UVM Testbench Automation for AMS Designs

J. B. David
Innophase, Inc.
2870 Zanker Rd, Suite 200
San Jose, CA 95134-2133

H. Chang
Designer's Guide Consulting, Inc.
3043 Meridian Ave #35
San Jose CA 95124

Abstract- In the world of design verification for analog and mixed-signal (AMS) Systems on Chips (SOCs) there are many problems, some of which are now relatively solved. AMS modeling has converged on Verilog-AMS and SystemVerilog real numbered modeling (SV-RNM), with simulator support available from major electronic design automation (EDA) vendors. Behavioral model development productivity is supported with tools available from some smaller EDA vendors. One of the remaining productivity gaps is in testbench automation. Digital design teams will often have a System-C, transaction level model (TLM) of the digital system under test from which both the RTL and a Universal Verification Methodology (UVM) testbench can be derived, however TLM does not work well as a specification for an analog or mixed-signal system, and UVM is a complicated stretch for the mixed-signal team to adopt. For smaller digital blocks where C or TLM models exist, or can be developed, as a reference specification, this specification will drive the development of a UVM bench for the block level design. However, for the equivalent level in the mixed-signal design comprising several circuits working together: an RF synthesizer, radio receiver or transmitter chain, or even an entire radio transceiver with many digital controls but little digital content, no simple and standard way of quickly creating a verification bench exists. This is where there is a convergence of a lot of activity ─ the system designer, analog lead, creation and update of the top-level schematic, designers need to start working together, and is therefore the place where there is often difficulty and a source for errors. It is also where circuit simulation is needed and becomes slow and often infeasible. This is the area that we address. We propose a standardized testbench architecture based on UVM and show a method to automate the construction of a bench for each design.

INTRODUCTION

Around 1990, when we were starting our careers, the power of digital design methodologies was being realized, and the availability of computers was expanding. Tools for analog and mixed-signal system analysis were also being explored. Several tools of the day were MAST [1], WATSCAD [2] and Easy5 [3], for mixed domain systems analysis, and SPICE [4] for circuit analysis. EDA did not yet provide support to validate the relationship between the design and simulation.

Today, as we build systems on chip, many verification teams [5] are using Verilog based Hardware Description Language (HDL) [6] with AMS extensions(VerilogAMS) [7] and the recently added user-defined net-type (UDN) feature in SystemVerilog, supporting digital design plus mixed signal modeling, and system level verification. The adoption of UVM [8] provides digital verification a standard approach, potential reuse, and availability of verification IP (intellectual property). Productivity enhancement for the mixed-signal verification team is available for modeling circuit level modeling. There is a tool to support automatic extraction of an AMS behavioral model from a schematic(Arana [9]), one another to construct models from a library of flexible elements, or model transistors in SV(Xmodel [10]) and the one we use to construct a behavioral model and test-bench from a textual specification(Models in Minutes(MiM) [11]).

It is currently possible for a mixed-signal verification team to quickly develop the models for a design, and even validate that the behavior matches the implementation of each block. But productivity suffers when it comes to design integration testing. Passing untested analog system designs up to the SOC team dramatically increases the number of possible issues to debug, impacting both the integration and modeling teams, creating unplanned re-work and delays. The development of intermediate testbenches for the analog parts is one solution. At the scale of over 100 DUT control pins in addition to signal, power, and bias pins, assembling a verification bench by hand is impractically time consuming, especially if UVM is involved.

The objective of our work is to construct a standard testbench for a design under test (DUT), so that the verification team can immediately start to focus on writing tests. The things we assume one can start with are a list of DUT ports (with name and direction at a minimum) and knowledge of the project naming conventions used so that the DUT ports can be classified into useful groups.

First, we discuss our standardized testbench architecture. This includes the handling of analog ports in a UVM environment, the introduction of a flexible standard interface and agent to support these, and the handling of register-based controls for the common case where the register interface and RTL is outside the scope of the test, but sets of register sequences are still a required abstraction.

Next, we show how to use Python [12] and Jinja [13] templates to construct DUT Specific testbenches from the port list and, if it is available, the register map.
Finally, we show how to use UVM to manage the testing with sequences, including sequences that depend on feedback from the design.
To conclude, we will present the cost (development time) and benefit (TB build time difference) when adopting this methodology.

PROPOSED AMS TESTBENCH ARCHITECTURE

For a generic testbench, it is useful to keep the architecture simple and universally applicable, at least for designs without embedded register interfaces as shown in Figure 1**.** An integration level test environment will need to provide significant flexibility in ordering test events, as well as supporting a variety of signal generation approaches. This contrasts with the circuit level model test provided by a tool like MiM [11], where we compare the model with the implementation, with one test that could be as simple as a sweep of all the logic control values with pass/fail determined for each step. UVM provides flexibility in sequences with support for running multiple tests from a single snapshot and standardized messaging and reporting. UVM will likely be used for the testbench at the next level of design integration, allowing for some UVM component reuse from this testbench. Lacking any reasonably standard alternatives, we selected UVM.

