UVM Ready: Transitioning Mixed-Signal Verification Environments to Universal Verification Methodology

Arthur FREITAS

Régis SANTONJA

Outline

Intro

Pre-UVM, Module-Based Environment

UVM Environment

Introduction

- Our products' top level is an analog schematic
- Verification requires several mixed-mode top-level simulations
- We describe here how we augmented our existing analog self-checking verification framework with UVM
- UVM gives us the power to verify hard-to-imagine mode transitions, digital configurations and analog setups

Outline

Intro

Pre-UVM, Module-Based Environment

UVM Environment

PRE-UVM ENVIRONMENT

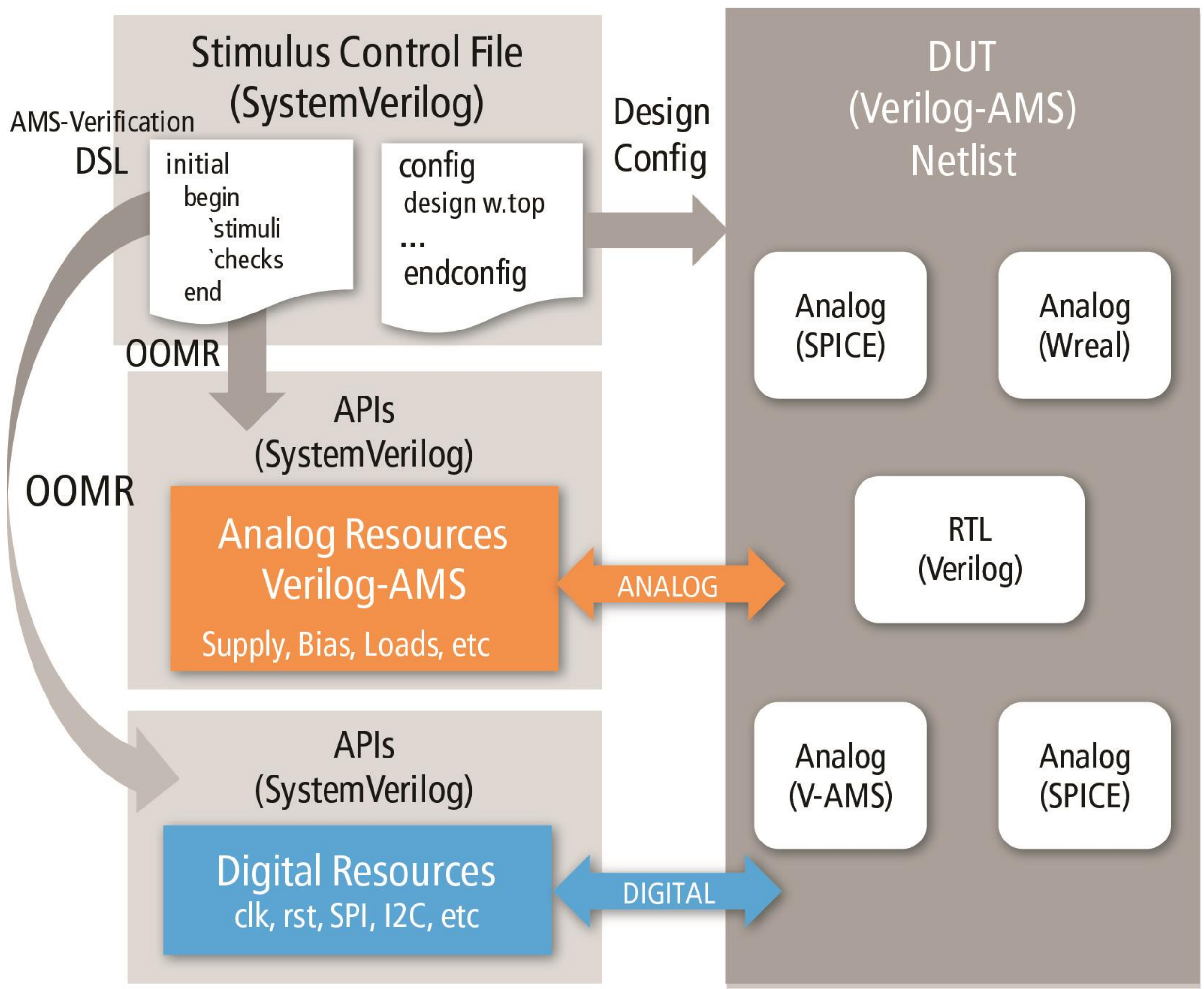
- Traditionally, most of the analog verification relied on waveform inspection
- Many analog engineers have limited knowledge of design verification languages
- Advanced verification methodologies are digital centric
- Maintaining two top-level verification environments to leverage the man power of analog designers and verification engineers is inpratical

