2015 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Comprehensive AMS Verification using Octave, Real Number Modelling and UVM

John McGrath, Xilinx, Cork, Ireland (john.mcgrath@xilinx.com)

Patrick Lynch, Xilinx, Dublin, Ireland (patrick.lynch@xilinx.com)

Ali Boumaalif, Xilinx, Cork, Ireland (ali.boumaalif@xilinx.com)

***Abstract*— Modern Analog-Mixed-Signal designs require comprehensive verification of both analog functionality and digital-analog-interaction. This paper outlines some new techniques to simplify and automate analog real-number modelling, as well as maximizing re-use of models both in System-Verilog and Matlab®. It also outlines how these techniques have been integrated into a UVM environment to enhance analog and full-system verification with randomization of analog stimulus, and analog functional coverage.**

Keywords—AMS Verification; Real Number Model; SystemVerilog; UVM; Netlist based RNM; Matlab; Octave

I. INTRODUCTION

Mixed signal design complexity has increased dramatically in recent years. A number of factors have led to this – increasing integration of analog functionality in SoCs, digitally-assisted-analog for performance/cost/power optimization, and advances in analog architectures to take advantage of the latest process nodes. This integration has led to tightly coupled interaction between analog and digital which requires comprehensive verification, while at the same time delivering fast run-time, low-overhead and above-all functional accuracy. Real-Number-Modelling (RNM) via SystemVerilog (SV) is beneficial when it comes to analog modelling – though the cost of model generation can be high, and checking the model and DUT performance and functionality is still required.

This paper outlines 3 techniques we have used to address these challenges:

- *Automated Netlist-based RNM to facilitate model generation and re-use.*
- *SV-Octave interface to facilitate results checking, predictor-model writing, and analog stimulus generation.*
- *UVM based environment to combine these components together and provide randomization, coverage and results scoring of both digital and analog functionality [1].*

A key advantage of all the approaches outlined here is that they are completely vendor neutral, allowing the approaches to be applied independently of the simulator being used.

II. NETLIST BASED REAL NUMBER MODELLING[2]

Real number modelling is a method of writing behavioural models of analog circuits using SystemVerilog. The main enabling feature for RNM is the ability to define the ‘real’ data-type as a port on a module. Previously where ports were limited to a values of 1, 0, X or Z, now a signal representing any discrete analog value can be used, and passed between modules. There are a few limitations when using the real type, and this will be explored in detail in this section.

A. Breaking Down Design Complexity

It is commonly the case that complex systems are composed of a number of interconnected 'building blocks'. This is true in general for analog designs - where even very complex circuits can be reduced to a combination of simpler components. Taking advantage of this fact is a key enabler for reducing the complexity of real-number model generation. Rather than model complex blocks at a high-level and risk mistakes, instead models should be built from a hierarchy of much simpler models. The ideal situation is one where the sub-models are easy to write, while still behaving exactly the same as the real analog circuit overall. This also enables re-use across designs and projects as many ‘building-block’ models can be made generic, or can be tweaked for use a new application.

B. *Schematics are Golden*

Analog designs are largely schematic driven - and it is the schematic that is used as the ultimate reference for the design that is to be taped-out. The schematic also contains a lot of information purely due to its structure - it brings together multiple analog-sub blocks, describes signal-flow, and includes custom control logic. Given this, it makes sense that as much information as possible should be extracted (re-used) from the existing schematics when generating a RNM of the analog system. For example, all custom-digital logic in the analog hierarchy should be used directly in the RNM, and only low-level analog blocks should be modelled. This allows re-use of the majority of the analog schematic hierarchy and hook-up - and reduces the number and complexity of items to be modelled dramatically. We refer to this as the netlist-based approach to RNM generation, and as will be shown much of this model generation can be automated.

