A UVM SystemVerilog Testbench for Directed and Random Testing of an AMS Low-Dropout Voltage Regulator

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

***Abstract*-A UVM-compliant testbench is developed to apply benchtop-style directed tests to an on-chip low-dropout CMOS voltage regulator. Eight directed tests verify the regulator's DC and AC response to line and load fluctuations, its programmed operating modes, power-supply rejection, etc. To enhance the effectiveness of the low-level directed tests in reaching unexpected corner cases, we apply high-level UVM randomization techniques to generate a randsequence of transient tests. Sub-cycle timing for stimulus and response is managed by means of the uvm_event_pool. Our goal is to present the UVM testbench mechanisms and coding techniques that proved most effective in verifying the circuit-level functionality of analog/mixed-signal blocks—a topic outside the usual scope of chip-level UVM testbench development.**

I. Introduction

Universal Verification Methodology (UVM) has dramatically advanced the state of the art in verifying complex system-on-chip (SOC) designs. Its innovations have raised the level of abstraction at which verification is done. Yet, for on-chip analog/mixed-signal (AMS) IP blocks, directed testing at a circuit level [1] is still a must. AMS IP blocks on a chip have to meet strict analog requirements such as gain, bandwidth, linearity, noise immunity, and overshoot.

Even the lowest-level directed tests, however can be more effective when enhanced with the constrained random verification (CRV) capabilities of UVM. In this paper, we develop a UVM-compliant testbench to apply CRV-driven directed tests to an AMS circuit often found on SOCs. Our design-under-test (DUT) is a programmable 45-nm CMOS low-dropout (LDO) voltage regulator. A simplified block diagram of the transistor-level regulator appears in Fig. 1.

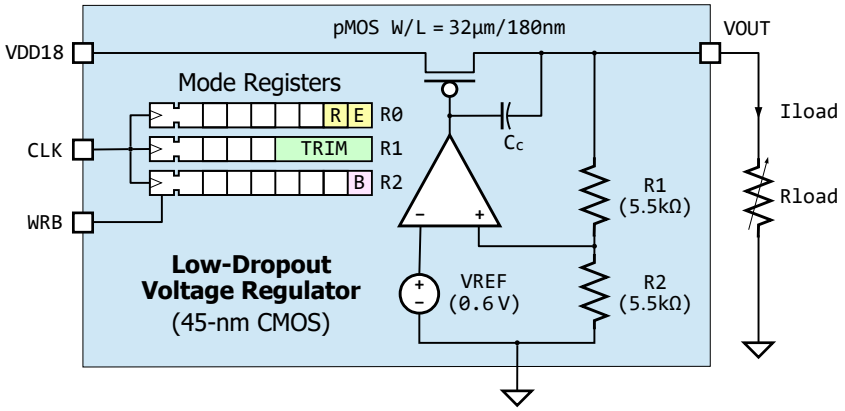


Figure 1. CMOS LDO Voltage Regulator Block Diagram

Our testbench includes eight directed tests. In addition, it can apply a **`randsequence`**-based test suite, alternating randomly between line-transient and load-transient tests—further stressing the DUT. During this random sequencing, circuit conditions for each directed test are carried over to the next. Imposing such reactive line and load conditions can better approximate the real-world demands faced by an on-chip regulator—and reach unanticipated corner cases. This random sequence of reactive tests detected more numerous, and more serious, failures than did individual tests.

---

1 C. Dančak is also a SystemVerilog instructor with UC San Diego, Dept. of Extended Studies, La Jolla CA 92093, USA.

The analog input to the DUT in Fig. 1. is **VDD18**, an unregulated supply voltage with a nominal value of 1.8 V. This line input is fed into a large pMOS pass transistor, whose conductivity is controlled by a continuous-time linear feedback loop. Regulated output voltage **VOUT** is typically 1.25 V. The load current can be adjusted through **Rload**.

Our testbench must verify the LDO's ability to regulate **VOUT** under fluctuating conditions of line voltage and load current. And it must exercise all the LDO's program modes, configured by writing control bits into registers R0–R2. These modes are: normal VOUT regulation; trimming of VOUT level; retention during power-down; or bypassing the entire regulator for automated testing. Asserting **WRB** enables writing to addressed registers on rising **CLK** edges.

We met these verification goals with the AMS *directed tests* in Fig. 2a. Each test is implemented as a UVM class derived from **uvm_test**. It launches a single sequence during **run_phase**, which generates and randomizes a data packet. Test **LOAD_REGLN**, for example, randomizes the real variable **VDD18_rnd**, constraining it to a valid input range. To run any test, its class name is specified on the simulator command line, with an option like the following:

```
+UVM_TESTNAME=TEST_LOAD_REGLN
```

Our testbench uses mostly standard UVM components. Nevertheless, components like the driver and monitor must be made aware of what test is currently running. We defined the enumerated *test type* in Fig. 2b to specify the active test anywhere in testbench code. Prefixing this type with **TEST_** yields the class name for that particular test.