PRE-UVM ENVIRONMENT

- Pre-uvm environment used a domain specific language (DSL) based on:
  - Pre-processor macros
  - SystemVerilog APIs
- Testbench resources controlled by OOMR from the testcase file
- Verlilog configurations define the abstraction level of the DUT
- All testcase information centralized in a single file

Pre-UVM environment

Module-Based Testbench (Verilog-AMS)

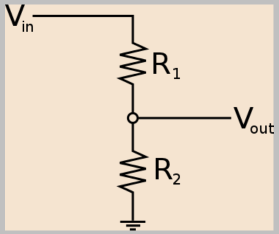


Voltage Divider Example (DUT)

File: *vdiv_ams.v*

```
/* voltage divider using 2 resistors*/
`include "constants.vams"
`include "disciplines.vams"

module  vdiv(in,out,gnd);
  input in, gnd;
  output out;
  electrical in, out,gnd;
  //voltage divider using to resistors to gnd, out is the mid-point
  Resistor   #(1K)    r1(in,out);
  Resistor   #(1K)   r2(out,gnd);
endmodule
```



File: *vdiv_wreal.vams*

```
/* voltage divider in wreal abstraction
Assumes that the output impedance of the previous
stage is low and that the input impedance of the next
stage is high */

module  vdiv(in,out,gnd);
   input in, gnd;
   output out;
    wreal in, out;
    assign out = (in – gnd) / 2.0;
endmodule
```

Voltage Divider Example (TB)

File: vbatt.vams

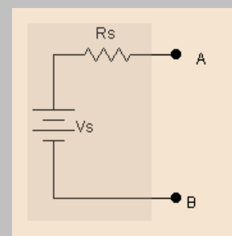
```
/* This is the v source to drive the dut */
`include "constants.vams"
`include "disciplines.vams"

module vbatt(output out);
  electrical out;
  parameter real trise = 1us;
  parameter real tfall =  500n;
  parameter real Rout  = 1m;     //low output impedance

  real vout; //controlled by analog
  real v;      //controlled by digital

  task set_vbat (input real val);
    v = val;
  endtask

  analog begin
    vout =transition(v,0,trise,tfall);
    I(out) <+ (V(out) - vout)/Rout;
  end
endmodule
```



Task to set vbatt by OOMR

Voltage Source with a series R

File: *tb.vams*

```
/* This is the testbench */
`include "constants.vams"
`include "disciplines.vams"

module tb;

  test test();       //this is our testcase instance

  wreal vout;      //coerce vout to wreal (infers a CM)
  electrical gnd;

  vbatt vbatt(w1); //v source to drive the dut

  vdiv  dut( .in(w1),
              .out(vout),
             .gnd(gnd)
              );

  analog begin
     V(gnd) <+ 0;
  end
endmodule
```

Voltage Divider Example (Testcase)