Our approach is to classify ports into two high-level categories: static control or status bits, and everything else. To handle these types, we propose using only two UVM components, one dedicated to the control/status (register) interface and the other for generic sequences. In building the testbench we further group the other signals into additional groups, depending on the ease of automating the stimulus. In our case these are static analog signals (power supply and ground and their monitor points if applicable), dynamic analog signals and dynamic digital signals, typically clock and data signals. The register and power blocks are relatively easy to generate automatically, but some dynamic signals types lend themselves to automation as well, i.e. sinusoids and clocks.

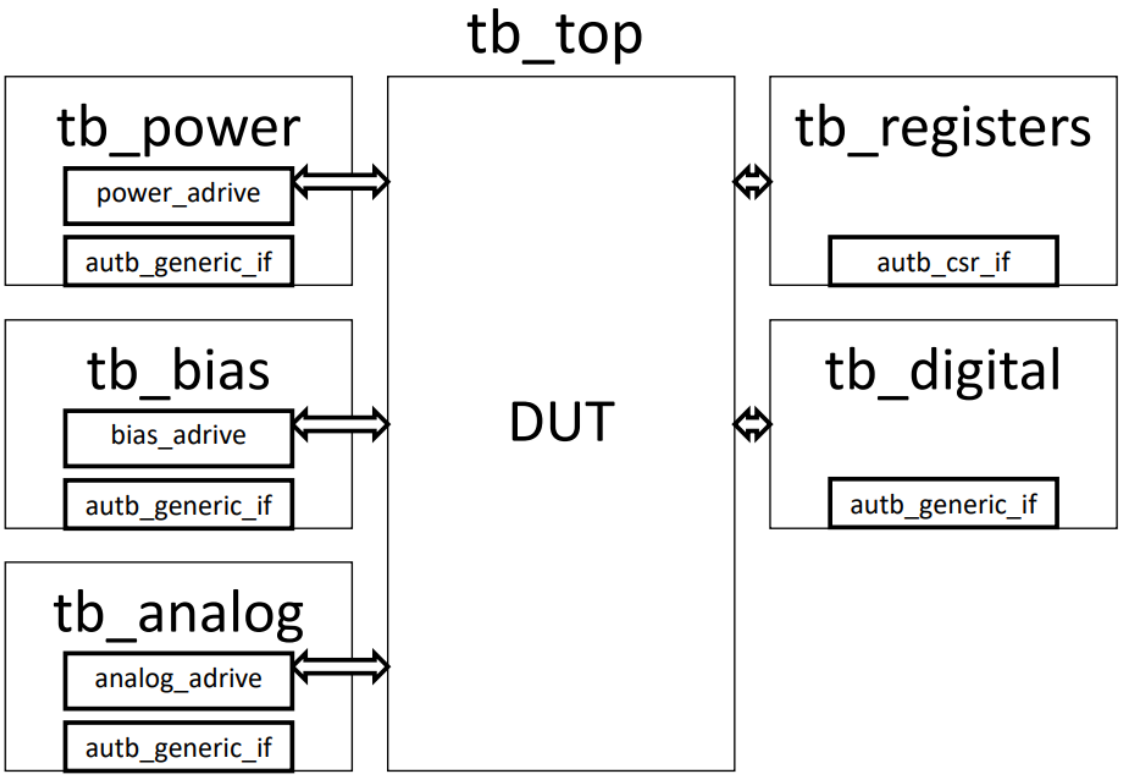

Figure 1
Testbench Architecture

Listing 1
Generic Interface Declaration

```
// interface for the autb_generic agent.
interface autb_generic_if();
    string settings[string], observations[string];
    event sample_trigger;
    string sequence_name, design_state_name;
endinterface
```

```
//interface for the autb_csr agent
interface autb_csr_if();
    int readonly[string], writable[string];
    string sequence_name, design_state_name;
    int delay_us, delay_ns;
    task delay(input int us_delay=1, ns_delay=0);…
endinterface
```

Analog-UVM(AU) generic interface.

In UVM, agent design can be essentially determined from the variables in the interface. For a generic approach, we use an interface that can accommodate any number of variables of any type as shown in the left column of Listing 1. We chose an associative array of string variables indexed by string names (inspired by [14]), as standard functions exist to cast other variables types (*int* and *real*) both to and from strings. The interface must accommodate bi-directional information flow. Two associative arrays are used in the interface, one, *settings*, for control variables sent to the DUT and the other, *observations*, for measurement variables from the DUT sent to the test. Three additional variables are added to the interface for reporting and synchronization. A *status_trigger* event variable enables the agent to request generation of an observation item. The sequence_name string provides the name of the most recent sequence to the design and the *design_state_name* allows some tagging of the observations with design state information. The interface has no clocking, and thus needs no ports.