RAND_TRANS, the last test in Fig. 2, is the **randsequence**-based test suite that randomly alternates between the **LINE_TRANS** and **LOAD_TRANS** tests. This sequence of transient tests models abrupt changes in the line voltage and load current, due perhaps to on-chip power-management phases or memory-usage spikes. For each directed test in the sequence, we *sample and hold* key circuit conditions, carrying them over as initial conditions for the next test.

II. UVM Testbench Organization

Fig. 3 is a block diagram of our testbench. Class objects appear as rounded rectangles, and modules as ordinary rectangles. Interface buses **DIF** and **MIF** are hierarchical. Analog bus signals like **A.VOUT** or **A.ILOAD** (purple wiring) are thus listed separately from digital bus signals like **D.TEST_TYPE** or **D.WRB** (green wiring), enhancing readability.

The voltage-regulator DUT is instantiated in a submodule named **FIXTURE**. This submodule also encapsulates all the XMODEL analog resources—DC and AC voltage sources, analog selectors and adders, a variable load resistor, and the various measurement primitives [2]—needed to drive or monitor the DUT's voltages and currents over time.

The transistor-level DUT was designed in Cadence ***Virtuoso***. From the DUT schematic, a SystemVerilog netlist of XMODEL primitives was extracted and characterized to SPICE accuracy, using the MODELZEN utility discussed in previous work [3]. This enabled us to simulate the AMS DUT itself, as well as the analog instrumentation around it, in pure SystemVerilog. This flow facilitates the rapid development of the benchtop-style AMS tests listed in Fig. 2a.

An Accellera UVM-AMS standard is currently under development to address the challenges of incorporating AMS verification into a UVM framework. The approach presented here, while in line with the emerging standard, does not require any of its mixed-signal-aware extensions, and is already in full use.

As Fig. 3 indicates, during each directed test a driver applies stimulus to the DUT inputs over interface bus **DIF**. A monitor captures the output response over bus **MIF**. After each test, the driver sends a single data packet **DRV_PKT** over a TLM path on up to the scoreboard **SCB**. Likewise, the monitor sends a single data packet **MON_PKT**. Based on the arriving stimulus and response data, **SCB** compares actual test results with the regulator's design specifications.

During **report_phase**, individual test results are printed. In the case of the **RAND_TRANS** test suite, the transcript prints a *scorecard* array **SCD**, summarizing at a glance the pass-fail results for all the individual tests that were run.

III. Isolating Class Hierarchy from Test Details

A major challenge in this work was to keep the high-level UVM class components independent of circuit-level test details. We wanted to develop a diverse set of analog/mixed-signal directed tests for the voltage regulator, often requiring iterations, without the need of tweaking existing UVM architecture to accommodate low-level test details.

By adopting the coding strategies below, we met this challenge. We were able to reuse most of the UVM classes and TLM pathways from previous work [4]. We implemented eight benchtop-style tests that can all be executed on the same UVM testbench, and are easily modified or extended—just like setting up wave generators, voltmeters, and other instruments on a lab benchtop. Three effective isolation techniques are outlined in the subsections below.

AMS LDO Voltage Regulator Tests				
Code	Test Type (TEST_TYPE_t)	Analog Quantity Measured	Result Compared Against Specs	Length (Cycles)
4'h0	IDLE_SUITE	Default enumerated test type.	—	—
4'h1	LINE_REGLN	$\Delta VOUT$, as $VDD18$ varies from min to max over specified range.	$\frac{\Delta VOUT}{\Delta VDD18} \div 1.25$	4
4'h2	LOAD_REGLN	$\Delta VOUT$, as $ILOAD$ varies from max to min over specified range.	$\frac{\Delta VOUT}{\Delta ILOAD} \div 1.25$	4
4'h3	PSRR_RATIO	Rejection of 10-kHz input hum.	$20 \log \frac{VOUT_{AC}}{VDD18_{AC}}$	4
4'h4	IDDQ_LEVEL	Quiescent $IDDQ$ for bleeder load.	$IDDQ$	2
4'h5	LINE_TRANS	$VOUT$ overshoot, undershoot.	V_{over}, V_{under}	2
4'h6	LOAD_TRANS	$VOUT$ undershoot, overshoot.	V_{under}, V_{over}	2
4'h7	TRIM_LEVEL	Trimmed $VOUT$ levels [n = 1–16].	$VOUT_n$	17
4'h8	CTRL_MODES	$VOUT$ for ENA, RET, BYP modes.	$VOUT$	7
4'h9	RAND_TRANS	Run LINE_TRANS, LOAD_TRANS tests in random sequence, with one test's conditions affecting the next.	$VOUT$	TRIALS