File: *test.sv*

```
/* This is the testcase file comprising the stimuli, checks, the design configuration, and sim options */
/*API macro defines. Listing the macros here just for illustration.
   Normally you put them in a separate file which is included here. */
`define V                            *1.0
`define mV                           *1e-3
`define wait_for(t)                  #(t);
`define set_vbatt(vc)                tb.vbatt.set_vbatt(vc);
`define check_v_min_max(s,mi,ma)     begin \
  if (mi <= s && s <= ma)            begin  $display( "check ok: %g < %g < %g", mi,s,ma); \
  end          else                  begin $display( "ERROR: %g < %g < %g", mi,s,ma);  errcnt++;  end end
module test;
  int errcnt;
  initial begin
         `wait_for(1ns);
        //ramp up to 12V
         `set_vbatt(12`V)
         `wait_for(2us);
        //perform analog check
         `check_v_min_max(tb.vout, 5.8`V, 6.1`V)

         //now down to 6V
         `set_vbatt(600`mV);
         `wait_for(1us);
         //perform analog check
         `check_v_min_max(tb.vout, 290`mV, 310`mV)
          `wait_for(2us);
          $display( "SIMULATION %sED !" , (errcnt == 0) ?  "PASS" : "FAIL" );
          $stop;
    end
endmodule

// please notice that you can use `defines to make configurations more readable
// for example
`define DUT_IS_WREAL instance tb.dut liblist wreallib;
`define DUT_IS_ELECT   instance tb.dut liblist amslib;
config topcfg;
  design simlib.tb;
  default liblist simlib amslib wreallib;
  //setting dut to ELECTRICAL abstraction.
  ` DUT_IS_ELECT
endconfig
//user can put here simulation options as verilog comments so that a pre-processor script can take them into account
//for example: temp=130
```

Macros defining the APIs of our analog verification language

Directed Test comprising stimulus and checks

Allows for checking internal nodes of the DUT

Design configuration in verilog syntax allowing users to choose the abstraction level of DUT blocks.