Analog-UVM generic agent sequence item and driver

The sequence item as shown in **Listing 2** also contains the same key variables as the interface, except those related to data synchronization. The standard key functions are provided , do_copy, do_compare and convert2string, modified

Listing 2
Declaration of autb_generic_seq_item and key functions

```
// sends control values to tb in name, value pairs in a hash  , with values converted to string
class autb_generic_seq_item extends uvm_sequence_item;
    // UVM Factory Registration Macro
    `uvm_object_utils(autb_generic_seq_item)

    //---------------------------------------
    // Data Members - extend and add random vars, after randomization copy into the hash.
    //---------------------------------------
    string settings[string];
    string observations[string];
    string design_state_name;

    // Standard UVM Methods:
    extern function new(string name = "autb_generic_seq_item");
    extern function void do_copy(uvm_object rhs);
    extern function bit do_compare(uvm_object rhs, uvm_comparer comparer);
    extern function string convert2string();
    extern function void do_print(uvm_printer printer);
    extern function void do_record(uvm_recorder recorder);

endclass:autb_generic_seq_item
```

```
function bit autb_generic_seq_item::do_compare(uvm_object rhs, uvm_comparer comparer);
  autb_generic_seq_item rhs_;
  string i;
  bit eq = 1;
  if(!$cast(rhs_, rhs)) begin
    `uvm_error("do_copy", "cast of rhs object failed")
    return 0;
  end

  eq &=  super.do_compare(rhs, comparer);
  foreach (rhs_.settings[i])
      eq &= settings[i] == rhs_.settings[i];
  foreach (rhs_.observations[i])
      eq &= observations[i] == rhs_.observations[i];
  eq &= design_state_name == rhs_.design_state_name;

  return eq;
endfunction:do_compare
```

Listing 3
Task declaration for autb_generic_driver run_phase showing sequence item handling.

```
42  task autb_generic_driver::run_phase(uvm_phase phase);
43    autb_generic_seq_item req;
44    autb_generic_seq_item rsp;
45    int psel_index;
46    string setting;
47    forever
48      begin
49        seq_item_port.get_next_item(req);
50        `uvm_info($sformatf("%s_DRIVER",
51          this.get_full_name().toupper()),
52          $sformatf("Starting sequence %s",
53          req.get_name() ),UVM_LOW)
54        foreach (req.settings[setting])
55          m_vif.settings[setting] = req.settings[setting];
56        m_vif.sequence_name = req.get_name();
57        seq_item_port.item_done();
58      end
59  endtask: run_phase
```

from the cookbook examples [15]. This is designed to be extended in the test environment based on the actual variables that are needed in each testbench module. Randomization features are not used at this time.

The agent driver copies the information from the sequence item into the interface as shown in Listing 3. The remaining agent code can be found in the locations in the Appendix.

AU register interface.

The primary difference between the generic interface and the register interface as shown in the right column of Listing 1, is that we know that we can use a single type (*int*) for all the settings and observation values. For the register interface we declare two associative arrays, "writable" and "readonly," again with string index, but of type "int." The "sequence_name" and "design_state_name" variables provide the same function as in the generic interface. In addition, we provide a "delay" function, to allow each sequence to model any required time consumption for the time it might take to write more than one register at a time thru a register interface bus. To avoid confusion with other standard packages we chose the name *autb_csr_if* (for analog UVM testbench control/status registers.)

AU csr agent

The csr agent differs from the generic agent only in that the associative array is a "int" type indexed by string rather than string type. The code can be found at the location in Appendix A.

Generic testbench block architecture

We will use the power section as our example for analog stimulus. The *tb_power* block contains two instances and code to transfer information between the two, as shown in Table 1.

The adrive block for DC values uses a real valued "set" variable and an "enable" signal. In the Verilog-AMS case, there are also variables for *Ron/Roff* and transition time, so these are also present in the SV-RNM model even though not used. The listing for both versions of the *adrive*, the SystemVerilog wrapper and the interface are shown below.

