A UVM SystemVerilog Testbench for Analog/Mixed-Signal Verification: A Digitally-Programmable Analog Filter Example

Charles Dančak [1]
Betasoft Consulting, Inc.
charles@betasoft.org

***Abstract*-A simple way of extending the UVM framework to verify an analog/mixed-signal device-under-test (DUT) is presented. A digitally-programmable analog bandpass filter circuit serves as an example. The SystemVerilog UVM testbench presented checks the filter's transfer gain at randomly-chosen frequencies against the values predicted by SPICE. It collects the results in a UVM-based scoreboard. It uses SystemVerilog assertions to check the supply current and bias voltage levels during power-down mode. The test suite can adjust the number of transactions until the desired coverage is met. The proposed testbench uses standard UVM components as-is by encapsulating all the analog-specific contents into a fixture submodule. It contains the DUT itself, generates analog stimuli, and measures analog amplitude using XMODEL primitives. By seamlessly integrating XMODEL's ability to run fast and accurate analog simulations into this UVM testbench, an efficient SystemVerilog-based verification with SPICE-level accuracy is demonstrated.**

I. INTRODUCTION

Today's system-on-chip (SoC) designs have to interact in a variety of ways with the analog world around us. This requires more analog/mixed-signal (AMS) circuits on a silicon die. Too often, subtle bugs in the AMS blocks, or in the surrounding digital-interface logic, can lead to costly chip redesigns. Around the industry, much effort has been invested in utilizing Universal Verification Methodology (UVM) techniques to verify the on-chip AMS functionality.

Various strategies have employed a UVM-SystemC library with custom AMS extensions [1] for automotive chips, a SystemVerilog real-number model (RNM) of a flash analog/digital converter [2], and UVM testbenches employing a mixture of SPICE, Verilog-AMS and **`wreal`** modeling for analog [3]. In addition, a UVM-based analog verification standard is under development by Accellera's UVM-AMS Working Group [4]. Its goal is to extend all the familiar enhancements of digital UVM testbenches—modular components, functional coverage metrics, concurrent assertions —to the AMS domain. This standard will include both analog and digital UVM agents. In turn, the analog agent will interact with *analog resources* able to drive or sample the analog signals of interest that enter or leave the SoC.

This work presents a UVM-based testbench to verify a digitally-programmable audio bandpass filter. An extension of a previous object-oriented programming (OOP) testbench [5,6], this UVM-compliant testbench reflects similar goals as the Accellera standard, and is built along the same architectural lines. One difference is that our approach requires neither co-simulation facilities nor user-defined RNM modeling, while still achieving SPICE-like accuracy.

A. High-Level Testbench View

A high-level view of the UVM-based testbench for our analog/mixed-signal device-under-test (DUT) appears in Fig. 1. Instead of the SystemVerilog application-programming interface (API) routines proposed in the Accellera standard, we rely on Scientific Analog's XMODEL [7] software to enable UVM macros and methods—all written in SystemVerilog—to interact with the DUT's analog inputs and outputs, verifying its operation across various modes.

XMODEL is a plug-in logic-simulator extension that represents an analog waveform using functional expressions, expressed in time domain and in Laplacian **s**-domain format. An event will occur only when a functional expression changes over time—not with every change in the signal's value. This can enhance simulation speed significantly [8].

1 C. Dančak is also a SystemVerilog instructor with UC San Diego Extension, La Jolla CA 92093, USA.

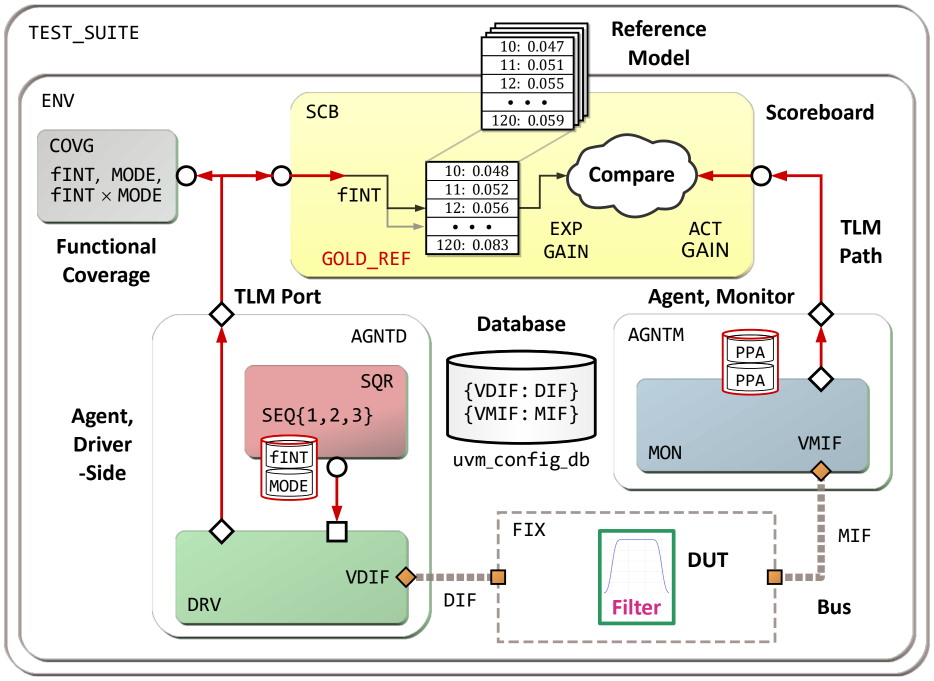


Figure 1. Upper Layers of UVM Testbench for Audio Filter

In this work, we show how an XMODEL simulation of the bandpass filter yields high speed, yet is accurate within 2% of HSPICE. We demonstrate seamless integration with familiar UVM capabilities like constrained randomization, functional coverage, transactional-level modeling (TLM) pathways, and concurrent assertion of analog properties.

In Fig. 1, the DUT is instantiated within a fixture submodule named **FIX**. It is connected to the driver and monitor objects in the usual UVM manner, via interface buses **DIF** and **MIF**. The **uvm_config_db** associates the actual buses (e.g. **DIF**) with their virtual counterparts (e.g. **VDIF**). In this figure, the buses and the fixture are indicated by dashed lines, since they are actually instantiated within the topmost UVM module, described in more detail in Section V.

B. *Two Agents*

The architecture includes a separate agent to handle each interface with the DUT. The driver-side agent, **AGNTD**, is involved in sending a sequence of data packets from sequencer **SQR** to the driver **DRV**. Each packet has stimulus data to be driven into the DUT. This includes a random integer frequency **fINT** constrained to the filter's audio range of 10–120 kHz, and one of eight randomly-chosen passband **MODE** values. Every packet is assigned an identifying tag.