Figure 2a. Directed Tests for Voltage Regulator

```
/* ENUMERATED TEST_TYPE
 * Declared in package TYPES.
 */
   typedef
     enum bit [3:0] {
     //-Test Type- -Code-
       IDLE_SUITE = 4'h0, //Default.
       LINE_REGLN = 4'h1,
       LOAD_REGLN = 4'h2,
       PSRR_RATIO = 4'h3,
       IDDQ_LEVEL = 4'h4,
       LINE_TRANS = 4'h5,
       LOAD_TRANS = 4'h6,
       TRIM_LEVEL = 4'h7,
       CTRL_MODES = 4'h8,
       RAND_TRANS = 4'h9
     } TEST_TYPE_t;
```

2b. Packaged Test Types

A. Encapsulate DUT and Analog Resources in a Fixture

We found it best to encapsulate all analog/mixed-signal functionality inside of the **`FIXTURE`** submodule in Fig. 3. This includes the DUT, modeled in SystemVerilog as a circuit of XMODEL primitives like **`pmosfet`**, **`resistor`** and **`capacitor`**, and instrumentation primitives like **`sin_gen`** or **`meas_pp`**. It can also include analog assertion code, as demonstrated in a prior work [3]. The **`FIXTURE`** submodule interacts with higher-level testbench components only by exchanging signals over virtual interface buses **`VDIF`** and **`VMIF`**, and by awaiting global events that synchronize the timing of stimulus and response. The UVM hierarchy is thus effectively decoupled from low-level test details.

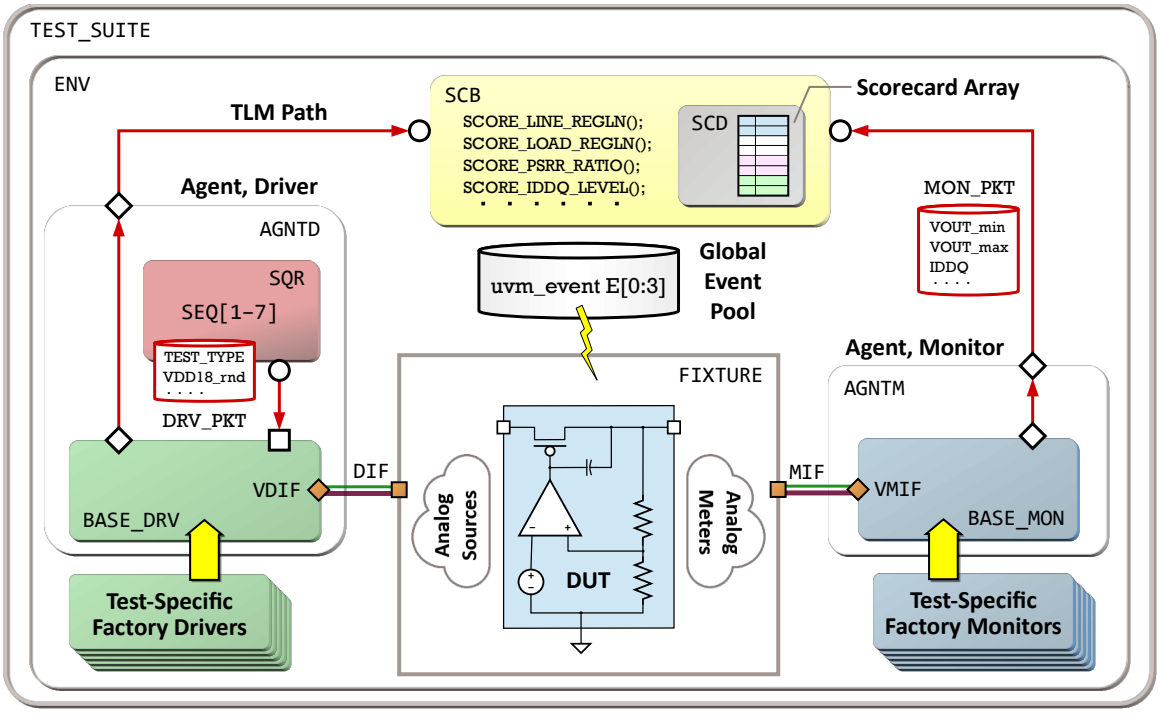


Figure 3. UVM Testbench Organization

B. *Build Factory-Replacement Drivers and Monitors*

We relied heavily on the factory replacement of drivers and monitors to keep UVM class components decoupled from testing details. The *base* driver and monitor in Fig. 3 are *independent* of any test. During **build_phase**, they are replaced by test-specific *derived* versions. Code Sample 1 shows a typical derived driver. As highlighted in red, it has just one test-dependent line of code—a call to a test-specific *command task* named **APPLY_LINE_REGLN_tf()**.