For example: As a test-case we targeted a complex analog-switch matrix which consisted of >50 input sources, numerous different multiplexer types, switch-cells and decoding/control logic. This was completely modelled by using the netlist-based approach. This model only required the writing of a single core switch-cell RNM in SystemVerilog, and the rest of the model was auto-generated from the schematic netlist. Any subsequent updates to the schematic to add new inputs or modify the hierarchy simply required re-netlisting to re-generate the model with the same core switch-cell.

Results: This test-case (partially illustrated in Figure II-1) also found a number of analog functional issues in the initial analog schematic hook-up, including contention, polarity inversion and mismatches vs the spec. This illustrates another key benefit to the netlist-based RNM approach – not only does it provide accurate analog models for digital verification, but it also enables coverage of the analog functionally throughout the design phase.

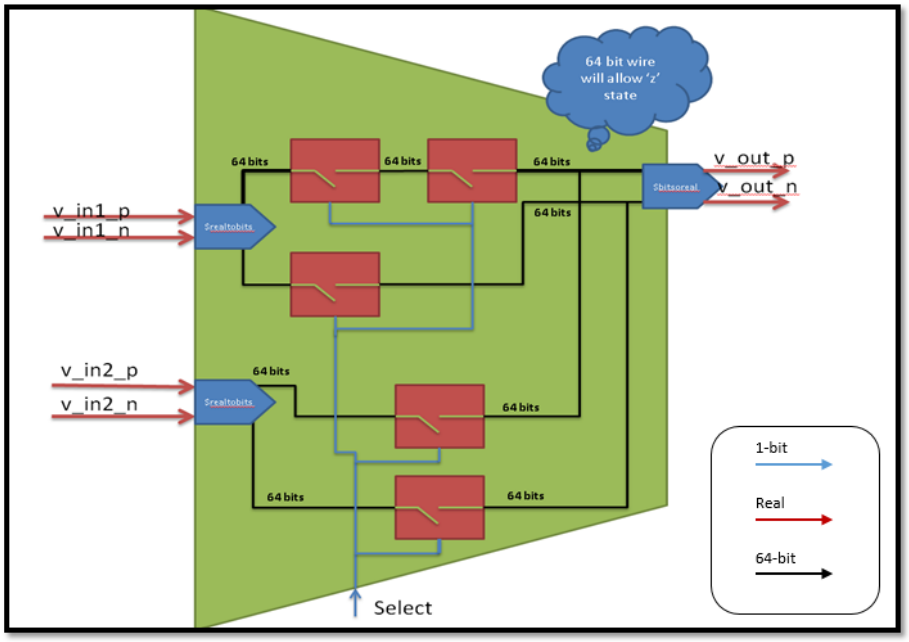

Figure II-1: Example 2:1 Netlisted multiplexer RNM Model

C. *Guidelines to Enable Automatic Netlist-Based RNM Generation*

While not required - in order to take the best advantage of the netlist-based approach, analog schematics should ideally be structured to partition the low-level analog functionality from the digital/control logic associated with it. Similarly, the design hierarchy should be careful to not expose any 'bare transistors/passives' at a higher level of the hierarchy, as this could prevent efficient automation of the model generation. The tweaks required to existing analog schematics to follow this recommended structure are usually small, and local. E.g. simply wrapping a purely analog function in a level of hierarchy is usually sufficient to enable the automatic RNM generation.

The high-level schematics guidelines are:

- *Take advantage of the netlist where possible - model at the lowest feasible level of analog sub-block.*
- *Partition digital logic from analog functionality - even locally.*
- *Use signal-flow – 'inouts' are to be avoided on RNM signals where possible.*
- *Encapsulate analog-feedback within the model where possible.*
    - *E.g. an amplifier gain-stage should only expose the input and output nodes.*

These guidelines have other side benefits when followed. - This up-front design partitioning allows for flexibility of mixing various views of the design to enable other verification tasks, such as schematic <-> Verilog co-simulation, or schematic <-> Verilog-A simulations.

