix is fixed at time 0)

• Generally, avoid ‘string’ datatypes in Verilog-AMS as support is inconsistent and the LRM is not clear

• OOMR to analog-owned variables not allowed – they are not part of the analog matrix
Proxy “hook-up”

UVC package

class osc_bridge_proxy extends uvm_ms_proxy;
    ...
    pure virtual function void config_wave(...);
    ...
    real freq_out;
    ...
endclass

module top;
    ...
    osc_bridge osc_bridge(.clk_outp, .clk_outn, .clk_in);
    ...
    initial begin
        uvm_config_db#(osc_bridge_proxy)::set(null,"*freq_adpt*","bridge_proxy",top.osc_bridge.__uvm_ms_proxy);
        run_test();
    end
endmodule

Proxy instance in MS Bridge module

module osc_bridge(...);
    ...
    osc_bridge_core #(...) core (...); // AMS model
    ...
    class proxy extends osc_bridge_proxy;
    ...
    function void config_wave(input real ampl, bias, phase, freq);
        core.ampl_in = ampl;
        core.bias_in  = bias;
        core.phase_in = phase;
        core.freq_in  = freq;
    endfunction
    endclass

proxy __uvm_ms_proxy = new();
    ...
    always_comb
        __uvm_ms_proxy.freq_out = core.freq_out;
    ...
endmodule

Implement

UVM config setting

Instance of analog resource

Proxy instance in MS Bridge module

Proxy “hook-up”

Proxy instance in MS Bridge module

Passes values to agent component to “monitor” waveform

Passes values to analog resource to “program” waveform

UVM config setting

module top;
    ...
    osc_bridge osc_bridge(.clk_outp, .clk_outn, .clk_in);
    ...
    initial begin
        uvm_config_db#(osc_bridge_proxy)::set(null,"*freq_adpt*","bridge_proxy",top.osc_bridge.__uvm_ms_proxy);
        run_test();
    end
endmodule

Proxy “hook-up”

UVM config setting

module top;
    ...
    osc_bridge osc_bridge(.clk_outp, .clk_outn, .clk_in);
    ...
    initial begin
        uvm_config_db#(osc_bridge_proxy)::set(null,"*freq_adpt*","bridge_proxy",top.osc_bridge.__uvm_ms_proxy);
        run_test();
    end
endmodule

Proxy “hook-up”

UVM config setting

module top;
    ...
    osc_bridge osc_bridge(.clk_outp, .clk_outn, .clk_in);
    ...
    initial begin
        uvm_config_db#(osc_bridge_proxy)::set(null,"*freq_adpt*","bridge_proxy",top.osc_bridge.__uvm_ms_proxy);
        run_test();
    end
endmodule
Proxy ↔ Analog Resource

MS Bridge

```plaintext
class osc_bridge_proxy;
    function void config_wave(...);
        core.ampl_in = ampl;
        core.bias_in = bias;
        core.phase_in = phase;
        core.freq_in = freq;
    endfunction

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

endclass
```

Push

```plaintext
osc_bridge_core (...);
...
real ampl_in;
real bias_in;
real phase_in;
real freq_in;

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;
```

Pull

Interpolated value

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

Monitored

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

//real min, max; //from base class
always_comb begin
    __uvm_ms_proxy.min = core.min_a;
    __uvm_ms_proxy.max = core.max_a;
end

Analog generates update
UVM Phasing Requirements for UVM-MS

• Analog resources will have parameters and UVM should have a means to read/modify/write them before simulation consumes time
• Implement methods getParameters() / setParameters() in proxy
• Use existing UVM phases to guarantee read/modify/write order

<table>
<thead>
<tr>
<th>UVM Phase</th>
<th>What should happen for AMS resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>build</td>
<td></td>
</tr>
<tr>
<td>connect</td>
<td>Read parameters values from ‘SV+VAMS’ module (Instrument/Passive) into the agent’s configuration.</td>
</tr>
<tr>
<td>end_of_elaboration</td>
<td>Modify agents parameters based on test requirements</td>
</tr>
<tr>
<td>start_of_simulation</td>
<td>Apply agents parameters to ‘SV+VAMS’ module (Instrument/Passive)</td>
</tr>
</tbody>
</table>
| run               | Must consume some time to allow DC OP to happen before agents drive sequence items so that synchronization system works. Recommend task run_phase() ;
                   | if($realtime <= 0.0) #1step; // cause a DC OP to occur                                              |
Analog Resource Configuration