SystemVerilog code for the data packets, the cyclic randomization of **MODE**, and conversion of **fINT** to a random real frequency **fREAL** (necessary in the Synopsys **VCS** simulator we used) is found in our previous articles [5,6]. In this article we focus on the changes needed to migrate from an OOP -based to a fully UVM-compliant testbench. We sought to keep the UVM hierarchy simple. Thus, we have omitted the usual monitor object in the driver-side agent.

Monitor-side agent **AGNTM** is responsible for sending the DUT's response to the scoreboard **SCB**. The same packet type is used, but the focus is on the measured peak-to-peak voltage amplitudes (**PPA_OUT**, **PPA_IN**) seen at the DUT analog input and output. Monitor **MON** samples data arriving on the **MIF** bus, then reassembles it into packet form.

Stimulus and response data are forwarded by the driver- and monitor-side agents up to the scoreboard along TLM pathways, represented by the red arrows in Fig. 1. Intuitively, each arrow corresponds to a **.connect()** call. Most of the pathways in the figure connect the packet source to a *single* sink. But one TLM path, along the left side of Fig. 1, can *broadcast* its packets to multiple sinks: in this case the scoreboard **SCB** and coverage object **COVG**. This path is marked by small white diamond icons representing TLM *analysis* ports [9]. Object **COVG** is discussed in Section VIII.

C. *Scoreboard*

The scoreboard is the most DUT-specific object in the testbench. It compares actual versus expected filter gain, at the applied frequency and mode. Actual gain is **PPA_OUT/PPA_IN**. Expected gain is found through a table look-up.

First, the scoreboard selects one of a set of eight mode-specific tables. Each table was generated by performing an HSPICE linear AC analysis, then reformatting the results into a SystemVerilog associative array. The scoreboard can scan the selected table to find the expected **VOUT** amplitude closest to **fINT**. The expected gain is then **VOUT/VIN**.

In the next section, we delve into the details of the fixture submodule. There, we will focus on how an analog DUT can be linked into a UVM testbench. Our goal is to keep the UVM components of the testbench as simple as possible.

II. Inside the Fixture Submodule

A Cadence ***Virtuoso***-generated schematic view of the fixture appears in Fig. 2. Submodule **FIX** has interface port **FREQ_IN** connecting to **DIF**, and **AMPL_OUT** connecting to **MIF**. This submodule contains the DUT itself, and various XMODEL primitives [10] used to apply analog stimuli to the filter and measure its analog response. These primitives play the same role in our testbench as the abstract analog resources envisioned in Accellera's UVM-AMS standard.

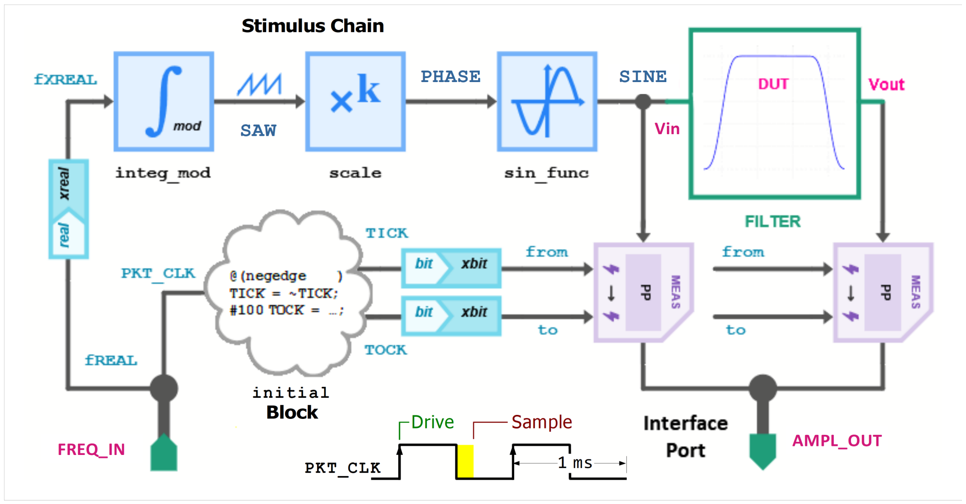


Figure 2. A ***Virtuoso*** Schematic of Fixture Submodule

A. Bandpass Filter

The transistor-level DUT is detailed in Fig. 3. It is an active *RC* audio bandpass filter with a two-stage op-amp. Its components are all XMODEL analog and digital primitives, using device models from a CMOS 45-nm SPICE library. Its digital control-logic can switch various transmission gates on or off, to vary the filter's passband and gain.

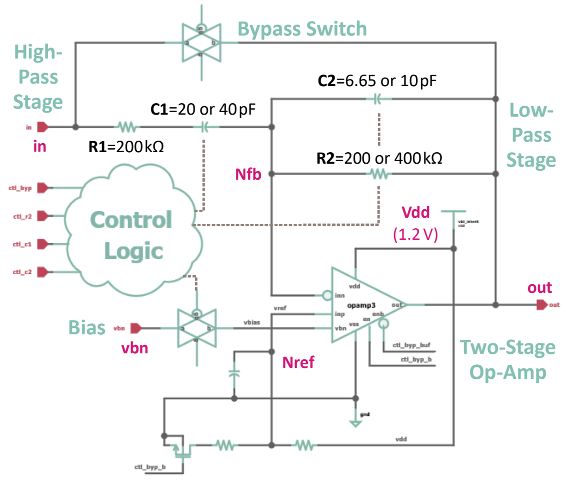


Figure 3. Automotive Bandpass Filter DUT

The most-significant digital control bit puts the DUT in bypass/power-down mode. Three lower-order control bits select one of eight programmable filter modes, as listed in Fig. 5 below. Our straightforward testing plan is to apply a sequence of stimuli to the DUT. Each transaction will consist of a random mode and a sinusoidal input of random frequency. Actual filter gain can then be compared against expected filter gain, in any mode, by the scoreboard. We then power down the DUT for some number of cycles, recover, and finally resume testing for several more cycles.

B. Stimulus Chain

A stimulus-generation chain along the upper half of Fig. 2 transforms an incoming randomized **fREAL** value into a sinusoidal input at that frequency, ready to apply to the DUT port **in**. By integrating **fREAL** over time t (modulo 1), we create a ramp signal $f\,t$. Taken modulo-1, the result is a periodic sawtooth of frequency f. When scaled by 2π, it yields a phase argument ωt. Feeding this argument into sine function **sin_func**, with an amplitude **VIN** of 100 mV, we obtain the desired sinusoidal filter input. This signal is designated **SINE** in the fixture code listed in Fig. 4 below.