TABLE 1
TESTBENCH COMPONENT ROLES

Block	top_tb	tb_power_stim tb_bias_stim tb_analog_stim	tb_register_stim tb_digital_stip
Code function	import test package and start uvm test	connect interface variables and [power | bias | analog]_adrive variables	variable declarations and port assignments
			connect interface variables and variable values

Listing 4
Verilog-AMS view of power_adrive module

```
//Verilog-AMS HDL for "simple_uvm_testcase_stim_power_adrive" "verilogams"
`include "constants.vams"
`include "disciplines.vams"
module simple_uvm_testcase_stim_power_adrive (
    output  AVDD1P8 ,   output          AVSS  );
    electrical   AVDD1P8,    AVSS;
    reg sample_trigger = 0;     real AVDD1P8_vset;
    integer AVDD1P8_enable;     real AVDD1P8_roff = 10K;
    real AVDD1P8_ron = 10;     real AVDD1P8_tt = 1n;
    real AVDD1P8_rratio;    electrical AVDD1P8_vdrive;
    real AVDD1P8_rout;
    always @(posedge sample_trigger) begin //copy monitored voltages to obserations
    end
    analog begin
        AVDD1P8_rratio = transition(AVDD1P8_ron/AVDD1P8_roff,0,AVDD1P8_tt,AVDD1P8_tt);
        AVDD1P8_rout = AVDD1P8_roff * pow( AVDD1P8_rratio,
                        transition(AVDD1P8_enable,0,AVDD1P8_tt,AVDD1P8_tt) );
        V(AVDD1P8_vdrive) <+ transition(AVDD1P8_vset,0,AVDD1P8_tt,AVDD1P8_tt);
        I(AVDD1P8,AVDD1P8_vdrive) <+ V(AVDD1P8,AVDD1P8_vdrive)/AVDD1P8_rout;
        V(AVSS) <+ 0;
    end
endmodule
```

The *tb_analog* block may require additional types of sources, to support dynamic waveforms. Two options that can be easily automated are sinusoidal sources, and reading wave data from a file. In addition, similar monitors for outputs may be needed. Our simple example case only needs observation of static voltages. Power and bias modules would default to using DC value observation.

The generic agent interface provides the connection to UVM and the test. The agent and interface are designed not to require changes for each design. Associative arrays provide a convenient way to provide the required flexibility. Separate associative arrays are provided for stimulus and observation. A sample trigger is provided to allow control of observation data collection in the absence of other triggers.

The power, bias and analog *adrive* modules may be coded in SystemVerilog or Verilog-AMS, depending on the DUT representation to be used. All ports of the driver are connected through the block stimulus top as interconnect for flexibility. The signal type is established in the *adrive* module. For Verilog-AMS these are declared as electrical as shown in Listing 4.

Voltage sources are declared with a second electrical node to establish a resistive branch. In the SV-RNM case, shown in Listing 5, we use a single “discrete electrical” UDN which resolves node voltage based on any number of Thevenin or Norton equivalent drivers. For ease of modeling in our selected tool, MiM, these are provided as a *DE_thevenin* and *DE_norton* module which are instantiated on each (discrete electrical) port of our models. These

Listing 5
SystemVerilog-RNM view of power_adrive module

```
//SystemVerilog RNM HDL for "simple_uvm_testcase_stim_power_adrive" "svrnm"
module simple_uvm_testcase_stim_power_adrive (  output interconnect  AVDD1P8,
    output interconnect     AVSS );
    real AVDD1P8$Vobs, AVDD1P8$Iobs, AVDD1P8$Vdrv, AVDD1P8$Rdrv;
    real AVSS$Vobs, AVSS$Iobs, AVSS$Vdrv, AVSS$Rdrv;
    DE_thevenin Xtcvr_AVDD1P8( AVDD1P8,AVDD1P8$Vobs, AVDD1P8$Iobs, AVDD1P8$Vdrv, AVDD1P8$Rdrv);
    DE_thevenin Xtcvr_AVSS( AVSS,AVSS$Vobs, AVSS$Iobs, AVSS$Vdrv, AVSS$Rdrv);
    reg sample_trigger = 0; // only used in the vams implementation
    real AVDD1P8_vset;              bit AVDD1P8_enable;
    real AVDD1P8_roff = 10e3;       real AVDD1P8_tt = 1e-6;
    real AVDD1P8_ron = 0.1;
    always_comb
        if (AVDD1P8_enable) begin
            AVDD1P8$Vdrv = AVDD1P8_vset;    AVDD1P8$Rdrv = AVDD1P8_ron;
        end else begin
            AVDD1P8$Vdrv = 0.0;             AVDD1P8$Rdrv = AVDD1P8_roff;
        end
    initial begin
        AVSS$Vdrv = 0.0;
end
endmodule
```