C. *Call Test-Specific Command Tasks*

As in Code Sample 1, the factory driver for **LINE_REGLN** calls the test-specific task **APPLY_LINE_REGLN_tf()** during **run_phase**. This command task contains all the test-specific details of applying minimum and maximum line voltages to DUT input **VDD18**. Its counterpart, monitor command task **CHECK_LINE_REGLN_tf()**, contains all the details of monitoring the **VMIF** bus, and capturing the DUT's output voltage **VOUT** after each line-input change.

Sample 2 shows the key code for **APPLY_LINE_REGLN_tf()**. First, a test type is established. Cycle by cycle, the task applies minimum and maximum line-input levels to the DUT, supplied by **dc_gen** sources in the fixture. They are multiplexed to the DUT input simply by driving selector values like **3'd2** onto the bus signal **VDIF.D.SEL_VDD**.

```
class LINE_REGLN_DRIVER extends BASE_DRIVER;
  `uvm_component_utils(LINE_REGLN_DRIVER)
  .   .   .   .   .   .
  task run_phase(uvm_phase phase);
    @(negedge VDIF.PON_RST);  //Wait till after reset pulse.
    @(posedge VDIF.TEST_CLK); //Begin test on cycle boundary.
    .   .   .   .   .   .
    seq_item_port.get_next_item(DRV_PKT); //Get one packet.

  //Call test-specific command task:
    APPLY_LINE_REGLN_tf(DRV_PKT, VDIF);

    seq_item_port.item_done();
    APD0.write(DRV_PKT);  //Forward packet to SCB via TLM.

  endtask: run_phase

endclass: LINE_REGLN_DRIVER
```

Code Sample 1. Factory Driver for **LINE_REGLN** Test

We coordinated the timing of stimulus and response with UVM events, as detailed in Section IV. The command **TRIGGER_EVENT()** called in Sample 2 encapsulates the event-based timing details, as shown in the rectangular inset. All events are *triggered* by a factory driver, and then *detected* by a factory monitor or by code blocks in the fixture.

This approach leads to a straightforward chain of command. First, the named test is specified from the simulator command line, as in Section I. A factory driver and monitor are built. In turn, they call the *command tasks* which apply test-specific stimulus and measure the DUT response. These factory components then forward the data packets **DRV_PKT** and **MON_PKT** up to the scoreboard. Aware of the **TEST_TYPE**, the scoreboard then evaluates test results.

Run named test → Build test-specific driver/monitor → Call command tasks → Apply stimulus; check response.

Our approach could readily be scaled up to a complex on-chip subsystem with a wide variety of AMS IP blocks. Using the command-task approach in subsection *C*, an analog verification engineer with minimal UVM expertise could conceivably write a detailed, circuit-specific directed test for an analog filter, equalizer, transceiver, ADC, DAC, or other AMS block, ready to be incorporated into a standard SOC-level UVM testbench.

IV. Synchronizing Stimulus and Response

Another challenge in this work was coordinating stimulus and response timing across widely-separated testbench components. Adding to the difficulty were the sub-cycle timing aspects of transient tests like **LOAD_TRANS**—with its abrupt changes in load current. We sought a robust yet intuitive scheme for writing a command task for each of the directed tests—some measuring voltage, others current—ideally working from hand-drawn timing diagrams.

```
/* APPLY LINE_REGLN STIMULUS */
  task APPLY_LINE_REGLN_tf(
    inout LDO_PKT  DRV_PKT,
    input AMS_IF_t VDIF
  );
    TEST_TYPE_t TEST_TYPE;

  //Specify this task's test type:
    TEST_TYPE = LINE_REGLN;

  //Packetize and send to fixture:
    DRV_PKT.TEST_TYPE = TEST_TYPE;
    VDIF.D.TEST_TYPE  = TEST_TYPE;

 /* Apply line inputs cycle by cycle */
    .   .   .   .   .
    @(posedge VDIF.TEST_CLK);
    VDIF.D.SEL_VDD = 3'd2;     //Select VDD18_max.
    TRIGGER_EVENT(3);
    .   .   .   .   .
    @(posedge VDIF.TEST_CLK); //End on cycle boundary.
  endtask: APPLY_LINE_REGLN_tf
```

```
//Body of TRIGGER_EVENT() task:
  #tSETTLE;   //Analog settling time.
  E[I].trigger(); //Trigger UVM event.
  TRIG_TIME = E[I].get_trigger_time();
  uvm_report_info("DCMD", $sformatf(
    "DRV triggered event E[I] at %t",
    TRIG_TIME), UVM_LOW
  );
```

Code Sample 2. Driver-Side Command Task for **`LINE_REGLN`** Test

UVM has a wide variety of methods for capturing low-level timing information. Marriott and Ronan [5] broadcast DDR retiming data to multiple byte lanes over a TLM analysis port. Bromley [6] suggested that capturing low-level signal transitions in a UVM data packet is much too heavyweight a mechanism. He proposed the **`uvm_event`** as a more effective candidate for transferring low-level timing data. We adopted Bromley's approach, using the global **`uvm_event_pool`**. It ensures crisply-synchronized stimulus and response across all testbench components.