This circuit can either be captured in ***Virtuoso*** using the XMODEL library, or written in SystemVerilog as shown.

A few *conversion* primitives (narrow rectangles) appear in Fig. 2. A **real_to_xreal** converter is required at left to change the incoming **real** frequency value into the corresponding **xreal** type required by XMODEL primitives. The other converters in Fig. 2 change an ordinary **bit** signal into **xbit** type. The converters enable SystemVerilog code—shown as a cloud in Fig. 2—to interact almost seamlessly with these XMODEL primitives and data types.

Workhorse user-defined type **xreal** is a SystemVerilog **struct**, specified in included header file **xmodel.svh**. It has a value field, as well as other fields XMODEL uses to simulate analog signals in an event-driven fashion. Unlike the user-defined **struct** types often employed in RNM [11], we need no knowledge of its internal fields, but simply substitute **xreal** type for ordinary SystemVerilog types **real** or **shortreal** wherever needed in our fixture circuit.

Similarly, the workhorse **xbit** type is also a **struct**; its fields support high-precision simulation of digital edges and pulse widths. In Fig. 2, this precision enables us to derive two accurate trigger signals for timing measurements.

Each converter has an inverse. Thus, **xreal_to_real** can be instantiated to convert back to ordinary **real** type.

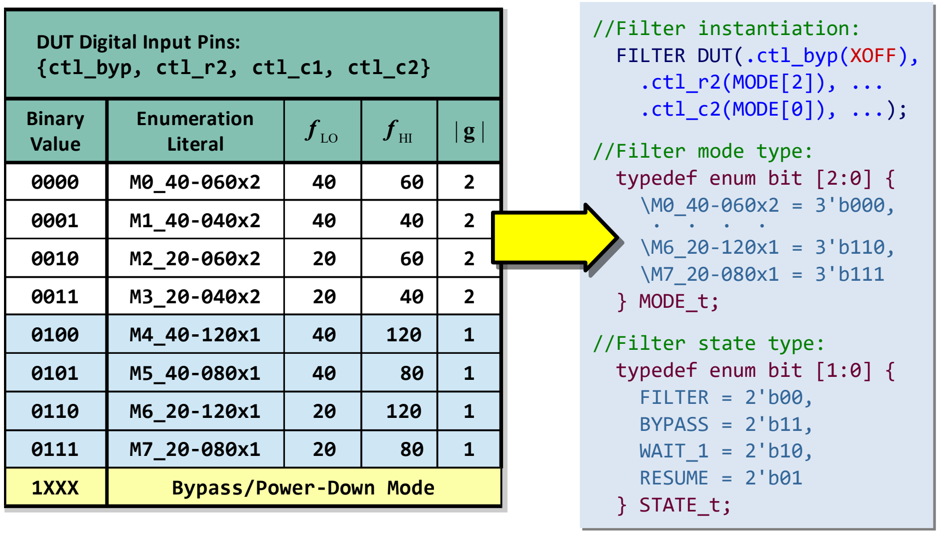
```
//Fixture signals, using XMODEL data type:
  xreal fXREAL, SAW, PHASE, SINE;
  real_to_xreal R2XR(.in(FREQ_IN.fREAL), .out(fXREAL));

//Sawtooth of period 1/f:
  integ_mod
    #(.scale(1), .modulus(1), .init_value(0) )
    F_SAW(.in(fXREAL), .out(SAW));

//Convert f to ω using math package:
  scale #(.scale(2 * M_PI))
    F2W(.in(SAW), .out(PHASE));

//Sawtooth ramp to sine cycle:
  sin_func
    #(.mode("sin"), .scale(VIN))
    F_SIN(.in(PHASE), .out(SINE));
```

Figure 4. Fixture Code to Generate sin(ωt)

As the clock waveform at the bottom of Fig. 2 indicates, stimulus is driven into the DUT at the very beginning of the 1-ms **PKT_CLK** cycle. Response is sampled later, around mid-cycle, allowing ample time for transients to settle.

No conversion is needed to apply the *digital* stimulus to the programmable filter. The four-bit control word, of the form **{XOFF, MODE}**, is wired directly to the filter's digital pins, as indicated by the DUT instantiation code in Fig. 5. The **XOFF** bit is handled separately from **MODE**. Never randomized, it will be *explicitly* asserted in the **BYPASS** state, to power down the DUT. It is deasserted in states **FILTER**, **WAIT_1**, and **RESUME** to restore normal filtering action.

Escaped naming is used to enhance the readability of the mode names and their passbands, and to further illustrate seamless integration between XMODEL and SystemVerilog syntax. The four enumerated states defined at the lower right of Fig. 5 are used to print status information during **run_phase** and **report_phase**—a convenient debug aid. In the next section, we show how these states are explicitly assigned during the sequencing of a test.



DUT Digital Input Pins:
{ctl_byp, ctl_r2, ctl_c1, ctl_c2}

Binary Value	Enumeration Literal	f_{LO}	f_{HI}	$\|g\|$
0000	M0_40-060x2	40	60	2
0001	M1_40-040x2	40	40	2
0010	M2_20-060x2	20	60	2
0011	M3_20-040x2	20	40	2
0100	M4_40-120x1	40	120	1
0101	M5_40-080x1	40	80	1
0110	M6_20-120x1	20	120	1
0111	M7_20-080x1	20	80	1
1XXX	Bypass/Power-Down Mode			

```
//Filter instantiation:
  FILTER DUT(.ctl_byp(XOFF),
    .ctl_r2(MODE[2]), ...
    .ctl_c2(MODE[0]), ...);
//Filter mode type:
  typedef enum bit [2:0] {
    \M0_40-060x2 = 3'b000,
     .  .  .  .
    \M6_20-120x1 = 3'b110,
    \M7_20-080x1 = 3'b111
  } MODE_t;
//Filter state type:
  typedef enum bit [1:0] {
    FILTER = 2'b00,
    BYPASS = 2'b11,
    WAIT_1 = 2'b10,
    RESUME = 2'b01
  } STATE_t;
```

Figure 5. Programmable Filter Modes and States

C. *Measurement Chain*

The middle portion of Fig. 2 shows the measurement chain that samples peak-to-peak voltage amplitudes at **`in`** and **`out`**. Measuring a peak-to-peak voltage swing requires a well-defined time span of at least one cycle. This time span is delimited by **`from`** and **`to`** trigger signals of type **`xbit`**. They are derived from an ordinary SystemVerilog **`initial`** block. It drives **`bit`** signals named **`TICK`** and **`TOCK`**. Signal **`TICK`** transitions on falling edges of the clock; **`TOCK`** transitions shortly after. Together, they define the yellow-highlighted measurement time span visible in Fig. 2.