D. Methodology

Combining what has been discussed in sections A. B. and C. into a methodology, we used a script to automatically convert the standard schematic netlist into a fully-functional RNM. The methodology is described as follows:

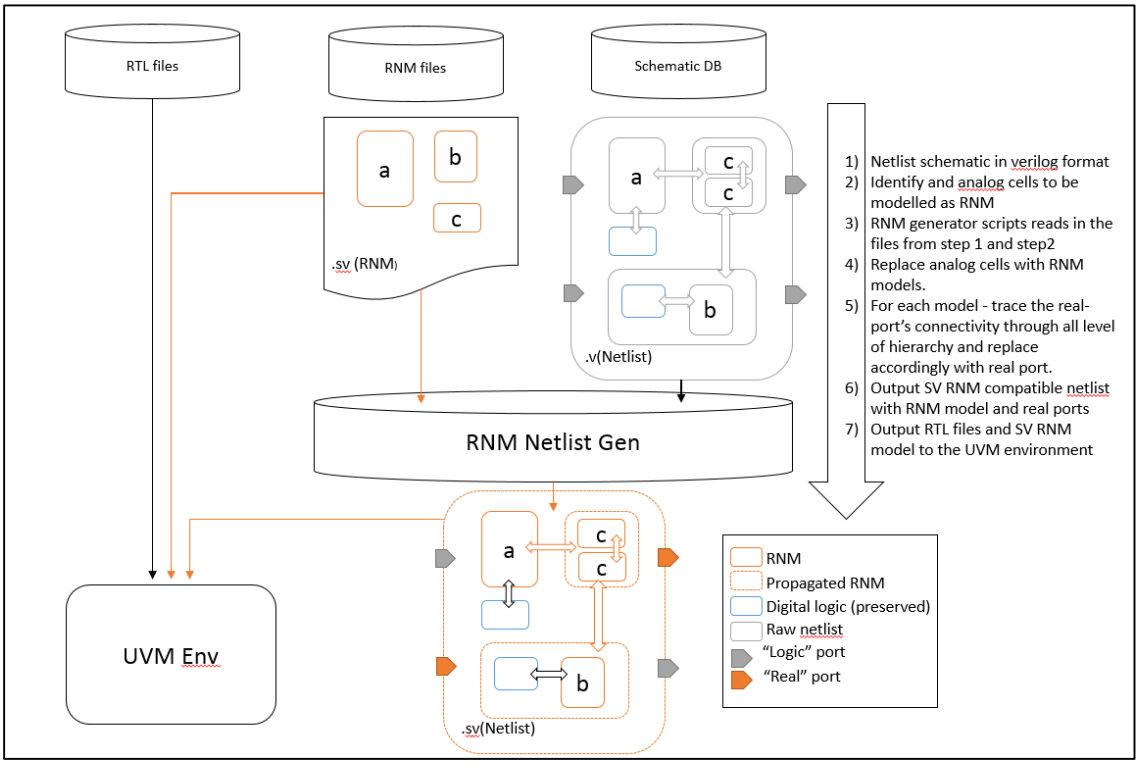

Figure II-2: Automatic RNM Generation Methodology

After this process, the RNM outputted consists of a modified netlist with all real-signals propagated to all analog sub-blocks. If the analog schematics change during the design phase, the process can be re-run. As the flow is automated, this leads to very low overhead model updates with the additional benefit of maintaining model functional integrity with the analog design. A key advantage of this method over some proprietary methods is that the modified output netlist is available for inspection and to aid in debug. This can be a powerful tool when modelling complex systems and interactions.

E. Model Validation

The maintenance and validation of the RNMs of the lower-level analog sub-blocks (e.g. comparators, amplifiers, etc.) is critical to ensure the system is accurate modelled, and can be achieved by a number of methods:

1) In-system co-simulation where some or all RNMs can be replaced by their spice/spectre equivalents
2) Side-by-Side co-simulation where common stimulus is provided to the RNM and spice/spectre block and the output compared
3) The use of EDA tools for model comparison [4].