UVM's global-event machinery is activated by the few lines of code in the rectangular inset to Fig. 4. We reused the event array **`E[0:3]`** across all tests. Calling static UVM method **`get_global()`** assigns a handle from the pool to each event **`E[J]`** in the array. These lines of code must be duplicated in each driver-side command task that *triggers* events, as well as in each monitor-side command task and fixture code block that *awaits* events. Because the event pool is a *singleton* object, accessing it from multiple components will only create a *single shared* instance [7].

Fig. 4 diagrams the entire event chain for a typical directed test, **`LOAD_REGLN`**, of length four cycles. A driver-side command task (upper left) triggers an event each cycle. A monitor-side command task (upper right) awaits the event after calling its **`wait_trigger()`** method. Notice that events are triggered *late* in the clock cycle, allowing plenty of time for analog outputs to settle. This enables the monitor code to immediately sample the measured value of **`VOUT`**. This simple voltage measurement was performed by an XMODEL **`meas_value`** primitive instantiated in the fixture.

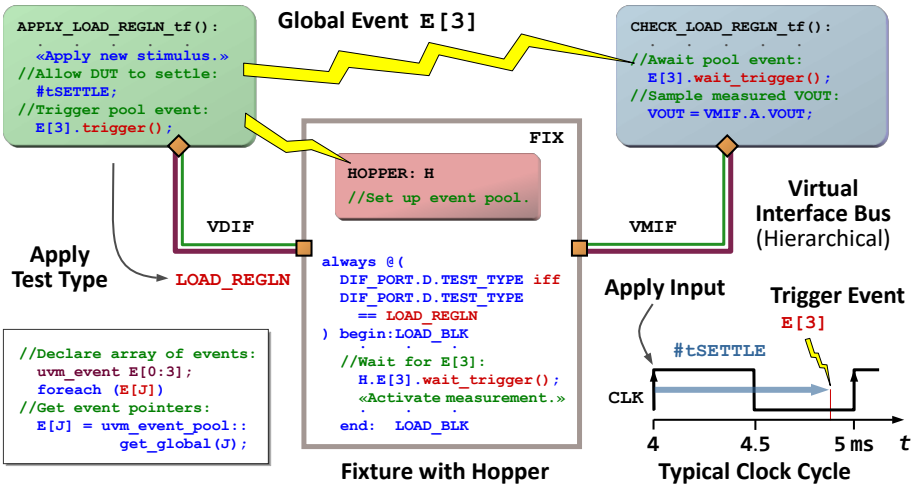


Figure 4. Event Chain for a **`LOAD_REGLN`** Test Across Testbench Components

The **FIXTURE** code that activates the voltage measurement appears in the middle of Fig. 4. An **always** code block looks for a change in the **TEST_TYPE** to **LOAD_REGLN**. It then waits for each successive event **E[J]** in that test, by calling the method **E[J].wait_trigger()**. In turn, the appropriate XMODEL measurement primitive is activated, as discussed in previous work [3]. Thus, measurements are performed for specific tests at precise times in the cycle.

Because a **uvm_event** object cannot be declared inside a submodule, we embedded a class named **HOPPER** into the fixture. It is derived from **uvm_object**. The fixture can then refer to any event by the short pathname **H.E[J]**.

An array of four UVM events proved adequate for all the tests in Fig. 2a. Even for the repetitive **TRIM_LEVEL** test, involving 16 cycles of trimming **VOUT**, we found it easy to *reuse* event **E[1]** sixteen times by calling **E[1].reset**.

V. Directed Line and Load Transient Tests

This section delves into the **LOAD_TRANS** and **LINE_TRANS** tests in greater detail. These tests verify the DUT's transient response to abrupt line or load changes—an aspect of its functionality which should be verified by any thorough SOC-level testbench. We show how UVM events can handle the sub-cycle timing aspects of transient tests.

The timing diagram in Fig. 5 shows these tests as operations within a longer random sequence. Each test applies a single line-voltage or load-current step, taking two cycles. Section VI covers generation of the random sequence.

The driver-side command task successively triggers events **E[0]** and **E[1]**, telling the monitor-side command task just when to measure **VOUT**. Measurements like **VOUT_sag** span a time interval, delimited by these two events. Regardless of transient ringing, primitive **meas_min** will accurately report the lowest peak within this time interval.

Test **LOAD_TRANS** starts on a clock edge at 7 ms in this specific sequence. (The 1-ms clock was shown in Fig. 4.) On this edge, the sampled-and-held circuit conditions from a previous test, **VDD18_sah** and **ILOAD_sah**, are applied to the DUT. This initializes current test conditions in a manner dependent on prior activity—or else utilizes a default.

