2014 DESIGN AND VERIFICATION DVCON CONFERENCE AND EXHIBITION EUROPE

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

Arthur Freitas, Freescale Semiconductor, Inc., Analog & Sensors, Toulouse, France
(arthur.freitas@freescale.com)

Régis Santonja, Freescale Semiconductor, Inc., Analog & Sensors, Toulouse, France
(regis.santonja@freescale.com)

***Abstract*— This paper describes the transitioning of a module-based mixed-signal verification environment to Universal Verification Methodology (UVM). The result is a top-level mixed-signal verification framework that supports constrained-random verification and is backwards compatible to legacy module-based test cases. Thanks to a proxy systemVerilog® interface, the same mixed-signal drivers can be shared among the UVM test cases and the traditional module-based stimulus. Instead of wiggling signals, the UVM drivers use task calls to outsource bus driving to the already existing module-based drivers. This methodology not only allows the re-use of UVM block-level components at the top-level, but also the integration of the UVM register layer and all its built-in tests. Furthermore, the methodology uses verilog configurations to elect the desired abstraction level of the design under test (DUT). All information required to describe a test case (i.e., stimulus, design configuration and simulation options) is gathered in one single file, thereby significantly improving test development.**

Keywords— Mixed-Signal Verification, UVM, Self-checking Analog Simulation

I. INTRODUCTION

To verify mixed-signal integration at SoC level, our verification plans include several mixed-mode simulations in which some analog blocks are configured as SPICE models and others as behavioral models. The acceleration gained with the adoption of real-number models unleashed a series of methodology enhancements that in the past were unattainable. Among them is the introduction of UVM, the industry standard for metric-driven, constrained-random verification. Randomization provides us the power to verify the sheer number of mode transitions, digital configurations and analog setups.

This paper describes how we augmented our existing methodology with UVM while maintaining backwards compatibility to the well-established directed testing approach. We start by introducing the reader to our pre-UVM verification environment, which allowed us to perform self-checking mixed-signal verification. We then show how we extended the environment to support UVM. This method was successfully employed in the development of our next-generation battery monitoring IC.

II. PRE-UVM ENVIRONMENT

In our division, the main features of our products are analog. Despite the rapid increase of digital logic in every new project (e.g., signal processing and calibrations), our top-level remains analog.

Most analog engineers have limited knowledge of design verification languages, yet their assistance to execute top-level verification is paramount. Analog engineers are used to analog-centric simulation environments. Their testbenches are normally created using schematic entry and multiple configuration views. Traditionally, most of the analog verification relied on waveform inspection. By contrast, the advanced verification methodologies employed by verification engineers are digital centric. That is, they are command-line driven and employ object-oriented languages such as systemVerilog. Maintaining two top-level verification environments to leverage the man power of both teams in top-level verification is impractical.

In 2012, we introduced a digital-centric verification environment that enabled our analog engineers to perform self-checking top-level simulations. It comprised a domain specific language (DSL) based on pre-processor macros and systemVerilog APIs. Digital and analog resources were controlled by out-of-module reference (OOMR) from a testcase file. Verilog configurations allowed the user to select the DUT abstraction required for each testcase. All this information was centralized in a single file, which meant that each testcase was a systemVerilog file passed as a parameter to a simulation launching script.

Our verification environment used a single full-transistor netlist for all simulations. Once the netlist was created, all other simulation related tasks were performed outside the analog development environment using text-based files. The same pre-compiled netlist was then shared among all test-cases. We compiled different design components in separate simulation libraries and we used configuration written in Verilog to instruct the simulator which model to use for each instance in the design. Please notice that the design configuration is not done at the netlisting stage (as in a conventional flow), but at the elaboration stage, thereby making the verification environment completely independent of the analog development environment. Our verification environment was first introduced in [1].

Figure1 depicts the verification environment in a typical mixed-signal design (with an analog top-level). The module-based analog and digital resources (e.g., drivers and monitors) are controlled by a systemVerilog control file via OOMR. The same systemVerilog file also contains the design configuration in verilog format (more info in [2]). Simulation options are entered as comments. Each mixed-signal testcase is self-contained in a single file, thus there are less files to maintain and revision control (i.e., less check-ins, check-outs).