• Analog components tend to be placed with initial values as parameters. e.g. a decoupling cap on an LDO output

• Allow the MS Bridge to have parameters that are copied from UVM configuration in connect_phase

• Test cases can override the configuration, which is then set in the analog resource in start_of_simulation_phase

```plaintext
class osc_bridge_proxy;
    function res_config getParameters();
        res_config cfg = new();
        cfg.res_val = i_core.rseries_val;
        ... return(cfg);
    endfunction

    function void setParameters(res_config cfg);
        i_core.rseries_val = cfg.res_val;
        ...
    endfunction

    osc_bridge_core (...);
    ...
    parameter res_val = 200;
    ...
    // Initial values set from parameter, then set by setParameter in proxy
    real rseries_val = res_val;
    ...
```
UVM-MS Phasing

connect

end_of_elaboration

start_of_simulation

run

Read VAMS params into UVM cfg

virtual function void
my_driver::connect_phase(...) {
    cfg.copy(proxy.getParameters());
    endfunction : connect_phase

Set VAMS initial values before t=0

virtual function void
my_driver::start_of_simulation_phase(...) {
    proxy.setParameters(cfg);
    endfunction : start_of_simulation_phase

(Optional) Modify params from UVM test

virtual function void my_test::end_of_elaboration_phase(...) {
    env.agent.cfg.rseries = 1e4; // 10k rseries in this test
    endfunction : end_of_elaboration_phase
Example Walk-through

UVM digital to UVM-MS
Frequency_Adapter DUT

clk_in

en_mux

sel_mux[1:0]

fx2

f/2

f/2

4:1

pw_adj[8]

ampl_adj[2]

clkout_p

clkout_n

sr_adj[2]
UVM TB – add analog capability
Freq_adapter Waveforms

Digital

VAMS

Digital clks

Analog clks
Model of Frequency Adapter Ports in SV

module freq_adapter (  
    output logic CLKOUT_P,CLKOUT_N; // differential output  
    input logic CLK_IN;             // clock input  
    input logic en_mux, [1:0] sel_mux; // register control  
    input logic [7:0] pw_adj, [1:0] sr_adj, ampl_adj;  
);
module freq_adapter import cds_rnm_pkg::*; (  
  output wreal4state CLKOUT_P,CLKOUT_N; // differential output  
  input wreal4state CLK_IN;            // clock input  
  input logic en_mux, [1:0] sel_mux;   // register control  
  input logic [7:0] pw_adj, [1:0] sr_adj, ampl_adj;  
);  

RNM uses event solver so just need to convert logic to real voltage
module freq_adapter (CLKOUT_P, CLKOUT_N, CLK_IN, en_mux, sel_mux, pw_adj, sr_adj, ampl_adj);
output CLKOUT_P, CLKOUT_N; // differential output
input CLK_IN; // clock input
input wire [2:0] en_mux, [1:0] sel_mux; // register control
input [7:0] pw_adj, [1:0] sr_adj, ampl_adj; // digital control voltage

uvm_agent (UVC)
  sequencer
driver
config
monitor

Top
  intf

MS Bridge
  analog resource (VAMS)
    proxy

DUT (VAMS)
electrical uses analog solver that takes into account VIR

electrical uses analog solver that takes into account VIR
Analog Resource for SV-RNM/VAMS

Option 1

- Automatically inserted Connect Modules (CM) converts logic signal values to SV-RNM or electrical equivalents (depending on the DUT)
  - IE card parameters used to configure the connect modules inserted (supply voltage, rise time, drive resistance, etc)
  - No changes required to UVM driver

Not recommended where control over critical analog signals needed.
Analog Resource for SV-RNM/VAMS

Option 2

- Analog resource uses proxy attributes to generate analog signal algorithmically
  - Proxy used to pass attributes that define type and shape of analog signal
  - Same agent/MS Bridge with swappable analog resource for VAMS electrical signals or SVRNM real/user-defined signals
  - Requires override of UVM driver and sequence item to change functionality from driving signals through interface to passing values through proxy

This is the option used for the demo

Recommended for continuously changing signals such as sine wave
Steps to create a UVM-MS agent

• Create Bridge module
  • Contains Analog Resource and Proxy

• Extend classes for Driver, Monitor and Sequence Item
  • Use set_type_override_by_type to use extended classes
module osc_bridge import cds_rnm_pkg::*;
    input wire osc_clk,
    output wire osc_clk_p,
    output wire osc_clk_n
);