Each edge is an active trigger. The same signals are fed to both **`meas_pp`** primitives. As a result, the monitor will sample the amplitude data on **`MIF`** a little after mid-cycle. This leaves a sufficient time span to measure amplitudes.

The fixture thus encapsulates analog stimulus and measurement primitives, and their **`xreal`** or **`xbit`** connections, within one convenient submodule. Its role is similar to the *harness* that has been proposed in the Accellera standard.

As we describe in Section VII, the fixture also includes primitives to support the assertion of analog properties: leakage-current level during power-down mode, and the subsequent recovery of bias voltage. Another practical use for the fixture—omitted from Fig. 2 to reduce clutter—is to feed through test-management signals from the driver's side to the monitor's side. These include integer packet tags, trigger signals like **`TOCK`**, and the current state name.

III. A "FLAT" TEST SEQUENCE

The last section described the architecture of a fixture circuit able to exercise the programmable filter in any of its modes or states. In this section, we devise a detailed test suite to apply to the DUT. Our test plan has several phases:

- Operate the DUT in normal filtering mode for an adjustable number of cycles (**`TRIALS`**). Collect coverage metrics.
- Power down and bypass the DUT for an adjustable number of cycles (**`INACTV`**). This includes one **`WAIT_1`** cycle for restoring power. Collect no coverage metrics during this inactive period, but rely on assertions to pass or fail.
- Resume normal filtering operation for an arbitrary fixed number of cycles (4) to test recovery from power-down, and to collect additional coverage of applied frequencies and modes.

In UVM methodology, the simplest way to implement such a test plan is to define three distinct *sequences,* which are then run consecutively. This strategy is known as a *flat* (i.e., neither hierarchical, layered, nor parallel) approach [12]. The starting point is the sequencer **`SQR`**, an object that sends packets to the driver in Fig. 1.

Our simple flat-sequence approach requires only a minimal sequencer object. We instantiate the base-class UVM sequencer class *as-is*, inside of the driver-side **`AGNTD`**. We can then parameterize it to our specific **`PACKET`** type:

```
//Send sequence items of type PACKET to driver:
  uvm_sequencer  #(PACKET) SQR;
```

Object **SQR** is connected to the driver by a TLM path, the short downward red arrow in Fig. 1. A UVM sequence is an *algorithmic* packet generator. Its algorithm is defined by a **body()** task. Fig. 6 shows the task code for **SEQ1**. It is left *untimed*. It outputs a series of packets to apply random stimuli to the DUT during normal filtering operation:

```
task body();   //SEQ_FILTER: SEQ1
  TX_PKT = PACKET::type_id::create("TX_PKT");
//Set invariant fields:
  TX_PKT.XOFF  = 1'b0;     //Deassert power-down bit.
  TX_PKT.STATE = FILTER;   //Normal DUT state, FILTER.
  for (int I=1; I <= TRIALS; I++)   //Knob from command line.
  begin:LOOP
    start_item(TX_PKT);    //Called by DRV at rising clock edges.
    ++TX_PKT.TAG;          //Driver and monitor tags must match.
    TX_PKT.randomize();    //Randomize frequency and mode.
    finish_item(TX_PKT);
  end:  LOOP
endtask: body
```

Figure 6. Task Defining a Sequence for Normal Filtering

Once the task creates a driver-side packet **TX_PKT**, it explicitly deasserts power-down bit **XOFF**, and sets the state field to **FILTER**. These two fields do not vary from packet to packet. Then we iterate for some number of **TRIALS**. In each pass, the frequency and mode will be randomized. The packet's tag is incremented. A handshake is carried on with the driver object behind the scenes, by calling the pair of UVM methods **start_item** and **finish_item()**.

This sequence is completely untimed. It is left up to the driver exactly *when* to apply the packet fields to the DUT. As shown by the driver code snippet below, method **get_next_item()** is called on rising edges of **PKT_CLK**. The other two sequences in the test suite, **SEQ2** and **SEQ3**, are coded in a similar manner:

```
//Driver code gets next packet from SEQ1 on SQR
//at the very beginning of each PKT_CLK cycle:
  @(posedge VDIF.PKT_CLK);
  seq_item_port.get_next_item(TX_PKT);
```

UVM refers to adjustable control variables like **TRIALS** and **INACTV** as *knobs*. They are set from the simulator's run-time command line, as described in Section IV, without recompiling testbench code. We can thus, implement a complex test suite, suitable for an analog/mixed-signal DUT, as a series of well-defined UVM sequences. And we can adjust the test length at run time, enabling us to either increase functional coverage or reduce the test iterations. The next section describes how these three untimed sequences are run in consecutive order from the highest level.

IV. Test Suite Class

The highest-level object in the UVM component hierarchy of Fig. 1 is **TEST_SUITE**. It instantiates an environment **ENV**, with all of its subcomponents. Class **TEST_SUITE** is derived from **uvm_test**, but is never directly instantiated itself. Instead, the UVM phasing system automatically creates an instance named **uvm_test_top**, and begins to run it at simulation time 0. As outlined in the next section, this will occur when **run_test("TEST_SUITE")** is invoked.

As Cummings [13] has pointed out, it seems intuitive to refer to this test-specific class as **TEST_SUITE** instead of the more conventional "testcase," since only one such test can be run during a UVM simulation. Thus, **TEST_SUITE** in our example comprises three individual test sequences, defined in the previous section. It can in general be more complex, involving parallel or hierarchical sequences (in which a top sequence includes one or more subsequences).

A. Launching Three Sequences

The code for our three sequences is outlined in Fig. 7. Task **build_phase()** gets the values for adjustable knobs; we defer its details to the next subsection. Task **run_phase()** uses the knob values to launch individual sequences of desired length. Notice the optional call to method **set_drain_time()**. This prevents **TEST_SUITE** from ending the simulation abruptly when the last packet in sequence **SEQ3** is applied to the DUT. By setting one cycle of *drain time*, we ensure that the simulation continues past mid-cycle. This is a simple way to allow the very last stimulus to propagate through the analog DUT, so the monitor can accurately measure the filter's last response amplitude [14].