However, the key point here is that by modelling at the lowest level feasible, the maintenance requirements of these models is very low. Re-using these low-level models for other designs also allows for re-use of the model verification, which is an added benefit.

F. Handling Contention

The SV real data-type has some limitations when modelling analog designs - the most prominent of which is that ports must be input or output – inouts and multiple drivers are not allowed. As a result, X or Z states cannot be modelled directly. The *nettype* feature which removes some of these limitations is discussed in the next section - however, here we outline an alternative that proved to be very powerful and only required the conventional types to be used.

As previously mentioned, where possible the schematics should always use the correct signal flow via input and output ports on blocks. However, in certain cases, as in the analog-multiplexer example Figure II-1, there are legitimate multiple drivers on nets, and inouts must be used in the schematic. To model this behaviour in SV – the script auto-detects these situations and in the case of multiple drivers, it converts the reals to 64-bit 'logic' values using *$realtobits()*. This allowed the 'logic' data-types to resolve any contention, and then the result is converted the result back to reals using *$bitstoreal()*.

G. Nettype

SystemVerilog-2012 [4] introduced a number of features enable RNMs, one of the most useful being *nettype*, which has the ability to allow a user to define resolution functions. This is an advantage over the method of handling contention outlined in the previous section. However, it was observed at the time of writing to have inconsistent support across vendors, and in some cases required extra licenses. The license cost is a particular burden, as a key advantage of RNMs is that they can be incorporated into large regression runs and any extra licenses required for them will limit the RNMs utility. A final limitation was the incompatibility of nettype with certain co-simulation flows. These issues are not inherent to the specification, so can be resolved by EDA vendors. However, the limitations precluded its use in our environment at this time. Once broadly adopted and supported by vendors, both nettype and the type-less '*interconnect*' keyword should enhance the ability to use netlists directly without transformation, which will be very welcome for AMS verification.

H. Summary of Examples

Table II-1 illustrates the results achieved in terms of numbers/lines of unique RNM code written by following this netlist-based methodology versus to number of modules that were auto-generated to build up the total number of modules in the analog block being modelled. As is shown, there are significant reductions in the effort required to model these systems. It should be noted that some of the RNMs written had only small differences from others – so with better schematic partitioning the numbers shown could be improved further.

Table II-1: Netlist-based RNM Productivity Improvement

Test Case	**RNM Statistics**			
	#Modules (Written / Auto-Gen / Total)	***% Modules Written***	***#Lines Written / Auto-Gen / Total***	***% Lines Written***
Analog Multiplexer (example)	7 / 61 / 68	10%	204 / 4900 / 5104	4%
ADC with Sensors	41 / 412 / 453	9%	1418 / 27587 / 29005	4.8%

III. INTEGRATING SYSTEMVERILOG WITH MATLAB/OCTAVE

Modern analog designs and the higher-level systems they are integrated into often begin by first being modelled in Matlab®. This high-level Matlab modelling environment also commonly provides both stimulus generation as well as analysis routines for the analog results. For example, an ADC system being designed in Matlab could consist of a model of the ADC and any associated calibration, plus signal-processing of the data. As a lot of effort goes into developing these system reference models - is it natural to desire maximal re-use of them during all stages of verification. Traditional methods of doing this would be to dump vectors from Verilog-simulations and do offline comparisons between these vectors and the Matlab result. However, for modern systems which may include dynamic calibration and analog-digital feedback, a vector based approach may be cumbersome. This is further compounded when trying to verify these systems in a constrained-random UVM environment, where thousands of scenarios may be required to be run in order to hit coverage goals. Generating and managing these 'vector-dumps' and their results can become a large overhead.

Instead, we propose a more stream-lined solution, which not only handles the comparison of the DUT with the Matlab-based reference model, but also provides for analog stimulus generation, analog based performance measurements and metric collection.