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

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

//Selection bit to choose between single-ended clock and differential clock
parameter bit diff_sel = 0;
parameter passive = 0;

//Class proxy extends osc_bridge_proxy included in osc_pkg.sv
//The implementation for the config_wave push function is defined here
class proxy extends osc_bridge_proxy;

function new(string name = "");
    super.new(name);
endfunction : new

function void config_wave(input real ampl, bias, freq, enable);
    core.ampl_in = ampl;
    core.bias_in = bias;
    core.freq_in = freq;
    core.enable = enable;
endfunction

endclass

proxy __uvm_ms_proxy = new("__uvm_ms_proxy");
osc_driver ➔ osc_ms_driver

UVM

```verbatim
class osc_driver extends uvm_driver #(osc_transaction);
// The virtual interface used to drive and view HDL signals.
virtual interface osc_if vif;
// Count transactions sent
int num_sent;
// Period of the generated clock
real period;
// Component macro
'`uvm_component_utils_begin(osc_driver)
   '`uvm_field_int(num_sent, UVM_ALL_ON)
   '`uvm_field_real(period, UVM_ALL_ON)
   '`uvm_component_utils_end

int test_config;
// Constructor - required syntax for UVM automation and utilities
function new (string name, uvm_component parent);
   super.new(name, parent);
endfunction

virtual function void build_phase(uvm_phase phase);
   super.build_phase(phase);
endfunction

function void connect_phase(uvm_phase phase);
   if (!uvm_config_db#(virtual osc_if;):get(this,"","vif", vif))
      'uvm_error("MOVTP","virtual interface must be set for ":,get_full_name(),",vif")
endfunction
// UVM run() phase
task run_phase(uvm_phase phase);
   get_and_drive();
endtask : run_phase
```

UVM-MS

```verbatim
class osc_ms_source_driver extends osc_driver;
protected osc_bridge_proxy bridge_proxy;
osc_ms_transaction ms_req;
// Count transactions sent
int num_sent;
// Get value to drive onto diff_sel bit diff_sel;
// Provide implementations of virtual methods such as get_type_name and create
'`uvm_component_utils_begin(osc_ms_source_driver)
   '`uvm_field_int(diff_sel, UVM_ALL_ON)
   '`uvm_component_utils_end

virtual function void build_phase(uvm_phase phase);
   super.build_phase(phase);
endfunction

virtual function void build_phase(uvm_phase phase);
   super.build_phase(phase);
endfunction

protected function void build_phase(uvm_phase phase);
   if (!uvm_config_db#(osc_bridge_proxy):get(this,"","bridge_proxy", bridge_proxy))
      `uvm_error(get_type_name(),"bridge proxy not configured");
endfunction
```