The next statement in the task is a call to **`raise_objection()`**. This tells the UVM phasing system that the test suite would *object* to halting the run until the task body is finished. Without it, the run could halt prematurely. Next, **`SEQ1`** is created, given a **`TRIALS`** knob, and started on sequencer **`SQR`**. The other two sequences are launched in the same way. Recall that statements in a SystemVerilog task are executed consecutively, as if enclosed in a **`begin-end`** block [15]. All three sequences thus run in textual order. Finally, phase method **`drop_objection()`** can be called.

```
class TEST_SUITE extends uvm_test;
  function void build_phase(…);...
  /* TRIALS knob extracted from command line. */
  endfunction: build_phase

  task run_phase(uvm_phase phase);
    SEQ_FILTER  SEQ1;  //Normal filtering for TRIALS cycles.
    SEQ_BYPASS  SEQ2;  //Power-down, then wait, for INACTV cycles.
    SEQ_RESUME  SEQ3;  //Resume normal filtering for 4 more cycles.
  //Allow for last clock cycle to complete:
    phase.phase_done.set_drain_time(this, 1000us);
    phase.raise_objection(this);
  //Sequence 1.
    SEQ1 = SEQ_FILTER::type_id::create("SEQ1");
    SEQ1.TRIALS = TRIALS;     //Pass knob to SEQ1.
    SEQ1.start(E.AGNTD.SQR);  //Start SEQ1 on SQR.
  /* Code for Sequence 2. */
  /* Code for Sequence 3. */
    phase.drop_objection(this);
  endtask: run_phase
```

Figure 7. A Task to Run Three Consecutive Sequences

B. *Passing Down Knobs*

UVM is geared towards reconfiguring test details *without* having to recompile testbench code. In compliance with this methodology, we migrated away from defining **`TRIALS`** and **`INACTV`** as ordinary SystemVerilog parameters. We instead pass these knobs down from the run-time command line. The example below is for the **VCS** environment:

```
simv  +TRIALS=36  +INACTV=4  +UVM_NO_RELNOTES  -l LOG
```

The knob values must be known prior to any time advance. We thus put the extraction code into the **`TEST_SUITE`** method **`build_phase()`**, as listed in Fig. 8. To extract the values, we used the **`uvm_cmdline_processor`** method **`get_arg_value()`**. It is a UVM-style refinement of Verilog's familiar **`$value$plusargs()`** system function [16]. As is typical with the UVM base-class library, we first create a local instance of processor **`CLP`**, and then proceed to invoke its **`get_arg_value()`** method. Extracted values **`36`** and **`4`** are seen as strings, just as in Verilog. But they are easily converted into integers by SystemVerilog's handy **`atoi()`** string method [17].

```
class TEST_SUITE extends uvm_test;
  . . . . . .
//Declare command-line plusargs:
  int    TRIALS,  string TRIALS_s; ...
  function void build_phase(uvm_phase phase);
  //Create CLP instance:
    uvm_cmdline_processor CLP;
    CLP = uvm_cmdline_processor::get_inst();
  //Extract command-line plusargs:
    CLP.get_arg_value("+TRIALS=", TRIALS_s); ...
  //Convert string to int:
    TRIALS = TRIALS_s.atoi; ...
    . . . . . .
  endfunction: build_phase
```

Figure 8. Passing Command-Line Arguments Downward

Once the integer value of a knob such as **`TRIALS`** is extracted from the command line, it still must be passed down to those sequences that reference it. This was illustrated by a corresponding assignment statement back in Fig. 7.

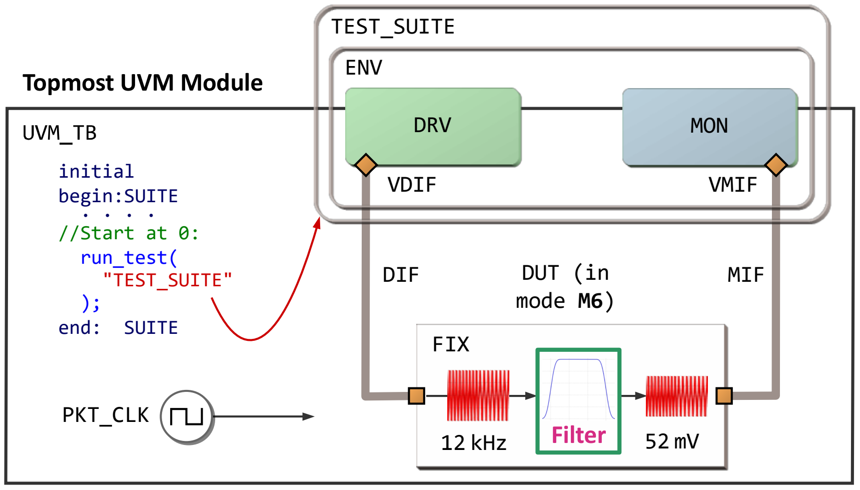
V. Top-Level Module

The top-level module **`UVM_TB`** is depicted schematically in Fig. 9. It is this module that is actually elaborated and simulated by **VCS** (using the XMODEL plug-in extension). A typical XMODEL *manifest file* is shown in Fig. 10.

A. Selecting the Test Suite

`UVM_TB` contains a procedural code block, the **`initial`** block in Fig. 9. It invokes **`run_test("TEST_SUITE")`** at time 0. The phasing system uses this information to create instance **`uvm_test_top`**, and to run it, as indicated by the red arrow in the figure. Though UVM does support command-line selection of different test suites, we explicitly coded the test-suite string name into the **`initial`** block, since our simple test plan only requires this single suite.

This topmost module not only instantiates the fixture submodule along with the DUT, but also instantiates the two interface buses, **`DIF`** and **`MIF`**. They carry fixture signals to and from driver and monitor, via the virtual interfaces. Bundled into these buses are **`PKT_CLK`** and an asynchronous active-high **`RST`**, generated by SystemVerilog code.



```
initial
begin:SUITE
  . . . .
//Start at 0:
  run_test(
    "TEST_SUITE"
  );
end:  SUITE
```

Figure 9. Topmost UVM Module

B. The XMODEL Command Line

Notice in Fig. 10 that XMODEL can pass on any valid compile-time or run-time option to the host logic simulator. Options are prefixed and suffixed with double dashes. The simulator command is then simply: **`xmodel -f man.f`**

```
--sim vcs
--top UVM_TB
--timescale 1us/1us
--elab-option -ntb_opts uvm-1.2 --
--sim-option  -assert quiet --
«packaged class files»
FILTER.sv
FIXTURE.sv
UVM_TB.sv
```

Figure 10. Typical Manifest File (**`man.f`**) for an XMODEL Run

The previous sections have outlined the various components of the UVM-compliant testbench we used in verifying our analog/mixed-signal DUT. In the remaining sections, we delve into a few components of interest in more detail.