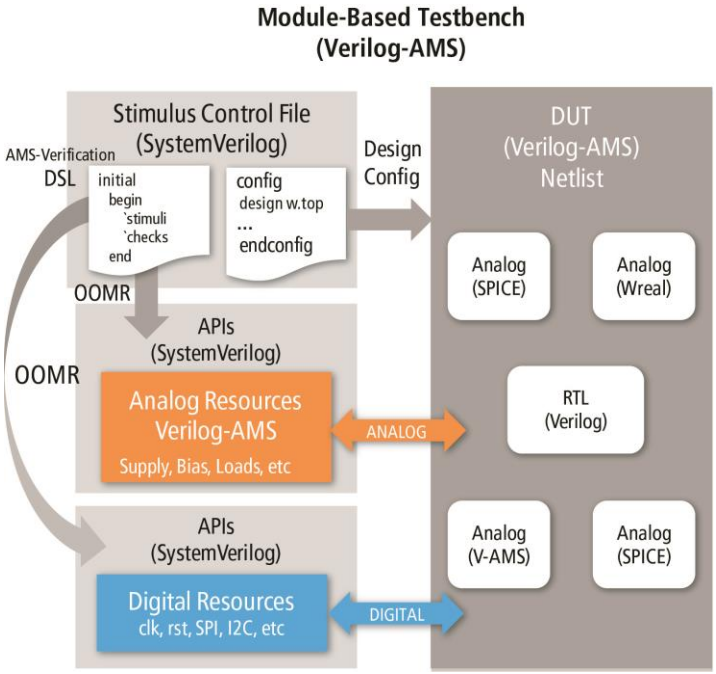


Figure1 – Pre-UVM Verification Environment

To illustrate this methodology consider a simple voltage divider and its wreal model representation.

Table 1 – Example Voltage Divider Design

File: vdiv_ams.v	**File: *vdiv_wreal.vams***
/* 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**	***/* 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**

Table 2 depicts a voltage driver (i.e., *vbatt*) and a testbench to verify our voltage divider.

Table 2 - Voltage Driver and Testbench

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
```

Table 3 illustrates a testcase file. By defining simple pre-processor macros, we created a verification DSL, which is used by analog engineers to verify the design. For example, ***set_vbatt*** is a verilog macro that sets our voltage source *vbatt* via OOMR. The multi-line verilog macro ***check_v_min_max*** provides an easy way to perform voltage checks. Please notice that stimulus, checks, and the design configuration are comprised in the same file. In the testcase described below, we drive different voltages to the dut's input, wait for a settling time and check that the output voltage is within the expected levels.

Table 3 - Testcase file containing the directed test and the design configuration in verilog format

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_vbat(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
```

Macros defining the APIs of our mixed-signal verification language

Directed Test comprising stimulus and checks

```
        `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
```

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

Each testcase scenario can be described in a single file that is used as argument for the simulation launching script. For example, to run the testcase described in Table 3 the user types the following command on a UNIX terminal:

./runsim.py --gui -t ./test.sv

Simulations options (such as temperature, Cmin, vabstol, etc) can be specified as verilog comments which are parsed by the simulation launching script.

III. THE UVM ENVIRONMENT

Our mixed-signal verification methodology can co-exist with UVM as long as the UVM drivers use the same systemVerilog APIs. To accomplish that, the virtual interfaces referenced in the UVM drivers calls the existing module based APIs. The UVM drivers simply forward the transactions to the virtual interface. The actual signal wiggling is performed by legacy module-based drivers. To make the system backwards compatible, we created a scenario file for each test (that way the same launching script could be employed). To do so the UVM top module always runs the same *uvm_test* calling an extern virtual task. The task is the actual test scenario, and is implemented in a separate file. The power of UVM is now at the user's fingertips but the same pre-processor APIs can still be used to generate directed tests.

Figure 2 depicts our UVM environment. The UVM top module allows users to write tests either in the old directed-test syntax, (which are transparently mapped into UVM constraints) or in conventional UVM syntax (e.g., `uvm_do_with macros). If the UVM top module is not instantiated, our testbench is still functional and supports the legacy module-based testcases.