// Gets transactions from the sequencer and passes them to the driver.
task osc_ms_source_driver::get_and_drive();
forever begin
   // Get new item from the sequencer
   item_port.get_next_item(req);
   // Drive the item
   drive_transaction(req);
   fork
      #100+1ns; // Time for transaction
      begin
         start = simple_thread);
         #1ns) bridge_proxy.sampling_do = 1;
         #1ns) bridge_proxy.sampling_do = 0;
      end
   join
   // Communicate items done to the sequencer
   seq_item_port.item_done();
endtask : get_and_drive
```
osc_monitor → osc_ms_monitor

UVM

class osc_monitor extends uvm_monitor;

// Virtual Interface for monitoring DUT signals
virtual interface osc_if vif;

osc_transaction osc_clk_transaction, osc_clk_p_transaction;

// This TLM port is used to connect the monitor to the scoreboard
uvm_analysis_port #(osc_transaction) item_collected_port;

function void build_phase(uvm_phase phase);
| super.build_phase(phase);
endfunction: build_phase

function void connect_phase(uvm_phase phase);
| if (!uvm_config_db#(virtual osc_if)::get(this, get_full_name(), "vif", vif))
| | uvm_error("NOVIF",""virtual interface must be set for: ",get_full_name(),",vif")
| if (!uvm_config_int::get(this,"","diff_sel", diff_sel))
| | uvm_error("NOCONFIG",""value must be set for: ",get_full_name(),",diff_sel")
else "uvm_info("CONFIG\_\_","Value of ",get_full_name(),$sformatf("",".diff Sel = \%d",diff_sel))
endfunction: connect_phase

UVM-MS

class osc_ms_source_monitor extends osc_monitor;

protected
osc_bridge_proxy bridge_proxy;

// Virtual Interface for monitoring DUT signals
virtual interface osc_if vif;

// Count transactions collected
int num_col;

virtual function void build_phase(uvm_phase phase);
| super.build_phase(phase);
| if(!uvm_config_db#(osc_bridge_proxy)::get(this,"","bridge_proxy","bridge_proxy")
| | uvm_error(get_type_name(),"bridge proxy not configured");
endfunction

task osc_ms_source_monitor::collect_transaction();
| // This monitor re-uses its data items for ALL transactions
| transaction = osc_ms_transactions::type_id::create("transaction", this);
| forever begin
| @posedge bridge_proxy.sampling_done);
| // Begin transaction recording
| void!begin_tr(transaction, "analog_clock source Monitor");
| transaction.data_type = OSC_MS\_SAMPLE;
| transaction.amp = bridge_proxy.amp_out;
| transaction.bias = bridge_proxy.bias_out;
| transaction.freq = bridge_proxy.freq_out;
| uvm_info(get_type_name(),"source transaction collected \"MS\", transaction.sprint(), UVM\_LOW); //Temporarily dropped verbosity
| if (checks_enable)
| | perform_checks();
| if (coverage_enable)
| | perform_coverage();
| // Send transaction to scoreboard via TLM write()
| item_collected_port.write(transaction);
| num_col++;
| fork
| begin : wait for sampling done
| @posedge bridge_proxy.sampling_done;
| disable wait for timeout;
end
osc_transaction → osc_ms_transaction

UVM

class osc_transaction extends uvm_sequence_item;

  rand real freq; // frequency of input clock
  bit diffSel;

  `uvm_object_utils_begin(osc_transaction)
  `uvm_field_real(freq, UVM_ALL_ON)
  `uvm_object_utils_end

  // Constraints go here
  constraint default_freq_c {
      freq > 5e8;
      freq < 1e9;
  }

  // Constructor - required syntax for UVM automation and utilities
  function new (string name = "osc_transaction");
      super.new(name);
  endfunction: new

endclass: osc_transaction

UVM-MS

class osc_ms_transaction extends osc_transaction;

  rand osc_ms_data_type_e data_type;

  // Drive fields
  rand real ampl;
  rand real bias;
  rand bit enable;

  // Measurement fields
  rand real delay; // Delay in ns
  rand int duration;

  `uvm_object_utils_begin(osc_ms_transaction)
  `uvm_field_enum(osc_ms_data_type_e, data_type, UVM_DEFAULT)
  `uvm_field_real(ampl, UVM_DEFAULT)
  `uvm_field_real(bias, UVM_DEFAULT)
  `uvm_field_int(enable, UVM_DEFAULT)
  `uvm_field_int(delay, UVM_DEFAULT)
  `uvm_field_int(duration, UVM_DEFAULT)
  `uvm_object_utils_end

  // Constraints go here
  // To override, use the same constraint name or TCL to disable
  constraint default_drive_trans_c {
      ampl > 0.95;
      ampl < 1.05;
      bias inside [ -0.05:0.5 ];
      enable dist { 1'b0 := 1 , 1'b1 := 5 };
  }

  constraint default_measurement_trans_c {
      duration > 70;
      duration < 32;
      delay > 0.0;
      delay < 1.0;
  }

endclass: osc_ms_transaction
class freq_adpt_tb extends uvm_env;

// component macro
uvm_component_utils(freq_adpt_tb);

registrs_env registers;
osc_env freq_generator;