VI. Custom Scorecard Printing

A UVM-compliant testbench relies on UVM macros or methods—like **`uvm_report_info()`**—to print packet data or scoreboard results to a log file or screen. Traditional Verilog **`$display()`** statements are not recommended [18].

Typically, a UVM testbench outputs a large volume of information. The UVM printing routines all have a *verbosity* argument, which users can set from the command line to adjust the level of detail printed out for an individual run.

To avoid cluttering the scoreboard with print-related code, we encapsulated printing routines, printer knobs, string processing, and header and footer strings into a subclass named **SCORECARD**. This subclass has a table named **DATA**, declared as an associative array in the listing of Fig. 11. As each transaction is simulated, the scoreboard will store both stimulus and response data into a row of the table. The **TX_PKT** tag serves as an integer index **N** into the table.

Each row of **DATA** is a structure, with specific fields for the applied frequency and mode, filter state, and measured actual and expected gain. When the scoreboard is done storing this tabular data, the scorecard **SCD** can easily print it out in UVM tabular format, by iterating with a **foreach()** loop. An associative array is an effective SystemVerilog storage option whenever the row count is not known in advance [19]. The custom printing routine in Fig. 11 is over 60 lines of code; the figure below lists only its key features. It is invoked *indirectly* by calling **SCD.print**().

```
//SCORECARD FOR TABULAR PRINTING:
  class SCORECARD extends uvm_object;
    . . . . . .
  //Associative array of structures:
    DATA_t  DATA[int];

  //Declare local print-policy object:
    LOCAL_PRTR  printer;

  //Custom tabular print routine:
    function void do_print(uvm_printer printer);
    //Disable "Name-Value" header, type and size:
      printer.knobs.header     = 1'b0;
      printer.knobs.type_name  = 1'b0;  . . . .
     //Print fixed table heading string:
      printer.print_generic("", "", 0, HEADING_s);
    //Print DATA[] row by row:
      foreach (DATA[N])
      begin:PRINT_LINE
        . . . . . .
      //Print table row as a generic line:
        printer.print_generic("", "", 0, {TX_LINE_s, RX_LINE_s});
      end:  PRINT_LINE
      printer.print_array_footer();
    endfunction: do_print
```

Figure 11. Scorecard Printing Routine

It is critical to declare a *local* UVM print-policy object. To customize the tabular print format, we needed to adjust several UVM printer knobs. If this is done *globally*, then all tabular UVM printing—including testbench topology—is impacted. With **printer** localized, other tabular printing remains unaffected. We can selectively disable unneeded features of the printout , such as the default *Name-Value* header, and the *type* and *size* columns.

UVM's tabular printing is intended to display an *object* with all its properties. Our goal here is not to display object **SCD**, but to print out a custom table summarizing all the transactions applied to an analog/mixed-signal device. Thus, we made heavy use of the UVM **print_generic()** method [20]. It can print an arbitrary string *value*.

In Fig. 11, **print_generic()** is called from inside a **foreach()** loop, which visits every row in the **DATA** array and prints it out. The same method can be employed outside the loop, to display fixed header and footer information.

The scoreboard works closely with the scorecard to evaluate and print the simulation results. It receives incoming packets from the driver and the monitor, as shown in Fig. 1. It then passes down the relevant packet fields to the **SCD** array **DATA**. The scoreboard code snippet below shows how driver packet fields are stored in the structure members of the array, using the packet tag as index. For printing purposes, a shorthand mode name like **M4** is also extracted:

```
//Store input fields in DATA:
  SCD.DATA[N].fINT = TX_PKT.fINT;
  SCD.DATA[N].MODE = TX_PKT.MODE;
```

Fig. 12 shows the results for a run of 36 **TRIALS**—including a UVM topology table, to show it was not affected. Details of how the scoreboard compares actual to expected filter gain at an applied frequency and mode, by looking up voltage amplitudes from an HSPICE-generated reference table, are found in our previous articles [5,6]. Notice the footer, which demonstrates that worst-case discrepancy in transfer gain between HSPICE and XMODEL is below 2%.

```
----------------------------------------------------------------
Name                        Type                        Value
----------------------------------------------------------------
uvm_test_top                TEST_SUITE                   @343
  E                         ENV                          @356
    AGNTD                   AGENTD                       @365
    AGNTM                   AGENTM                       @374
    COVG                    COVERAGE                     @393
    SCB                     SCOREBOARD                   @383
----------------------------------------------------------------

SCD   @392
----------------------------------------------------------------
TX_  fINT  MODE STATE   RX_     gACT        gEXP        gERROR
TAG   kHz  Name  Name   TAG   (OUT/IN)    (HSPICE)   (gACT-gEXP)
----------------------------------------------------------------
  1    34   M7  FILTER    1  0.80081911  0.79615200  0.00466711
  2    45   M0  FILTER    2  1.18788707  1.17873200  0.00915507
  3    10   M4  FILTER    3  0.24548713  0.24592900 -0.00044187
  4    16   M1  FILTER    4  0.86157682  0.85269000  0.00888682
 ..    ..   ..  . . .         . . . .     . . . .      . . . .
 37    XX   MX  BYPASS   37  0.00000000  0.00000000  0.00000000
 38    XX   MX  BYPASS   38  0.00000000  0.00000000  0.00000000
 39    XX   MX  BYPASS   39  0.00000000  0.00000000  0.00000000
 40    XX   MX  WAIT_1   40  0.00000000  0.00000000  0.00000000
 41   110   M4  RESUME   41  0.69637779  0.68144600  0.01493179
 42    76   M7  RESUME   42  0.70367702  0.69537100  0.00830602
 43    12   M6  RESUME   43  0.51965732  0.51948100  0.00017632
 44    63   M1  RESUME   44  0.89859033  0.88735100  0.01123933
----------------------------------------------------------------
                          Maximum |gERROR| per run:  0.01713050
----------------------------------------------------------------
```

Figure 12. Default UVM Topology and Custom Scorecard Results

Several rows of the simulation transcript in Fig. 12 are shaded in gray. These represent an *inactive* DUT, either in power-down mode or waiting for supply voltage to be restored. During these cycles, no useful transfer-gain data is acquired. But, as outlined in the next section, we can use this time to assert and check *analog properties* of the DUT.

VII. Analog Assertions

A simulated test suite is shown in Fig. 13. This test applies sixteen **TRIALS** to the filter in normal operating mode. Coverage metrics are collected. The DUT then enters bypass state for four cycles. In this state the op-amp is powered down. By asserting analog design properties, however, we can still check important DUT features like those below:

- The supply leakage-current level **IDD** of the powered-down op-amp should be less than 5 nA during **BYPASS** state.
- The nMOS bias voltage **VBN** should recover during the subsequent **WAIT_1** state to its nominal level, 700 mV ± 50.