Outline

Intro

Pre-UVM, Module-Based Environment

UVM Environment

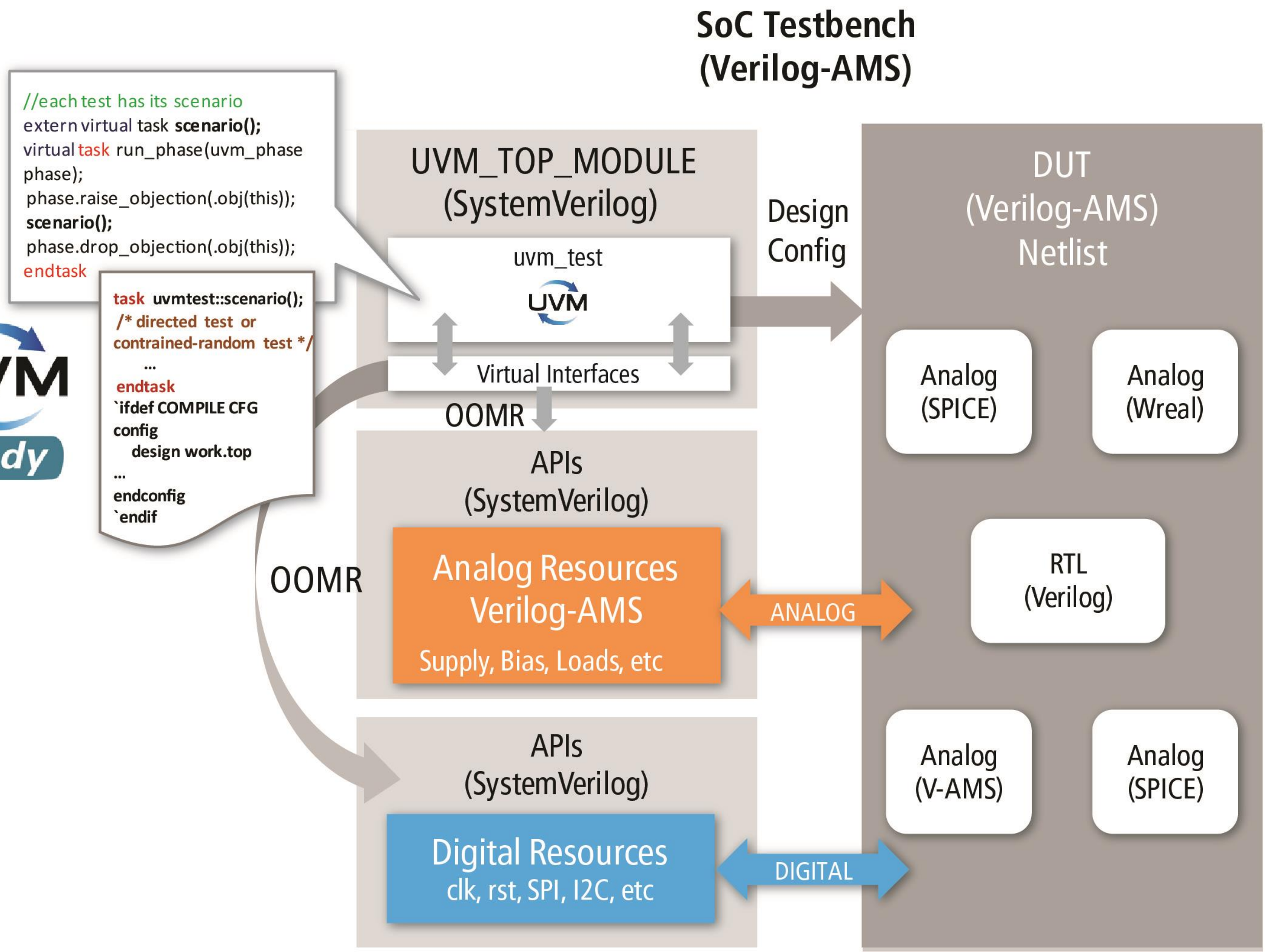
The UVM environment

- Prerequisite:
  - Backwards compatibility with well-established verification framework
  - Ability to write directed tests with the same syntax and format that our analog engineers are familiarized with
  - Re-use as much as possible the existing framework (e.g., analog mixed-signal drivers, simulation launching scripts, etc)
  - Be able to extend the framework to full-fledged UVM

The UVM environment

- Implementation embodiment:
  - Existing mixed-signal drivers shared among UVM and module based tests
  - UVM drivers outsource signal wiggling through task calls
  - Proxy systemVerilog interfaces link to existing MS drivers
  - Keep same testcase format by using extern virtual tasks

THE UVM ENVIRONMENT



UVM ready

Transaction and Sequence for our Voltage Source Example

File: vbatt_transaction.sv,

```
class vbatt_transaction extends uvm_sequence_item;

  // the voltages
  rand real vbatt;

  `uvm_object_utils_begin(vbatt_transaction)
    `uvm_field_real(vbatt,  UVM_DEFAULT)
  `uvm_object_utils_end


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


endclass
```

Radom real number to drive voltages

File: vbatt_sequence.sv

```
class vbatt_seq extends uvm_sequence
#(vbatt_transaction);
 // the  voltages
  rand real s_b;
    `uvm_object_utils(vbatt_seq)
  // Constructor
  …
  // Sequence body definition
  virtual task body();
        `uvm_do_with(req, {req.vbatt ==  s_b; }  )
  endtask
 // Constraints go here (-1V < = Vbatt <= 15V)
  constraint default_vbatt_voltage {
                              s_b >=  -1.0  && s_b <=  15.0;   }
 // pre and post body to raise and drop objections
  …
  endclass
```

Generic sequence which can be constrained by the user

UVM Driver and Proxy Interface

File: vbatt_driver.sv	File: vbatt_if.sv
<pre>class vbatt_driver extends uvm_driver #(vbatt_transaction); protected virtual interface vbatt_if vif; `uvm_component_utils(vbatt_driver) // Constructor; Build Phase … task run_phase(uvm_phase phase); super.run_phase(phase); forever begin // Get new item from the sequencer seq_item_port.get_next_item(req); // Drive the item vif.vbatt = req.vbatt; // Communicate item done to the sequencer seq_item_port.item_done(); end endtask endclass</pre>	<pre>interface vbatt_if ( ); // Import UVM package import uvm_pkg::*; `include "uvm_macros.svh"   real vbatt;  // Control events  event new_drv_values;  /* used by the monitor to detect changes and by the module TB to drive vbatt */   always @( vbatt ) begin    #1    → new_drv_values; // triggers module API     `uvm_info("IF", "Change on drive vbatt",UVM_LOW); end endinterface</pre>