On the next clock edge at 8 ms, the driver's command task steps the load current *up* (as in the figure) or *down,* using a constrained random value **ILOAD_rnd** supplied by **DRV_PKT**. This abrupt step in the load current will push the DUT's feedback loop to its limits—and the output **VOUT** typically reacts with significant undershoot or overshoot.

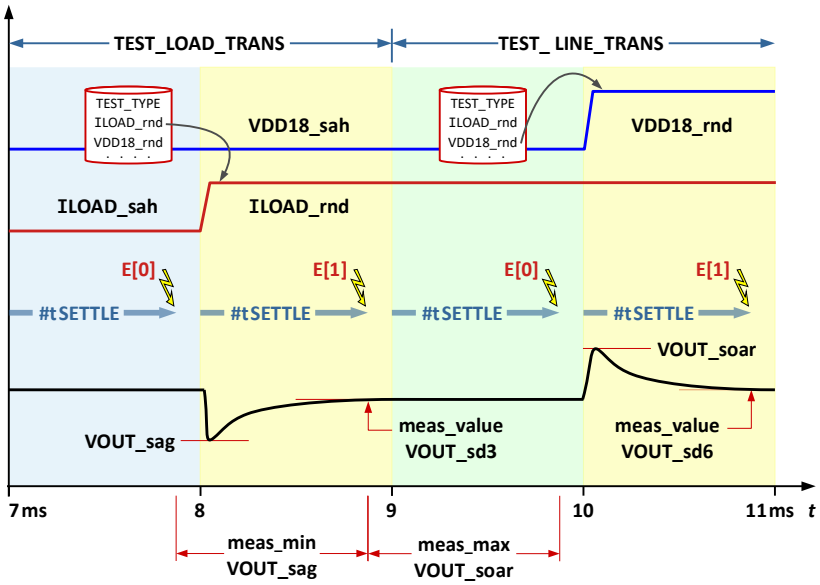


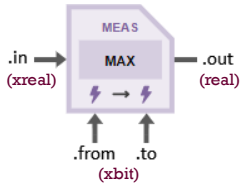
Figure 5. Timing Diagram for Consecutive Transient Tests

The monitor's command task measures the *undershoot* shown here—defined as the difference between the lowest peak (labeled **VOUT_sag**) and the subsequent steady-state value (**VOUT_sd3**). If the undershoot or overshoot on **VOUT** exceeds the design specification of 100 mV, a test failure is logged—indicative of a design flaw such as low speed, loop instability, or poor transient response.

The random sequence proceeds. At 9 ms in Fig. 5, the **LINE_TRANS** test begins, its circuit conditions carried over from **LOAD_TRANS**. At 10 ms, the line voltage steps *up* (as in the figure) or *down,* again stressing the feedback loop.

The step *height* is a constrained random value, determined by the difference **VDD18_rnd – VDD18_sah**. We reuse events **E[0]** and **E[1]** to activate a **meas_max** primitive. Sample 3 shows its SystemVerilog instantiation statement. It is activated by transitions on the one-bit **.from** and **.to** inputs, which in turn are driven by procedural fixture code:

```
/* INSTANTIATE meas_max PRIMITIVE
 * Measures maximum output voltage over a time
 * interval delimited by events E[0] and E[1].
 */
  meas_max  XP_VSOAR(
    .in(VOUT), .out(MIF_PORT.A.VOUT_soar),
    .from(TRIG_meas_x[0]), .to(TRIG_meas_x[1])
  );
```



Code Sample 3. Fixture Code to Instantiate **meas_max** Primitive

Measured undershoot and overshoot values are forwarded via **MON_PKT** to scoreboard **SCB**. Based on the test type, **SCB** calls a subroutine like **SCORE_LINE_TRANS()** to compare actual results against specifications. Our event-based scheme proved to be a natural and intuitive method for handling the timing for all drivers, monitors, and the fixture.

VI. Generating a Random Sequence of Transient Tests

UVM has several ways to generate a random series of operations. We employed SystemVerilog's **randsequence** generator [8] to create a randomly-alternating sequence of the two transient tests **LOAD_TRANS** and **LINE_TRANS**. Parameter **TRIALS** sets the sequence length. This test sequence, named **RAND_TRANS**, is intended to stress the DUT to its limits by better approximating the fluctuating real-world demands faced by an on-chip voltage regulator.