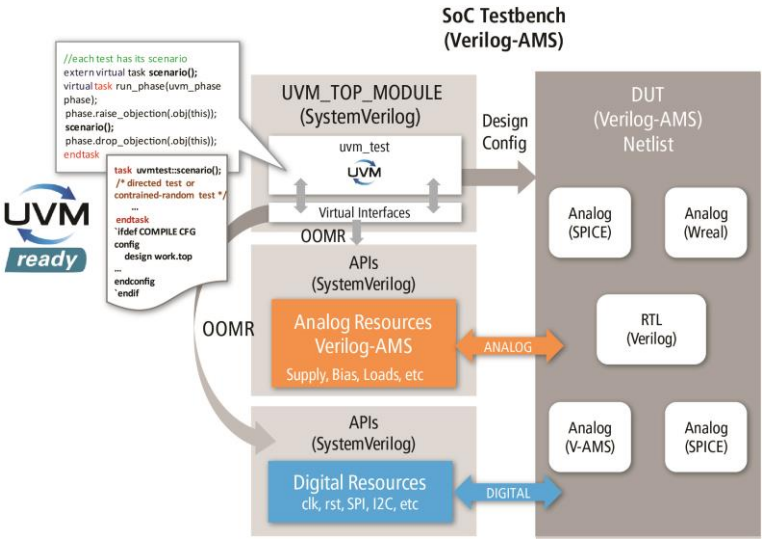


Figure2 – UVM Verification Environment

We now show this with our simple voltage divider example. We can define a UVM transaction and a generic sequence to command the existing Vbatt driver.

Table 4 - Generic transaction and sequence for vbatt driver

File: vbatt_transaction.sv,	*File: vbatt_sequence.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	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

Radom real number to drive voltages

Generic sequence which can be constrained by the user

The following table shows how the UVM driver and virtual interface looks like. The driver gets items from the sequencer and forwards them to the virtual interface. Please notice that the virtual interface does not have any ports and it is not really connected to the actual design. All it does is generate an event to instruct the testbench that vbatt must be updated. The legacy module based drivers take care of driving the design (i.e., pin wiggling).

Table 5 - Vbatt driver and interface

File: vbatt_driver.sv	*File: vbatt_if.sv*
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	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

Event to trigger external module-based driver

The UVM top module which is instantiated in the legacy testbench is depicted in Table 6. As new transactions are generated, events are created by the interface which in turn triggers *vbatt* updates to the legacy analog driver (using OOMR). Please notice that the environment always runs the same *uvm_test* depicted in *my_env_test.sv*. The testcase is actually implemented in an extern virtual task (named scenario) provided by the user.

Table 6 - UVM top module and test

File: uvm_top_module.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_vbatt(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
```

Virtual Interface drive event

OOMR to module-based driver

File: my_env_test.sv

```
`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
```

External task provided by the user

With the above infrastructure we have everything that is needed to execute existing legacy tests (using the existing module-based drivers) as well as create new directed tests using the traditional methodology. All we have to do is to constraint the UVM transactions generated with the generic sequence (*vbatt_sequence.sv*, Table 4) and map them to the legacy API macros. In our module-based example, the *set_vbatt* macro was defined as:

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

To use UVM, we re-map `set_vbatt to a multi-line macro. Please notice that *vbseq* is the generic *vbatt* sequence that we created in *vbatt_sequence.sv* (Table 4):

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

The check macros must now make reference to the *`uvm_error* macro. Table 7 shows how directed tests can be developed in the UVM framework. The syntax for creating directed tests is exactly the same as the one employed in the pre-UVM method.

Table 7 - UVM-based testcase file

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; }); .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
```

Macros defining the APIs of our Mixed-Signal Verification DSL

Outsourcing error handling to the UVM environment with uvm_error macro

```
task my_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
//user can put here simulation options as verilog comments so that a pre-processor script can take them into account
//for example: temp=130
```

EXTERNAL Task called by the my_env_test class

Same Syntax can be used to write legacy directed tests

The test described above can be launched with the same simulation launching script, just like our legacy tests.