osc_env freq_detector;

freq_adpt_scoreboard freq_adpt_sb;

// Constructor
function new (string name, uvm_component parent=NULL);
super.new(name, parent);
endfunction : new

// UVM build() phase
function void build_phase(uvm_phase phase);

// UVM-MPS
// set up bridge proxy pointer references to generator and detector UVCs
uvm_config_db #(osc_bridge_proxy)::set(this,"freq_generator.agent.*","bridge_proxy", top.generator_bridge._uvm_ms_proxy);

// override driver, monitor, and scoreboard with UVM-MPS versions
set_type_override_by_type(osc_transaction::get_type(),osc_ms_transaction::get_type());
set_type_override_by_type(osc_driver::get_type(),osc_ms_source_monitor::get_type());
set_type_override_by_type(freq_adpt_scoreboard::get_type(),freq_adpt_ms_scoreboard::get_type());

super.build_phase(phase);
endfunction : build_phase
freq_adpt_scoreboard → freq_adpt_ms_scoreboard

UVM

class freq_adpt_scoreboard extends uvm_scoreboard;

cover_e coverage_control = COV_ENABLE; // COV_ENABLE;

// component util macro
tuv_component_utils_begin(freq_adpt_scoreboard)
  uvm_component_utils_begin(freq_adpt_scoreboard)
  uvm_field_enum(cov_e, coverage_control, UVM_ALL_ON)
  uvm_component_utils_end
// define TLM port imp object
  uvm_analysis_imp_decl(_registers)
  uvm_analysis_imp_decl(_osc_gen)
  uvm_analysis_imp_decl(_osc_phy)

uvm_analysis_imp_registers #{(registers_packet, freq_adpt_scoreboard) sb_registers_in;
  uvm_analysis_imp_osc_gen #{(osc_transaction, freq_adpt_scoreboard) sb_osc_gen;
  uvm_analysis_imp_osc_phy #{(osc_transaction, freq_adpt_scoreboard) sb_osc_phy;

// write() virtual function void write_registers(registers_packet packet);
  registers_packet sb_packet;
  // Make a copy for storing in the scoreboard
  $cast(sb_packet, packet.clone()); // Clone returns uvm_object type
  reg_packets_in++;
  if(sb_packet.addr > 7)
    reg_in_drop ++;
  else begin
    if(sb_packet.addr & #1)
      reg_in_drop ++;
    end
  end
  int reg_in_req = sb_packet.addr;
  int mux_reg = INT_REG[reg_in_req];
  int sel_mux = MUX_REG[reg_in_req];
  endfunction

virtual function void write_osc_gen(osc_transaction packet);
  osc_transaction sb_packet;
  $cast(sb_packet, packet.clone()); // Clone returns uvm_object type
  if(sb_packet.addr & #1)
    write_output_regs(sb_packet, addr);
  else begin
    if(sb_packet.addr & #1)
      write_output_regs(sb_packet, addr);
    else begin
      if(sb_packet.addr & #1)
        write_output_regs(sb_packet, addr);
      else begin
        if(sb_packet.addr & #1)
          write_output_regs(sb_packet, addr);
        else begin
          if(sb_packet.addr & #1)
            write_output_regs(sb_packet, addr);
          else begin
            if(sb_packet.addr & #1)
              write_output_regs(sb_packet, addr);
            else begin
              if(sb_packet.addr & #1)
                write_output_regs(sb_packet, addr);
              else begin
                if(sb_packet.addr & #1)
                  write_output_regs(sb_packet, addr);
                else begin
                  if(sb_packet.addr & #1)
                    write_output_regs(sb_packet, addr);
                  else begin
                    if(sb_packet.addr & #1)
                      write_output_regs(sb_packet, addr);
                    else begin
                      if(sb_packet.addr & #1)
                        write_output_regs(sb_packet, addr);
                      else begin
                        if(sb_packet.addr & #1)
                          write_output_regs(sb_packet, addr);
                      end
                    end
                  end
                end
              end
            end
          end
        end
      end
    end
  end

virtual function void write_osc_phy(osc_transaction packet);
  osc_transaction sb_packet;
  $cast(sb_packet, packet.clone()); // Clone returns uvm_object type
  if(sb_packet.addr & #1)
    write_output_regs(sb_packet, addr);
  else begin
    if(sb_packet.addr & #1)
      write_output_regs(sb_packet, addr);
    else begin
      if(sb_packet.addr & #1)
        write_output_regs(sb_packet, addr);
      else begin
        if(sb_packet.addr & #1)
          write_output_regs(sb_packet, addr);
      end
    end
  end

UVM-MS

class freq_adpt_ms_scoreboard extends freq_adpt_scoreboard;

uvm_component_utils(freq_adpt_ms_scoreboard);

// write() function for registers is not required
virtual function void write_osc_gen(osc_transaction packet);
  osc_transaction sb_packet;
  $cast(sb_packet, packet.clone()); // Clone returns uvm_object type