The downward red arrow in the simulation waveforms indicates a *failure* of the leakage-current assertion, due to a deliberately-injected error. Such assertions are a powerful supplement to ordinary UVM test sequences, since they are localized to specific circuit nodes or branches. A failure thus *pinpoints* a problem area in the device under test.

The SystemVerilog assertion (SVA) code for checking leakage current was presented in our previous work [5,6]. Here we show the SVA code and XMODEL hardware for checking on the recovery of the nMOS bias-voltage level. Recovery from bypass/power-down begins in the **WAIT_1** state. Fig. 14 lists the relevant SVA code, a property and its assertion, alongside the fixture circuit used to measure the bias-voltage level at mid-cycle during the wait state.

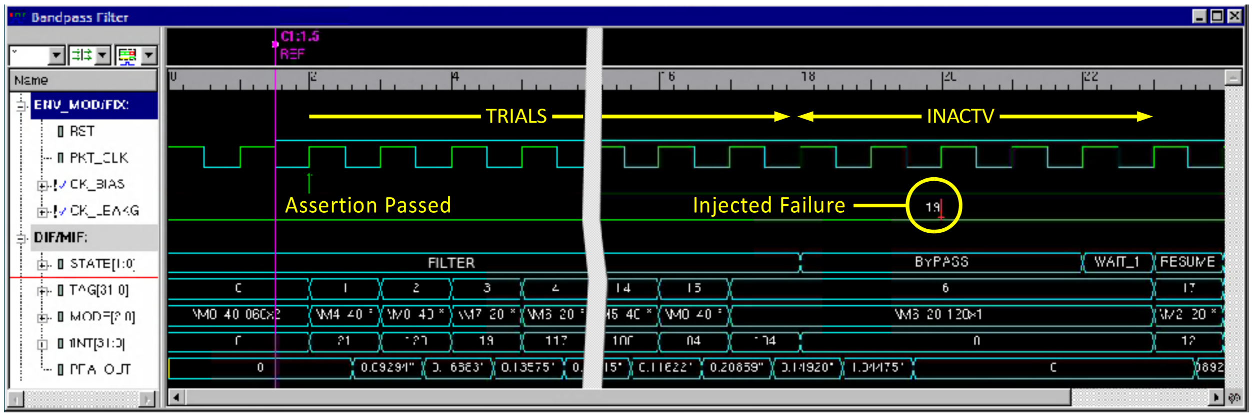


Figure 13. Simulated Test Suite with Power-Down and Recovery

Prior to this wait state, the CMOS transmission gate was switched off. As soon as the DUT enters **WAIT_1**, control bit **ctl_byp** is deasserted by the test sequence. The switch closes, and node voltage **vbias** should quickly return to its nominal level. We bring this node voltage out to an **xreal** variable **VBN_x**, using an **assign** statement. It is fed to the input of primitive **meas_value**. This samples its input voltage at around mid-cycle, triggered by the edges of **TICK**. The output of the primitive is **real** variable **VBN**. It is legal to reference **VBN** in SVA property expressions, provided that the overall expression evaluates to a Boolean. This check is triggered only during a **WAIT_1** cycle:

```
//Bias recovery leads:
  xreal VBN_x;  real VBN;
//Bias node inside of DUT:
  assign VBN_x = DUT.vbias;

//Bias voltage at TICK edge:
  meas_value
    M_VAL( //Real-valued output VBN:
      .in(VBN_x),.out(VBN),
      .trig(TICK_x)
    );

//Check nMOS bias during WAIT_1:
  property BIASING_pro;
    @(posedge FREQ_IN.PKT_CLK)
      $rose(FREQ_IN.STATE == WAIT_1) |->
        (VBN >= 0.650) &&  //700 mV ± 50.
        (VBN <= 0.750);
  endproperty: BIASING_pro

  CK_BIAS:
  assert property (BIASING_pro)
    «Report pass; else report failure.»
```

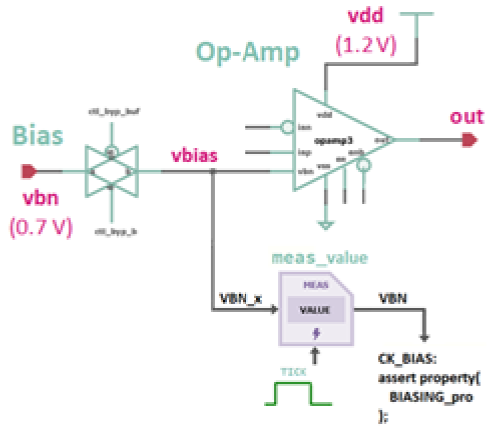


Figure 14. Asserting the Recovery of Bias Voltage

With the aid of XMODEL measurement and conversion primitives, a wide variety of analog assertions can thus be implemented, using the full range of SVA syntax. No language extensions are needed. This enables us to provide for coverage of design-specific analog/mixed-signal device properties—even during cycles when the DUT is inactive.

VIII. Functional Coverage Metrics

During normal DUT operation, we collect functional-coverage metrics utilizing a stand-alone coverage object **COVG**—in contrast to the *embedded* coverage class used in our previous OOP-based testbench [5,6]. This would enable the UVM factory substitution of test-specific coverage objects during different test suites.

The stand-alone coverage class listed in Fig. 15 is derived from **`uvm_subscriber`**. This base class has a built-in TLM port, represented by the small white circle icon on the **`COVG`** object in Fig. 1. Broadcast packets reach this port carrying the **`fINT`** and **`MODE`** values driven into the DUT at the start of a cycle. Since these values are randomly chosen, there is no guarantee our test suite will adequately verify filter operation for *all* its valid frequencies, modes, and combinations thereof. The role of cover group **`CVG`** is to gather metrics for reporting percent coverage statistics.