Event to trigger external module-based driver

UVM Top module and Generic Test

File: uvm_top_module.sv	*File: uvm_env_test.sv*

```
module tb_uvm_top;
  import uvm_pkg::*;
  import vbatt_pkg::*;

  vbatt_if vbatt_if_i();  //interface instantiation

  always @(vbatt_if_i. new_drv_values) begin

    /*detect transactions redirect to legacy
      module-based driver */

     tb.vbatt.set_vbat(vbatt_if_i.vbatt);

  end

  initial
   begin
     uvm_config_db   #(virtual interface vbatt_if)::set(
                          null, "*.*env*", "vif",  vbatt_if_i);
    //always run the test case provided by user
     run_test("my_env_test");
   end
endmodule
```

```
`include "uvm_macros.svh"
import uvm_pkg::*;
import vbatt_pkg::*;
`include "uvm_tb.sv"

class my_env_test extends uvm_test;
  `uvm_component_utils(my_env_test)
  vbatt_tb vbatt_tb_h;
  vbatt_sequencer seqr;

  // Constructor,  Build Phase, Connect Phase
  …
 /* The actual testcase is implemented in an
    external task provided by the user via command line
*/
  extern virtual task scenario();
  virtual task run_phase(uvm_phase phase);
    phase.raise_objection(.obj(this));
    scenario();  // Actual testcase provided by the user
    phase.drop_objection(.obj(this));
  endtask
endclass
```

Virtual Interface drive event

OOMR to module-based driver

External task provided by the user

Directed Tests using UVM

- The infrastructure allows:
  - Execute the existing legacy tests
  - Create constrained-random scenarios
  - EXAMPLE:

Module based api:

```
`define set_vbatt(vc)      tb.vbatt.set_vbat(vc);
```

API translation for UVM testcase:

```
`define set_vbatt(vc)   assert(vbseq.randomize()  \
                        with {vbseq.s_b == vc; }); \
                        vbseq.start(vbseqr);
```

Voltage Divider Example (UVM Testcase)

File: *uvm_test.sv*

```
/* This is the UVM testcase file comprising the stimuli, checks and the design configuration */
`ifndef COMPILE_CONFIG
 /*API macro defines. Listing the macros here just for illustration.
   Normally you put them in a separate file which is included here. */
`define V                              *1.0
`define mV                             *1e-3
`define wait_for(t)                    #(t);
//mapping set_vbatt macro  to UVM constraint
`define set_vbatt(vc)   assert(vbseq.randomize() with {vbseq.s_b == vc; }); \
                         vbseq.start(vbseqr);

`define check_v_min_max(s,mi,ma)    begin\
  if (mi <= s && s <= ma) begin \
    $display( "check ok: %g < %g < %g", mi,s,ma);\
  end else begin \
    $display( "ERROR: %g < %g < %g", mi,s,ma);  \
   `uvm_error("VOUT_ERR", "VOUT out of range");\
  end end
task uvm_env_test::scenario();
        //initializing the UVM env
         vbatt_seq vbseq;
         vbatt_sequencer  vbseqr;
         vbseqr = vbatt_tb_h.env.agent.sequencer;
         vbseq =  vbatt_seq::type_id::create(.name("vbseq"),.contxt(get_full_name()));
         `wait_for(1ns);

        //ramp up to 12V
         `set_vbatt(12`V)
         `wait_for(2us);
        //perform analog check
         `check_v_min_max(tb.vout, 5.8`V, 6.1`V)

         //now down to 6V
         `set_vbatt(600`mV);
         `wait_for(1us);
         //perform analog check
         `check_v_min_max(tb.vout, 290`mV, 310`mV)
          `wait_for(2us);
endtask
`else // !`ifndef COMPILE_CONFIG
`define DUT_IS_WREAL instance tb.dut liblist wreallib;
`define DUT_IS_ELECT   instance tb.dut liblist amslib;
 config topcfg;
   design simlib.tb;
   default liblist simlib amslib wreallib;
   ` DUT_IS_WREAL
 endconfig
`endif
//temp=27
```