Listing 6
SystemVerilog view of tb_power block

```
//SystemVerilog HDL for "simple_uvm_testcase_stim_power" "svlog"
module simple_uvm_testcase_stim_power (
    output interconnect  AVDD1P8,   output interconnect      AVSS );
import uvm_pkg::uvm_config_db;
autb_generic_if pwr_if(); // contains an associative array of string with string index settings
initial begin
    uvm_config_db#(virtual autb_generic_if)::set(null, "uvm_test_top", "power_vif", pwr_if);
end
simple_uvm_testcase_stim_power_adrive power_adrive(
    .AVDD1P8(AVDD1P8),  .AVSS(AVSS) );
bit sample_trigger;
always @(pwr_if.settings["trigger"] or pwr_if.sample_trigger) begin
    sample_trigger = pwr_if.settings["trigger"].atoi();
    if (sample_trigger ) begin
        power_adrive.sample_trigger = 1;
        #1ps power_adrive.sample_trigger = 0;
    end
end
    always @(pwr_if.settings["AVDD1P8_vset"])
        power_adrive.AVDD1P8_vset =  pwr_if.settings["AVDD1P8_vset"].atoreal();
    always @(pwr_if.settings["AVDD1P8_enable"])
        power_adrive.AVDD1P8_enable =  pwr_if.settings["AVDD1P8_enable"].atoi();
    always @(pwr_if.settings["AVDD1P8_ron"])
        power_adrive.AVDD1P8_ron =  pwr_if.settings["AVDD1P8_ron"].atoreal();
    always @(pwr_if.settings["AVDD1P8_roff"])
        power_adrive.AVDD1P8_roff =  pwr_if.settings["AVDD1P8_roff"].atoreal();
    always @(pwr_if.settings["AVDD1P8_tt"])
        power_adrive.AVDD1P8_tt =  pwr_if.settings["AVDD1P8_tt"].atoreal();
endmodule
```

each have five connections as shown in Table 2, for brevity in module code these are commonly connected by port order.

TABLE 2
UDN TRANSACTOR MODULES WITH PORT-TYPE INFORMATION.

Type	DE_thevenin	DE_norton	Dir	Description
UDN	Signal	signal	IO	the node connection
Real	Vobs	Vobs	O	the node voltage as resolved
Real	Iobs	Iobs	O	the branch current as resolved
Real	Vdrv	Idrv	I	the branch quantity driven
Real	Rdrv	Gdrv	I	the branch qualifier value driven

Power output behavior is modeled with a voltage setting and an enable. If the enable is set the voltage is driven to the values of *vset*, else it is driven to zero also, in the VAMS view only, a high resistance is used in the off state. Grounds are always driven to zero. Two variables in the adrive block are set from the sequence in the power block for static analog signals, per output port, *{{port}}_vset* and *{{port}}_enable* as shown in Listing 6.

We could generate the interface and matching sequence item with a real variable and bit per port but that would require a new interface and agent for every testbench. We could simplify and have an associative array of reals for the *vset* and another associative array of bit for the enable, but does not allow for additional types of variables to be added and used after the TB and sequences are generated. By using a single associative array of strings with string lookup we can pass a whole table of variables per block. Note that this pattern serves very well in the *tb_analog* block where we do not really know ahead of time what kinds of information need to be set from the test. Thankfully, SystemVerilog has string functions to convert both reals and integers to and from strings. These are used in the extended sequences and in the tb_power wrapper block as shown in Listing 6.

Generic testbench top architecture

The tb_top is now simply a portless module, connecting the DUT to each of the stimulus modules, plus the UVM package and the run_test command.

Argument for automation

The UVM testbench for our simple example requires 41 files, across 9 directories. 8 packages are compiled. Our production environment for a larger design creates 74 files. Without automation, this is not manageable for rapid testbench creation and deployment. It may need to be done once, but such an exercise should not be repeated without good reason. With a little automation, this can be accomplished for a new design in a few minutes, if the data is ready and the flow is set up. The data flow for this process is shown in Figure 2. The design specification for the following examples is shown in Listing 7.

Example Jinja Templates and Python rendering code

The tb_power_stim block testbench was rendered with the power block dictionary from the final processed testbench specification for our simple example. The module declaration needs to include all ports, and then are the two instances to place. The template code for this part of the module is shown in Listing 8. If bus port ranges are not included in the port part of the module declaration, they do need to be declared before the instances are added. Double braces, "{{ }}," indicate insertion points for data elements, and single braces with percent "{% %}" indicate instructions for the template engine. The resulting module declaration was shown in Listing 6.

A similar approach was used to generate the code blocks that respond to the changes in the settings in the interface to set the variables in the *adrive* block, as well as collect observed variable values from the *adrive* block and set the entries in the *pwr_observations* array. As well as the entire *adrive* block in both SV-RNM and Verilog-AMS flavors.