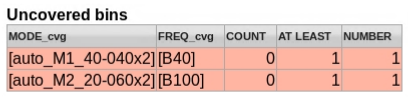
```
class COVERAGE extends uvm_subscriber #(PACKET);
 . . . .
//Coverage group for mode and frequency:
  covergroup CVG;
  //Applied input-frequency bins (6):
    FREQ_cvg: coverpoint TX_PKT.fINT
      {
        bins B10  = {[ 10: 19]};
        . . . .
        bins B100 = {[100:120]};
      }
  //Applied input-mode values:
    MODE_cvg: coverpoint TX_PKT.MODE;
  //Cross-coverage:  MODE x fINT
    CROSS_cvg: cross MODE_cvg, FREQ_cvg;
  endgroup: CVG

  virtual function void write(PACKET t);
    TX_PKT = t;   //Copy PKT pointer.
    if (TX_PKT.STATE == FILTER ||
        TX_PKT.STATE == RESUME
    )
      CVG.sample();
  endfunction: write
```

Figure 15. Collecting Functional-Coverage Metrics

The coverpoint for **`fINT`** specifies six explicit bins. Since **`MODE`** is enumerated, no explicit specification is needed. Metrics are collected every time **`CVG.sample()`** is called. To avoid collecting coverage data for an *inactive* DUT, we employ an **`if`** statement to conditionally sample only in states **`FILTER`** or **`RESUME`**. Notice that the call to **`.sample()`** occurs within a **`write()`** function, invoked whenever the driver broadcasts a new packet out of its analysis port.

As explained more fully in [5,6], the toughest coverage goal to meet is the *cross coverage* between six frequency bins and eight modes—a total of 48 combinations. The probability of hitting any one combination is only about 2%. Fig. 16 shows typical results. The statistics at left are obtained using method **`CVG.CROSS_cvg.get_coverage()`** for a run of 185 **`TRIALS`**—without the need of an external utility or HTML browser. The more detailed statistics at right are extracted using the **VCS `urg`** utility, and then displayed in a browser. They pinpoint which combinations remain uncovered. We reached 100% cross-coverage for a run of 195 **`TRIALS`**, using a random seed value of 3947.

```
UVM_INFO @ 196.000 ms: uvm_test_top.E.COVG [COVER]
--COVERAGE STATISTICS----
  Freq. Coverage: 100.00%
  Mode  Coverage: 100.00%
  Cross Coverage:  95.83%
```


Uncovered bins

MODE_cvg	FREQ_cvg	COUNT	AT LEAST	NUMBER
[auto_M1_40-040x2]	[B40]	0	1	1
[auto_M2_20-060x2]	[B100]	0	1	1

Figure 16. Coverage Statistics for Run of 185 Trials

In this section, we have briefly described a stand-alone coverage class, derived from **`uvm_subscriber`**, suitable for a digitally-programmable analog/mixed-signal DUT. Keeping the functional coverage code separate—instead of embedding it within other UVM components like a scoreboard or monitor—is often good practice [21]. As is evident in Fig. 15, coverage code tends to be DUT-specific, and may require connections to remote parts of the environment.

IX. Conclusions

This work has demonstrated a UVM testbench for verifying a digitally-programmable analog/mixed-signal DUT. An extension of previous work using SystemVerilog OOP code, the present testbench brings a number of UVM-style enhancements—modular components, configurable messaging and test length, TLM pathways, functional-coverage objects—into the AMS domain. The testbench utilizes standard UVM components to verify transfer gain for an audio bandpass filter at random frequencies and passband modes. A UVM scoreboard evaluates the results against a SPICE model. Tabular UVM printing is used to summarize the results. During power-down mode, leakage current and bias voltage levels are checked using analog assertions. This XMODEL-based flow achieved an accuracy to within 2% of HSPICE, with no sacrifice in speed, while requiring neither co-simulation facilities nor user-defined RNM models.

References

[1] Barnasconi, Einwich et al., "AMS System Level Verification and Validation using UVM in SystemC-AMS: Automotive Use Cases," IEEE Design & Test, 2014.

[2] Georgoulopoulis, Giannou, Hatzopoulos, "UVM-based Verification of a Mixed-Signal Design Using SystemVerilog," PATMOS 2018.

[3] Freitas & Santonja (Freescale), "UVM Ready: Transitioning Mixed-Signal Verification Environments to UVM," DVCon Euro 2014.

[4] "UVM-AMS: A UVM-Based Analog Verification Standard." [Online]. Available: https://2021.dvcon.org/presentation/workshop/uvm-ams-uvm-based-analog-verification-standard.

[5] Dančak, "SystemVerilog OOP Testbench (Part 1)." [Online]. Available: https://www.researchgate.net/publication/346061868_SystemVerilog_OOP_Testbench_for_Analog_Filter_A_Tutorial_Part_1.

[6] Dančak, "SystemVerilog OOP Testbench (Part 2)." [Online]. Available: https://www.researchgate.net/publication/350412143_SystemVerilog_OOP_Testbench_for_Analog_Filter_A_Tutorial_Part_2.

[7] Scientific Analog, Inc. XMODEL. [Online]. Available at: https://www.scianalog.com.

[8] J.-E. Jang, et al., "True Event-Driven Simulation of Analog/Mixed-Signal Behaviors in SystemVerilog: A Decision-Feedback Equalizing (DFE) Receiver Example," IEEE Custom Integrated Circuits Conference (CICC), Sep. 2012.

[9] Cummings, "OVM/UVM Scoreboards: Fundamental Architectures," SNUG 2013. §5.

[10] Scientific Analog, Inc. XMODEL. [Online]. Available at: https://www.scianalog.com/xmodel, FEATURE 2.

[11] Brennan, Ziller et al., "The How To's of Advanced Mixed-Signal Verification," DVCon Euro 2015, slide 35.

[12] S. Vasudevan, *Practical UVM: Step-by-Step Examples*, §13.6.1. San Jose, CA: 2016.

[13] Cummings, "UVM Message Display Commands: Capabilities, Proper Usage and Guidelines," SNUG Austin 2014. §7.4.1.

[14] Dave Rich blog, *About Drain Time in UVM,* on Stack Overflow, 26 Dec. 2017. [Online]. Available at: https://stackoverflow.com/questions/47976706/about-drain-time-in-uvm*.

[15] IEEE Std 1800-2017 SystemVerilog Language Reference Manual, §13.3 Tasks.

[16] S. Vasudevan, *Practical UVM: Step-by-Step Examples*, §7.6. San Jose, CA: 2016.

[17] IEEE Std 1800-2017 SystemVerilog Language Reference Manual, §6.16.9 Atoi().

[18] Cummings, "UVM Message Display Commands: Capabilities, Proper Usage and Guidelines," SNUG Austin 2014. §8.

[19] IEEE Std 1800-2017 SystemVerilog Language Reference Manual, §7.8 Associative Arrays.

[20] IEEE Std 1800.2-2017 Universal Verification Methodology Language Reference Manual, §16.2.3.3 print_generic.

[21] Bromley& Litterick (Verilab), "Effective SystemVerilog Functional Coverage: design and coding recommendations," SNUG 2016. §3.1.