As in Sample 4a, the **randsequence** construct resides within the **new()** function for class **TEST_RAND_TRANS**. Whenever this test is run from the command line, and its class constructed, the **for** loop is executed. On each pass, either one of the **randsequence** productions—**RS_LINE_TRANS** or **RS_LOAD_TRANS**—is randomly chosen, and its associated rule executed. Each rule, in curly braces, simply appends another test type onto the associative array **ORDER**. When the loop is done, **ORDER** holds a randomly-alternating series of **LINE_TRANS** and **LOAD_TRANS** types.

```
class TEST_RAND_TRANS extends uvm_test;
  `uvm_component_utils(TEST_RAND_TRANS)
  uvm_factory FACTORY;
  ENVIRONMENT ENV;

//Associative array of test types:
  TEST_TYPE_t ORDER [int] = '{1:LINE_TRANS};

  function new(. . .);
    .   .   .   .   .
  //Generate random ORDER of test types:
    for (int I = 1; I <= TRIALS; I++)
    begin:RS_LOOP
      randsequence(TRANSIENT)
        TRANSIENT:  RS_LINE_TRANS := 7
                  | RS_LOAD_TRANS := 3;
        RS_LINE_TRANS: {ORDER[I] = LINE_TRANS;};
        RS_LOAD_TRANS: {ORDER[I] = LOAD_TRANS;};
      endsequence

    end:  RS_LOOP
  endfunction: new
  .   .   .   .   .
endclass: TEST_RAND_TRANS
```

Code Sample 4a. Generating a Randomized Test Order

```
/* FACTORY RAND_TRANS_MONITOR CODE
 * Execute the monitor command tasks
 * in prescribed randsequence ORDER:
 */
foreach (ORDER[T])
begin:XORDER
  .   .   .   .
//Call task based on test type:
  case (ORDER[T])
    LINE_TRANS:
      CHECK_LINE_TRANS_tf(MON_PKT, VMIF);
    LOAD_TRANS:
      CHECK_LOAD_TRANS_tf(MON_PKT, VMIF);
  endcase
  .   .   .   .
end:  XORDER
```

4b. Calling Monitor Command Tasks in Prescribed Order

One advantage of **randsequence** is that users can *weight* the test probabilities. (We wanted *equiprobable* tests, but found that weights of 7 and 3 gave a more even distribution.) This construct also allows enumerated-type names.

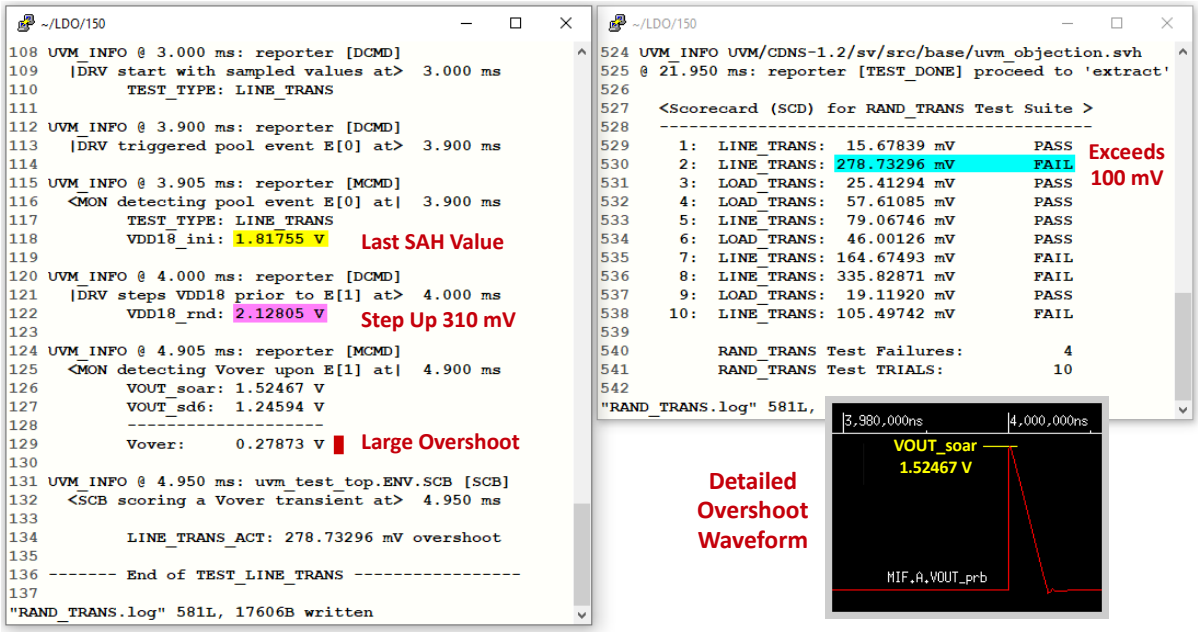


Figure 6. High-Overshoot LINE_TRANS Test During a RAND_TRANS Sequence

By generating this randomized array of test types at the very top level, then passing it down the hierarchy [9] to both the factory driver and monitor for the **`RAND_TRANS`** test, we have resolved a common SystemVerilog problem.