A. Overview and Advantages

The system described here consists of the direct integration of Octave with SystemVerilog. GNU/Octave [5] is an open-source alternative to Matlab®, which supports a large sub-set of its features. While it may not be able to replace the advanced Matlab® tool-boxes that are commonly used in the generation/derivation of systems, it is complete enough to implement the verification part – i.e. stimulus generation, numerical evaluation of the system, and results data-processing. Integrating SystemVerilog with Octave, has another advantage in that it can be used as part of a parallel regression environment without incurring additional license overheads.

There are a number of advantages to integrating System-Verilog and Octave for verification:

- *Realistic analog stimulus generation – coherent sine-waves, multi-tone, single-tone, etc.*
- *Low-cost predictor model generation – Matlab® analog-model reuse, DSP, filtering, etc.*
- *Real performance verification - reuse the same analysis code from Matlab.*
- *Directly carries over to Co-Simulation – reuse UVM testbenches directly in schematic cosim.*

B. Integration Method

The method of integration uses the SV-Direct Programming Interface (DPI). This is a C/C++ based interface to SV that builds on similar interfaces provided in previous generations of the Verilog language. Octave is also C/C++ based, and the octave-core runtime is provided as a C library. The Octave language is designed to be able to be embedded [6] into other programs, and as a result integrating it with the DPI is very straight-forward.

The core of the integration is a C++ wrapper file that translates/passes the data from the SystemVerilog DPI interface to the Octave library interface. What this means is that the octave-interpreter is running as a child-process of the SystemVerilog simulator, and data is passed between them directly. This provides for maximum performance during simulation and results very low overhead.

The wrapper exposes functions to SystemVerilog that can be called directly from testbenches. The code shown in *Listing 1* outlines a very simple example testbench usage – which consists 3 parts:

1) *The testbench-DPI interface,*
2) *The C-DPI-Interface,*
3) *An example calling of the code from the testbench*

The first 2 parts are portable and re-usable between projects and testbenches. The example also illustrates the potential for returning multiple results from Octave via SV ‘structs’ which further aids the readability of the testbench code.

```
// General Octave Setup Functions
import "DPI-C" function void oct_init(string path="");
import "DPI-C" function void oct_exit();

// Call Octave C Wrapper Functions
import "DPI-C" function void oct_fft(input  int  dyn_arr[],
                                     output real res_arr[]);
....

// result struct
typedef struct { ... real sfdr; } fft_result;

// Calls oct_fft function, and returns results in a struct for each eaccess
function fft_result fft(input int dyn_arr[]);
real res_arr[10];
begin
  oct_fft(dyn_arr, res_arr);
  fft.sndr = res_arr[0];
  ....
end
endfunction
-- INSERT --                                          17,18         Bot
```

```
initial begin
    for(int i=0; i<1024; i++) begin   // Gather Data
        @(negedge adc_clk);
        fix_arr[i] = adc_data;
    end
    result = fft(fix_arr);            // Call Octave
    $display("SNR: %f", result.snr);  // Report results
```

```
#include <octave/octave.h>
#include <svdpi.h>
....

// Macros
#define SV_ARR_REAL(arr,idx) *(double *)svGetArrElemPtr1(arr,idx)

// Octave FFT Wrapper
extern "C" void oct_fft (const svOpenArrayHandle dyn_arr,
                         const svOpenArrayHandle res_arr)
{
  // 1) Prepare inputs - arrays are converted to Matrix types for octave
  Matrix data_in = Matrix(1, len);
  for(int i=0; i<svLength(dyn_arr, 1); i++) {
    data_in(i) = SV_ARR_INT(dyn_arr,i);
  }

  // 2) Call the function
  octave_value_list func_in;      // create the array (matrix) to input
  func_in(0) = data_in;           // pass data
  octave_value_list out = feval("fanalyze", func_in);

  // 3) Return the results
  for(octave_idx_type i=0; i<out.length(); i++) {
    if(i < svLength(res_arr,1)) {
      SV_ARR_REAL(res_arr, i) = out(i).double_value();
    }
  }
}
```