Listing 7
Nestedtext representation of example design specification, after applying Name Convention and Register map information.

```
library: RFDV_scratch
name: simple_uvm_testcase
ports:
    AVDD1P8:
        direction: input
        porttype: power
        supply_nom: 1.8
        supply_type: volts
    AVSS:
        direction: input
        porttype: power
        supply_type: ground
    bg_enable:
        direction: input
        porttype: register
        reg_info:
            name: bg_enable
            blockid: top
            width: 1
            default: 'b0
            on_value: 'b1
            description: bias_blk enable
    clk_en:
        direction: input
        lsb: 0
        msb: 3
        porttype: register
        reg_info: ...
    clk_out_0: ...
    clk_out_1: ...
    clk_out_2: ...
    clk_out_3:
        direction: output
        porttype: digital
        digital_type: clock
    dll_lock_status: ...
    mclk_in: ...
    mode_sel_0: ...
    mode_sel_1: ...
    mode_sel_2: ...
    mode_sel_3: ...
    vcntrl0:
        direction: output
        porttype: analog
        analog_type: dc
    vcntrl1: ...
    vcntrl2: ...
    vcntrl3: ...
tests:
    sanity_test:
        sequences:
            top_on: register
        body: #100us;
```

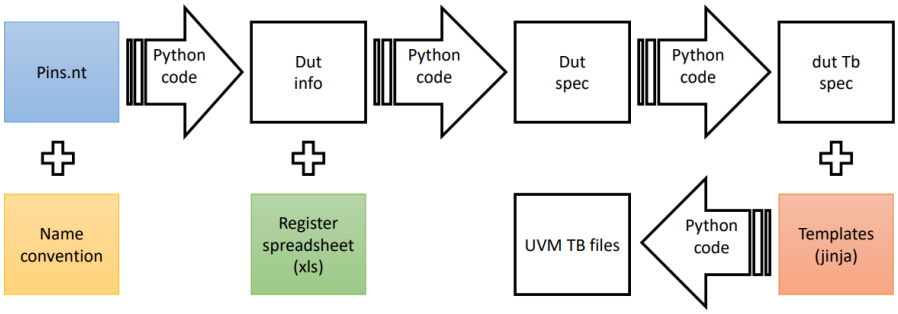

Figure 2
Data flow diagram for TB file generation.

Listing 8
Jinja Template for module declaraton and instances placed in tb_power module

```
1  //SystemVerilog HDL for "{{library_name}}", "{{name}}" "svlog"
2  module {{ name }} ( {% set comma=joiner(', ') %}{% for port,portitem in ports.items()  %}{{ comma() }}
3      {{portitem['dir']}} interconnect {%if portitem['msb'] %}[{{portitem['msb']}}:{{portitem['lsb']}}]
4  {%- else %}     {%endif%}   {{port}} {% endfor %}
5  );
6  import uvm_pkg::uvm_config_db;
7  autb_generic_if pwr_if(); // interface contains an associative array of string with string index settings
8  initial begin
9      uvm_config_db#(virtual autb_generic_if)::set(null, "uvm_test_top", "power_vif", pwr_if);
10 end
11 {{name}}_adrive power_adrive({% set comma=joiner(', ') %}{% for port,portitem in ports.items()  %}{{ comma() }}
12     .{{port}}({{port}}){%endfor%});
13 bit sample_trigger;
```

Example templates and python code (run as code snippets in a Jupyter notebook) and the Anaconda package specifications can be found at the locations in Appendix A.

FINISHING THE UVM BENCH.

Based on the NestedText design specification captured from the DUT portlist, project naming convention and register_map, the full testbench specification was built and all the testbench and compilation files were generated as shown in the diagram in Figure 2.

The UVM environment and initial sanity test are sufficient to validate design and tb elaboration and to see the power, bias and register defaults are applied. At this point standard verification work can begin. Turn on sequences for blocks generated from the register information can be added to the test. But there may be additional code needed in some tb blocks as discussed below. At this point, one can accept the script work as finished and work directly with the generated files. Alternatively one could continue to add features to the data-structure and templates and re-generate the testbench. A serious look at the Portable Stimulus Standard (PSS) [16] may help to inform an approach here.

Custom Analog signal generation (other than power and bias, i.e.: Sinusoids.)

For the analog block, one common requirement is driving inputs with sinusoidal signals, another is monitoring sinusoidal outputs. Once initially developed, this is probably most easily addressed by enhancing the data structure and the template, as one could easily follow a single pattern for all such cases. The simple test case we use to present the paper only needs to monitor the DC value of analog outputs.

Digital signal generation (other than register/logic controls, i.e.: Clocks)

Likewise in the digital block, a common requirement is to generate some clocks and monitor clock frequencies from the DUT. These will be relatively easily added to the template as well.

Writing Sequences to control the simulation.

Beyond simple “block turn on” sequences, (such as is sufficient for the example design here) each design will require design specific sequences that are not easily inferred from the design ports or the register map. As the sequence libraries contain at least one example sequence each, developing more will not be difficult for the verifier.

Controlling sequences based on design outputs.

While not yet enabled in the automated version of this flow, the first design utilizing this flow had such a requirement. It was a phase-locked loop (PLL) which utilized a binary search algorithm to set the voltage controlled oscillator (VCO) coarse frequency control bits. To enable this an observer interface was instantiated in the design (normally this should be bound to a design element, with a function to test if the value of the required variable was above or below the target value. The virtualized interface was made available to the environment and passed to the virtual sequence that was configuring the PLL. This pattern is very similar to that used to query the state of a DUT interrupt pin for interrupt testing. An example from the Verification Academy Cookbook [15] was referenced.