Figure 7. First Three Tests During the RAND_TRANS Sequence

The language does random ordering based on *thread locality*. Two independent processes, like a driver or monitor task, generate their *own* local ordering—they will not produce the *same* random sequence, unless we employ manual hierarchical-seeding techniques [10]. Our top-down approach guarantees that an *identical* random ordering of test types reaches both driver and monitor—in a manner impervious to code modifications or changes in logic simulator.

Once the **`ORDER`** array is passed down to the factory monitor and driver, they call their respective command tasks in lockstep order, as prescribed by the array. Sample 4b shows a segment of the **`RAND_TRANS_MONITOR`** code. Its **`case`** statement executes either the **`LINE_TRANS`** or **`LOAD_TRANS`** command task on each pass of the **`foreach`** loop. Similar code is found in **`RAND_TRANS_DRIVER`**. This approach could readily be extended to include additional tests.

VII. Simulated Results for a Rand_Trans Suite

This section demonstrates the results for a **`RAND_TRANS`** simulation which ran on Xcelium, with the random-seed option **`-svseed 3947`**. Altering this global seed will vary the random ordering of the **`RAND_TRANS`** sequence.

Fig. 6 shows portions of the transcript printed at the verbosity **`UVM_HIGH`**. At left, a particular **`LINE_TRANS`** test exhibits a high overshoot of 279 mV. This is presumably due to a large line-input step of 310 mV, determined by the difference **`VDD18_rnd - VDD18_sah`**. The inset at the right of Fig. 6 magnifies the overshoot waveform. Notice the overshoot is followed by a smaller undershoot; the monitor passes only the larger of the two to **`SCB`**. The scorecard portion of Fig. 6 summarizes all the tests in this particular sequence. Four transient tests failed out of a total of ten.

Fig. 7 displays digital, analog, and event waveforms for the same run. It zooms in on the first few tests of Fig. 6. At this timescale, the 279-mV overshoot is a tall, narrow spike. Notice that the spike occurs at the *start* of the cycle, in response to the input step at 4 ms. But the event **`E[1]`** that completes the **`VOUT_soar`** measurement occurs *later* in the cycle, at 4.9 ms, allowing plenty of time for the analog outputs to settle.

By randomizing the ordering of transient tests, and using the circuit conditions for one test to initialize the next, we stress the regulator's feedback loop and its design parameters to the utmost—making our UVM testbench more likely to reach the unanticipated corner cases prevalent on an actual mixed-signal silicon chip. Difficult-to-detect issues like unidentified race conditions, latch-up states, and unprogrammed register bits might well be revealed.

VIII. Conclusions

This work has presented a UVM-compliant testbench that can apply a diverse set of directed benchtop-style tests to an analog/mixed-signal voltage regulator, while isolating high-level UVM components from low-level test details. Techniques to meet these goals included factory drivers and monitors which invoked test-specific *command tasks.* The ability of directed tests to reach unanticipated corner cases was enhanced by generating a **`randsequence`**-based series of randomly-alternating line-transient and load-transient tests—using the circuit conditions for each test as initial conditions for the next. Stimulus and response were synchronized across testbench components—down to the sub-cycle timing level—with the UVM event pool. This work thus demonstrates testbench mechanisms and coding tactics that combine high-level CRV techniques with low-level directed tests, to maximize coverage of an AMS DUT.

IX. Acknowledgments

The author wishes to thank Jaeha Kim and Rafael Betancourt for many insightful and encouraging discussions.

References

[1] K. Jones (Rambus Inc.), “Analog and Mixed Signal Verification,” Part I.
[2] Scientific Analog, Inc. XMODEL. [Online]. See: https://www.scianalog.com/xmodel, FEATURE 2.
[3] C. Dančak, SystemVerilog OOP Testbench for Analog Filter: A Tutorial (Part 2). [Online]. See: www.researchgate.net/publication/350412143_SystemVerilog_OOP_Testbench_for_Analog_Filter_A_Tutorial_Part_2.
[4] C. Dančak, “A UVM SystemVerilog Testbench for AMS Verification: A Digitally-Programmable Analog Filter,” DVCon 2022.
[5] Marriott & Ronan, “Run-time Configuration of a Verification Environment: A Novel Use of the UVM Analysis Pattern.”
[6] J. Bromley (Verilab), “First Reports from the UVM Trenches,” §4.2.1.
[7] IEEE Std 1800.2-2017 Universal Verification Methodology Language Reference Manual, §10(c) Synchronization Classes.
[8] IEEE Std 1800-2017 SystemVerilog Language Reference Manual, §18.17 Random Sequence Generation.
[9] J. Bromley (Verilab), “Slicing Through UVM’s Red Tape” §IV.B, Euro DVCon 2016.
[10] D. Smith (Doulos), “Random Stability in SystemVerilog,” SNUG Austin 2013. §3.4.3.