  if(sb_packet.addr & #1)
    write_output_regs(sb_packet, addr);
  else begin
    if(sb_packet.addr & #1)
      write_output_regs(sb_packet, addr);
    else begin
      if(sb_packet.addr & #1)
        write_output_regs(sb_packet, addr);
      else begin
        if(sb_packet.addr & #1)
          write_output_regs(sb_packet, addr);
      end
    end
  end

// write() function for detector
virtual function void write_osc_phy(osc_transaction packet);
  osc_transaction sb_packet;
  $cast(sb_packet, packet.clone()); // Clone returns uvm_object type
  if(sb_packet.addr & #1)
    write_output_regs(sb_packet, addr);
  else begin
    if(sb_packet.addr & #1)
      write_output_regs(sb_packet, addr);
  end

// write() function for detector
virtual function void write_osc_det(osc_transaction packet);
  osc_transaction sb_packet;
  $cast(sb_packet, packet.clone()); // Clone returns uvm_object type
  if(sb_packet.addr & #1)
    write_output_regs(sb_packet, addr);
  else begin
    if(sb_packet.addr & #1)
      write_output_regs(sb_packet, addr);
  end

if(sb_packet.addr & #1)
  write_output_regs(sb_packet, addr);
else begin
  if(sb_packet.addr & #1)
    write_output_regs(sb_packet, addr);
  else begin
    if(sb_packet.addr & #1)
      write_output_regs(sb_packet, addr);
  end
end

How to check tests?

• Send expectations about ongoing outputs to agents monitoring meters.
  • Use Assertions to report errors when output is not within expected parameters
  • Test sends sequence item to agents as expectations change.
  • Test must be able to predict expectations.

• Reference model gets sequence items affecting the block and predicts expectations.
  • Can be sent to agents monitoring outputs. Where data can be used in assertions.
  • Sent as item to a scoreboard for check against similar item collected from DUT.
  • Scoreboard uses item compare method to determine match / no match.

• How to coordinate item generation between DUT and Reference model?
  • In digital designs the TLM model assumes that output transactions are generated as a result of input transactions.
  • Is such a transaction response the right approach in general for Analog?
Demo
UVM Messaging
Messages for Debug and Error Reporting

• Debugging activity inside a large environment with many agents 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) && !I_thr_triggered) I_thr_triggered = 1;
    else if(I_PLUS < 0.9) I_thr_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

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

osc_core.vams

`include “uvm_ms.vamsh”

`uvm_ms_info("FREQ_UPDATE",$sformatf("freq=%e Hz period=%e ns", freq_in, out_period),\ 
UVM_MEDIUM)

“uvm_ms.svh”

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

osc_bridge.sv

`include “uvm_ms.svh”

`uvm_ms_reporter // instantiates uvm_ms_reporter component to be used with messaging

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
UVM-AMS Standard Release Schedule

<table>
<thead>
<tr>
<th>Date</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/16/24</td>
<td>Freeze all content/Document Format</td>
</tr>
<tr>
<td>3/12/24</td>
<td>PR Draft</td>
</tr>
<tr>
<td>3/26/24</td>
<td>WG Approves PR Version</td>
</tr>
<tr>
<td>3/4/24</td>
<td>DVConUS</td>
</tr>
<tr>
<td>6/22/24</td>
<td>WG Approves Final Draft</td>
</tr>
<tr>
<td>6/22/24</td>
<td>BoD Approval/Release</td>
</tr>
</tbody>
</table>
Conclusions

• 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
Changes for UVM-AMS from UVM

• `uvm_*_printer print_real()` uses `%f` formatting, which truncates very small values
  • Override with `uvm_radix_real_exp`

• UVM messaging macros don’t work in modules
  • Created macros and methodology to support Analog Resource

• UVM-MS specific include/import files

<table>
<thead>
<tr>
<th>Statement</th>
<th>Usage</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>import uvm_ms_pkg::*;</td>
<td>Within the MS Bridge and uvm_ms_agent</td>
<td>Defines uvm_ms_proxy template class</td>
</tr>
<tr>
<td>`include “uvm_ms.vamsh”</td>
<td>Within analog_resource modules defined as Verilog-AMS</td>
<td>Defines UVM-MS messaging macro/functions</td>
</tr>
<tr>
<td>`include “uvm_ms.dmsh”</td>
<td>Within analog_resource modules defined as SystemVerilog</td>
<td>Defines UVM-MS messaging macro/functions</td>
</tr>
<tr>
<td>`include “uvm_ms.svh”</td>
<td>Within the MS Bridge to enable the messaging from the analog_resource</td>
<td>Requires the MS Proxy instance to be named __uvm_ms_proxy</td>
</tr>
</tbody>
</table>
Questions?