RESULTS

As J.B.D. had some prior experience with Jinja and Python, the methodology seemed likely to produce a useful future result. The extra development effort paid off, as only a small effort was needed on the third deployment.

Initial development

For the initial testbench development, approximately three weeks were required from start of the TB development until initial simulation was successful. A couple of days were the initial UVM environment debugging until the simulation completed build and connect phase and started run phase. Another week was needed to finish the bench (adding the observer interface and additional test sequences) and a 5th week working with the design team to resolve issues until the full design showed the functionality expected.

Subsequent Deployment Experience

A common experience with the second use of a self-built tool, is that many things in the initial development are less than optimal. One area in this case was in applying the naming convention to the design. After a couple of attempts to get this re-coded and working, we realized there was a way to provide a general search functionality with the details of the naming convention provided as a data structure in a *nestedtext* file. This allows separation of the details of naming convention from the Python code, enabling a different convention to be applied as needed to get the tool to do most of the sorting work. This refactoring added a week. A couple of simulator issues with the models as coded added a week of debug not related to this flow development. Small changes to the TB spec structure required edits to most templates with the result, again, that it took 3 weeks to get the initial version of the second bench up and running. That simulator issue required a test-case. A simple UVM test case (now the example design featured in this paper) was thrown together. The 3 scripts were copied to a new tb workspace, applied to the new design. The initial UVM bench was up and running the test through powerup in only 3 hours, after fixing a couple more template issues. Figure shows the doubler dll locking and dll loop voltage output voltage changing in our example circuit.

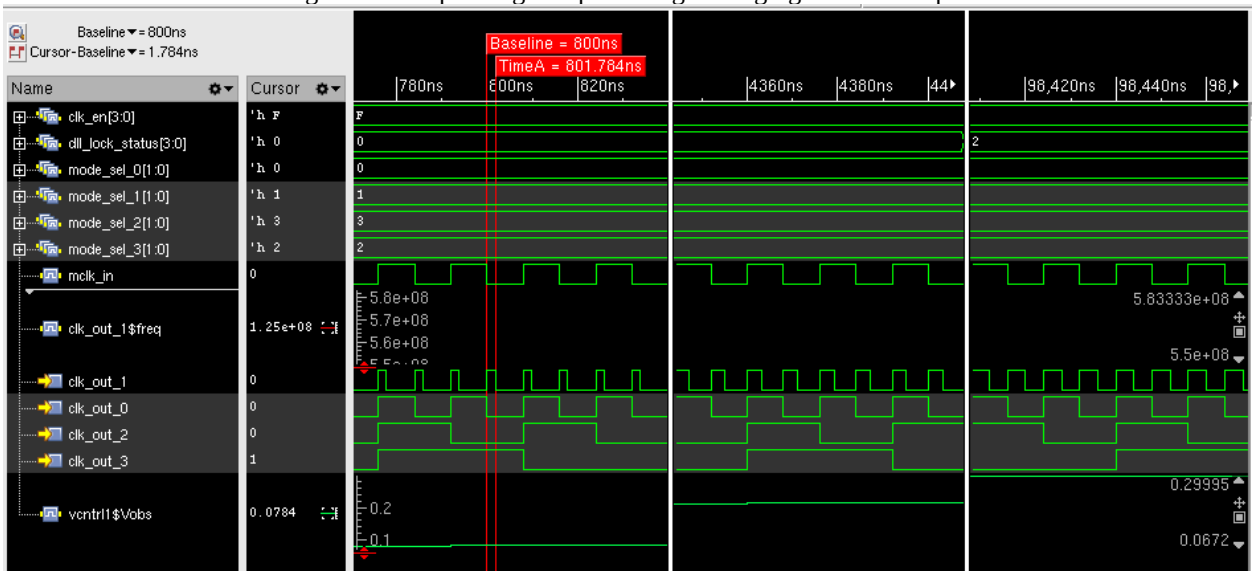

Figure 3 Simulation Results for the Example circuit with turn-on sequence added.

On Naming conventions

Naming conventions are nice until someone needs to enforce them. Then documenting them is essential, and this practically requires one to select a parser engine and parser language definition scheme for the purpose. Names that include *Ibias*, *avdd*, *avss* and *vldo* may seem easily identified as bias and supply types. But the blocks generating them may have controls that include the item being controlled (*en_ibias3*, *ldo_vsel*, *en_ldo_avdd1p3*. In this case it seems that the phrases indicating a control signal, take priority over the identification as a supply. This insight leads to the two level search algorithm used so far, but this author suspects that other approaches may still be needed before giving mixed-signal development teams advice on choosing a naming convention and enforcing it.