./runsim.py - -uvm --gui -t ./uvm_test.sv

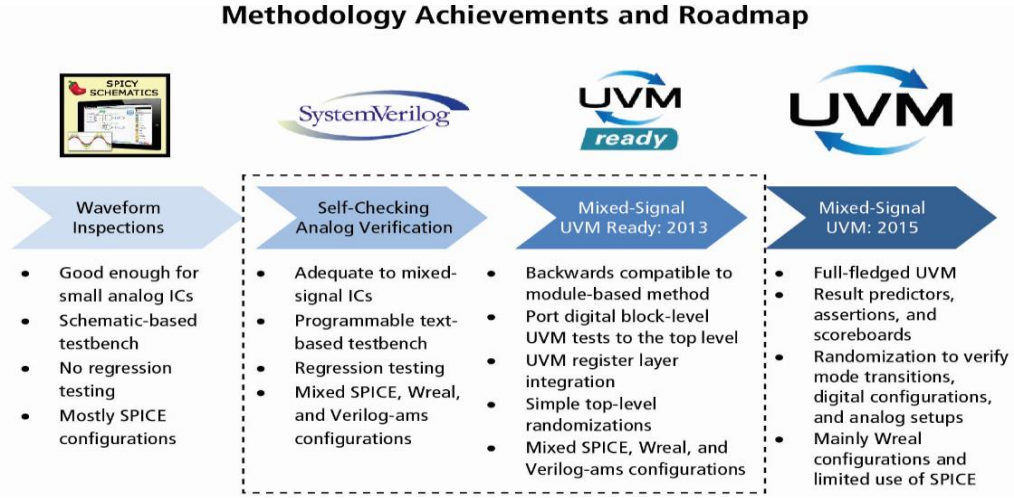
Like in the previous methodology, simulation options (e.g, vabstol, iabstol, temp, etc) can be entered as verilog comments which are parsed and interpreted by the simulation launching script. Please notice that the pass/fail judgment is now outsourced to the UVM library. Moreover, the verilog configurations must be conditionally compiled to prevent syntax errors.

In the current example the UVM transaction is very simple (i.e., a single voltage value), yet the same principle can be applied to complex transactions. For example, our IC had a SPI interface to write registers. The UVM driver did not handle the SPI pins (i.e., sclk, cs, miso, mosi), only the actual high-level transaction (i.e., address, data, rw). UVM transactions were simply forwarded to the legacy module-based drivers to toggle sclk, cs, and mosi.

One prerequisite of the new UVM environment was the ability to write directed tests using the same syntax that our analog engineers were familiarized with. However, this was not the main objective. By developing monitors and scoreboards, we can now augment the traditional directed-test methodology with a constrained-random one. With the new environment, advanced users can unleash the power of UVM to uncover hard-to-find bugs. The ***my_env_test::scenario*** task (Table 7) supports full-fledged UVM constructs to perform advanced verification.

IV. METHODOLOGY ACHIEVEMENTS AND ROADMAP

Figure 3 shows the evolution of our top-level mixed-signal methodologies and the roadmap for the adoption of full-fledged UVM. The sections within the dashed lines are the ones we covered in this paper.



The self-checking environment brought important benefits such as regression testing which in the past was not possible. The methodology was rapidly adopted by our analog designers which were familiarized with the directed-test practice. Our current "UVM-ready" method provides a gradual transition to the new verification paradigm.

V. CONCLUSION

We have successfully integrated UVM in the design verification of our next-generation battery monitoring IC. Analog designers accustomed with the traditional verification environment could continue to develop directed test using mixed-signal configurations in SPICE / Behavior Verilog-AMS abstraction. Design verification engineers could complement top-level verification with UVM-powered constrained random stimulus using real-number-model configurations. With UVM, a massive amount of test vectors can be produced with very little effort. Please notice that, in principle, we could use UVM to verify the design in SPICE abstraction, however runtimes become prohibitive.

The methodology is backwards compatible with the traditional module-based framework. Users who prefer to use UVM can do so by using the UVM testcase template and providing a command line option. In UVM mode, the UVM top level is instantiated in the module-based testbench and re-uses its existing infrastructure. Users who chose to continue to use the traditional simulation framework do not experience any difference.

Block-level tests developed in UVM by the digital group can be easily ported to the top-level. In the past this was not possible because the top-level verification environment did not support UVM. Moreover, the UVM Register Layer can now be integrated at the IC top-level environment saving us considerable development time to verify register sequences.

Finally, all information required to describe a mixed-signal simulation is gathered in one single file –a stark contrast to traditional flows, in which up to 4 views are required to describe stimulus, design configuration, and simulation options.

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.