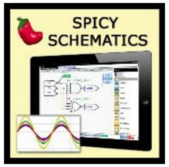
Macros defining the APIs of our analog verification language

Outsourcing error handling to the UVM environment with uvm_error macro

EXTERNAL Task called by the uvm_env_test class

Same Syntax can be used to write legacy directed tests

METHODOLOGY ACHIEVEMENTS AND ROADMAP



SystemVerilog

UVM ready

UVM

Waveform Inspections

- Good enough for small analog ICs
- Schematic-based testbench
- No regression testing
- Mostly SPICE configurations

Self-checking Analog Verification

- Adequate to mixed-signal ICs
- Programmable text-based testbench
- Regression testing
- Mixed SPICE, Wreal and Verilog-ams configurations

Mixed-Signal UVM Ready: 2013

- Backwards compatible to module-based method
- Port digital block-level UVM tests to the top-level
- UVM register layer integration
- Simple top-level randomizations
- Mixed SPICE, Wreal and Verilog-ams configurations

Mixed-Signal UVM: 2015

- Full-fledged UVM
- Result predictors, assertions and Scoreboards
- Randomization to verify mode transitions, digital configurations and analog setups
- Mainly Wreal configurations and limited use of SPICE and Verilog-ams models

Conclusion

- Successful transition of mixed-signal verification environment to UVM
- Methodology backwards compatible with traditional module-based framework
- Users can choose what environment to use
- all information required to describe mixed-signal simulation is gathered in 1 single file
- Block-level UVM can be easily ported to the top level
- Massive amount of test vectors can be produced with very little effort