CONCLUSION

Combining templating plus a flexible programming language in an interactive editing environment, mixed-signal verification teams can easily solve the testbench creation problem and get some benefit from the existing UVM standard.

APPENDIX

Agent and template Code examples are published at https://github.com/jbdavid-inno/analog-uvm-tb .
Example DUT with generated files are published at https://github.com/jbdavid-inno/simple_uvm_testcase .

REFERENCES

[1] H. A. Mantooth and M. Vlach, "Beyond SPICE with Saber and MAST," in *[Proceedings] 1992 IEEE International Symposium on Circuits and Systems*, 1992.

[2] J. Vlach, K. Singhal and M. Vlach, "Computer oriented formulation of equations and analysis of switched-capacitor networks," *IEEE Transactions on Circuits and Systems*, vol. 31, no. 9, pp. 753-765, September 1984.

[3] J. D. Burroughs and R. A. Hammond, "Control Analysis and Design Features of EASY5," in *1983 American Control Conference*, San Francisco, CA USE, 1983.

[4] C. C. McAndrew and L. W. Nagel, "SPICE Early modeling [bipolar transistors],," in *Proceedings of IEEE Bipolar/BiCMOS Circuits and Technology Meeting*, Minneapolis, MN, USA, 1994.

[5] K. Kundert and H. Chang, "Verification of Complex Analog Integrated Circuits," IEEE Custom Integrated Circuits Conference 2006, San Jose, CA, USA, 2006, pp. 177-184, doi: 10.1109/CICC.2006.320883.," in *IEEE Custom Integrated Circuits Conference 2006*, San Jose, CA, USA, 2006, 2006.

[6] "IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language," *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pp. 1-1315, 22 Feb 2018.

[7] "Verilog-AMS Language Reference Manual (VAMS-2023 Draft)," 03 Nov 2023. [Online]. Available: https://accellera.org/images/downloads/drafts-review/Verilog-AMS_2023_Draft.pdf. [Accessed 10 Nov 2023].

[8] "IEEE Standard for Universal Verification Methodology Language Reference Manual," *IEEE Std 1800.2-2020 (Revision of IEEE Std 1800.2-2017*, pp. pp 1-458, 14 Sept 2020.

[9] Orora Design Technologies, Inc., "Arana Behavioral Modeling Platform," [Online]. Available: https://orora.com/index.php?page=products-arana. [Accessed 13 Nov 2023].

[10] Scientific Analog, Inc., "xmodel: Empower SystemVerilog with Event-Driven Analog Models," 2023. [Online]. Available: https://www.scianalog.com/xmodel/. [Accessed 13 Nov 2023].

[11] Designer's Guide Consulting, Inc., "Analog Verification Products: Models in Minutes," [Online]. Available: https://designers-guide.com/main/products/. [Accessed 13 Nov 2023].

[12] Python Software Foundation, "Python - About," 2023. [Online]. Available: https://www.python.org/about/. [Accessed Nov 2023].

[13] Pallets, "Jinja - Introduction," 2007. [Online]. Available: https://jinja.palletsprojects.com/en/3.1.x/intro/. [Accessed 13 Nov 2023].

[14] R. Edelman, "No RTL Yet? No Problem UVM Testing a SystemVerilog Fabric Model," in *DVcon Proceedings*, San Jose, CA, USA, 2016.

[15] Mentor Graphics' Verification Methodology Team (Siemens Inc.), "Stimulus/Signal_Wait," 2023. [Online]. Available: https://verificationacademy.com/cookbook/stimulus/signal-wait. [Accessed 13 Nov 2023].

[16] Accelera Systems Initiative, Portable Test and Stimulus Standard Version 2.1, : Accellara Systems Initiati, 2023.

[17] Cadence Design Systems, "Cadence SKILL Language User Guide - Product Version IC23.1," September 2023. [Online]. Available: https://support.cadence.com. [Accessed 13 Nov 2023].

[18] YAML Development Team, "YAML: YAML Ain't Markup Language™," 1 Oct 2021. [Online]. Available: https://yaml.org/. [Accessed 13 Nov 2023].

[19] Anaconda Inc., "We're not just a company; we're a movement.," 2023. [Online]. Available: https://www.anaconda.com/about-us. [Accessed 2023].

[20] Anaconda Inc., "Conda," 2017. [Online]. Available: https://docs.conda.io/en/latest/. [Accessed 13 Nov 2023].

[21] Microsoft, "Visual Studio Code," 2023. [Online]. Available: https://code.visualstudio.com/. [Accessed 13 Nov 2023].

[22] K. Kundert and K. Kundert, "NestedText — A Human Friendly Data Format," 30 May 2023. [Online]. Available: https://nestedtext.org/en/latest/. [Accessed 13 Nov 2023].