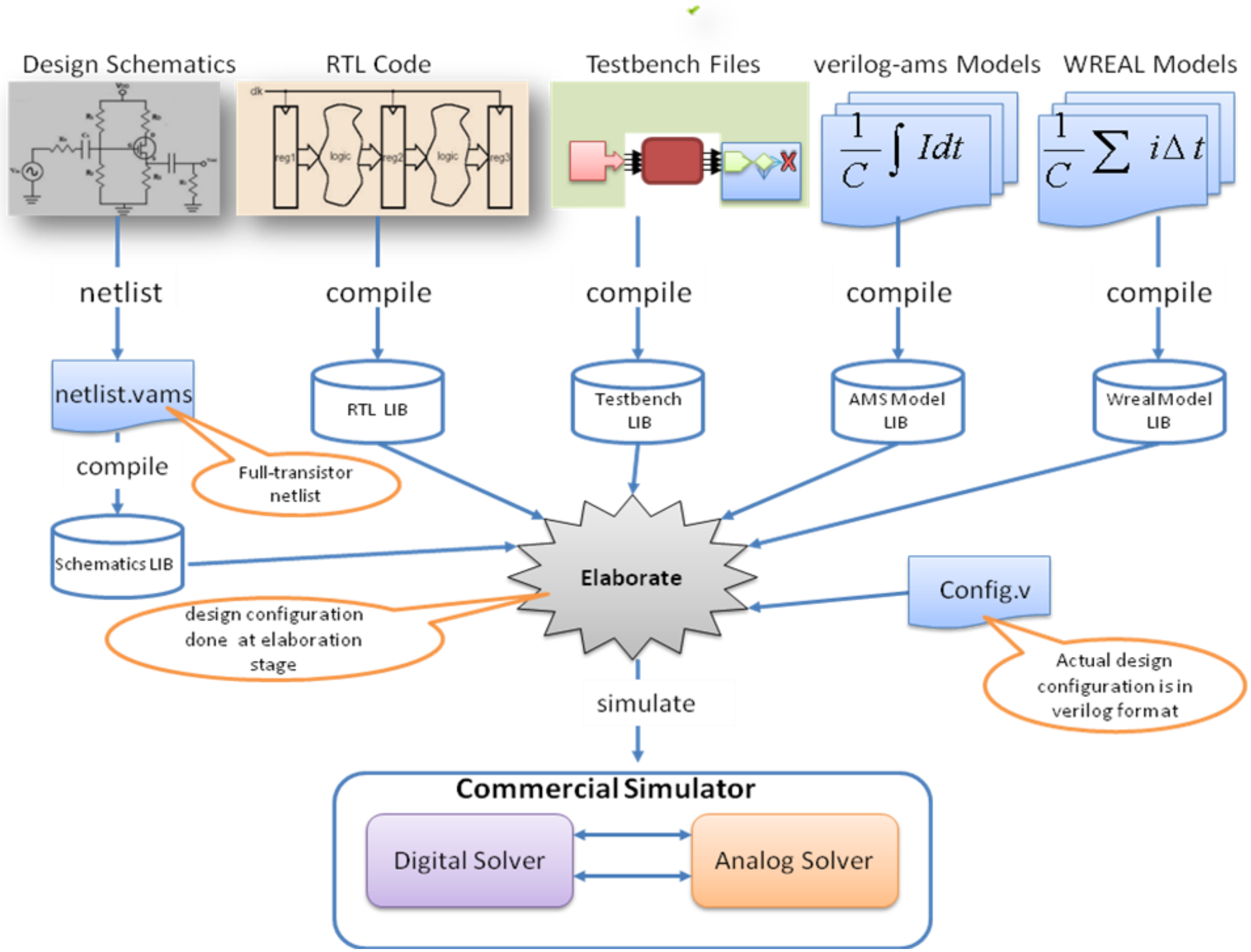
Questions

Finalize slide set with questions slide

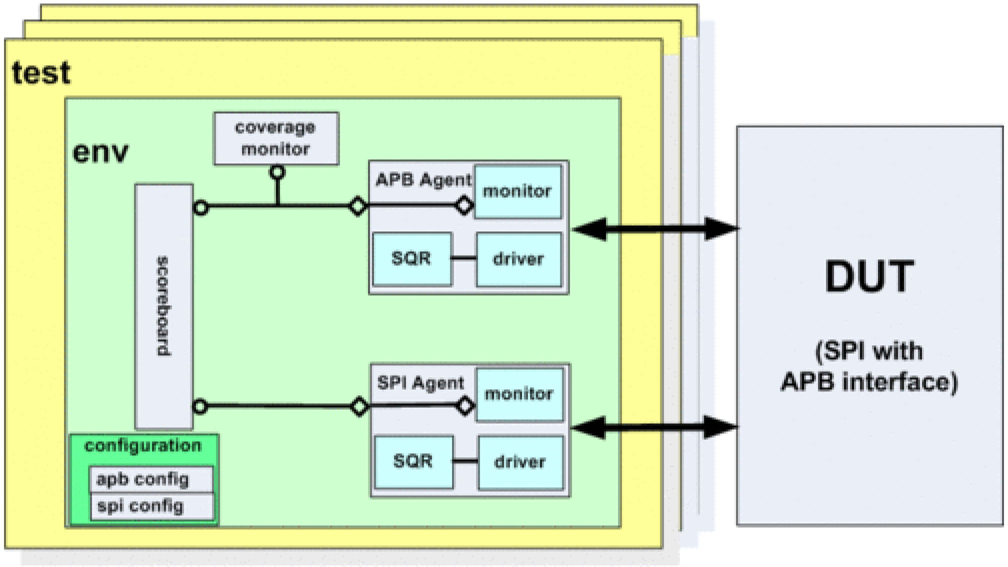
References

- [1] A. Freitas "Real-Valued Mixed-Signal Verification: An Abstraction Adjustable Methodology" CDNLive EMEA 2013, Munich.
- [2] "1800-2012 - IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language", IEEE Computer Society, USA, 2012.
- [3] "Standard Universal Verification Methodology Class Reference, Release 1.2", Accellera System Initiative, USA, 2014.

DESIGN CONFIGURATION AT ELABORATION TIME



Typical UVM Environment



Source: verificationacademy.com