Listing 1: **Upper-Left**: SV wrapper, **Lower-Left:** Example of calling Octave from SV, **Right:** DPI-C-Octave Bridge

In the above example, the function '*fft()*' calls the '*oct_fft()'* function directly via the DPI. This C wrapper in turn calls the Octave function '*fanalyze*'. It should be noted that '*fanalyze*' is not an octave built-in function, but instead a user written Matlab®/Octave script. The ability to call full scripts to process data and return the result directly within the simulation further underscores the power of this method – as many numerically intensive processing tasks can chained together in a single Octave .m file and the overall result returned to SV for result checking and score-boarding. It is also possible to use Octave to export results in the form of plots and images, as can be seen here:

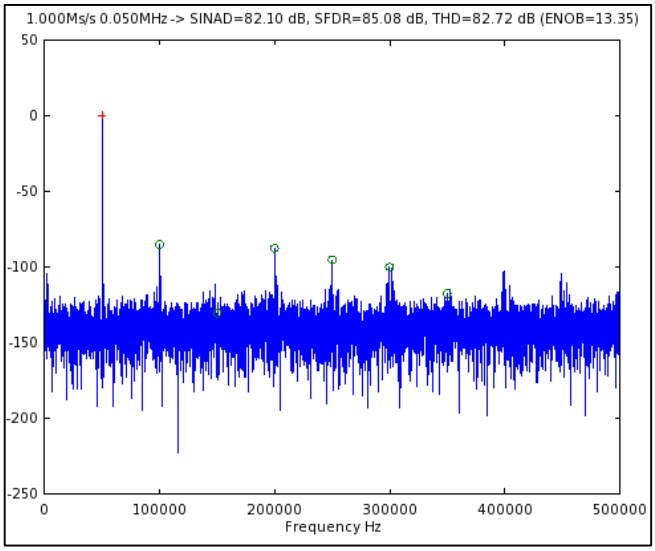

Figure III-1: Example FFT result image output from SV (via Octave)

C. Performance and Results

The SV-Octave integration greatly enhances an AMS-UVM environment by making maximal re-use of already verified reference Matlab code. This reduces the effort require to generate predictor models of complex AMS and signal processing data-paths. It also provides access from SV to the same tools uses to analyse the results of mixed signal systems, without requiring these to be re-implemented within SV. This method also allows for interactivity and feedback between the testbench and Octave – where SV randomization can be used to generate inputs to Octave which in turn can generate analog stimulus on-the-fly. These kinds of interactive and dynamic simulations are very difficult and cumbersome to achieve with a traditional vector-file based approach. The DPI interface was tested with a number of leading commercial Verilog simulators and shown to work with them all, which demonstrates the portability of the method. The direct integration also has a performance advantage of at least

1.5x over a vector-file approach as outlined in Table III-1. This performance delta increases for smaller data-packets, as the file-system overhead would be more significant.

Table III-1: Performance of Direct Integration vs Vector-File Approach

Test Case	**Measured Performance**		
	Proposed Method (time/1000 ops)	***Vector-File Method (time/1000 ops)***	***Performance Delta***
FFT (1024-point)	3.6s	5.7s	1.58x

IV. UVM ENVIRONMENT

UVM is a methodology for the functional verification of primarily digital hardware, however with some modifications it can be extended to aid verification of both analog and digital hardware. The proposed UVM environment is shown in Figure IV-1 Here the design-under-test (DUT) is surrounded by a number of SystemVerilog interface components [7]. These interface components facilitate interactions between the DUT and components of the verification environment. Most verification activity involves driving and observing the DUT's top-level I/O's which are connected directly to interfaces.

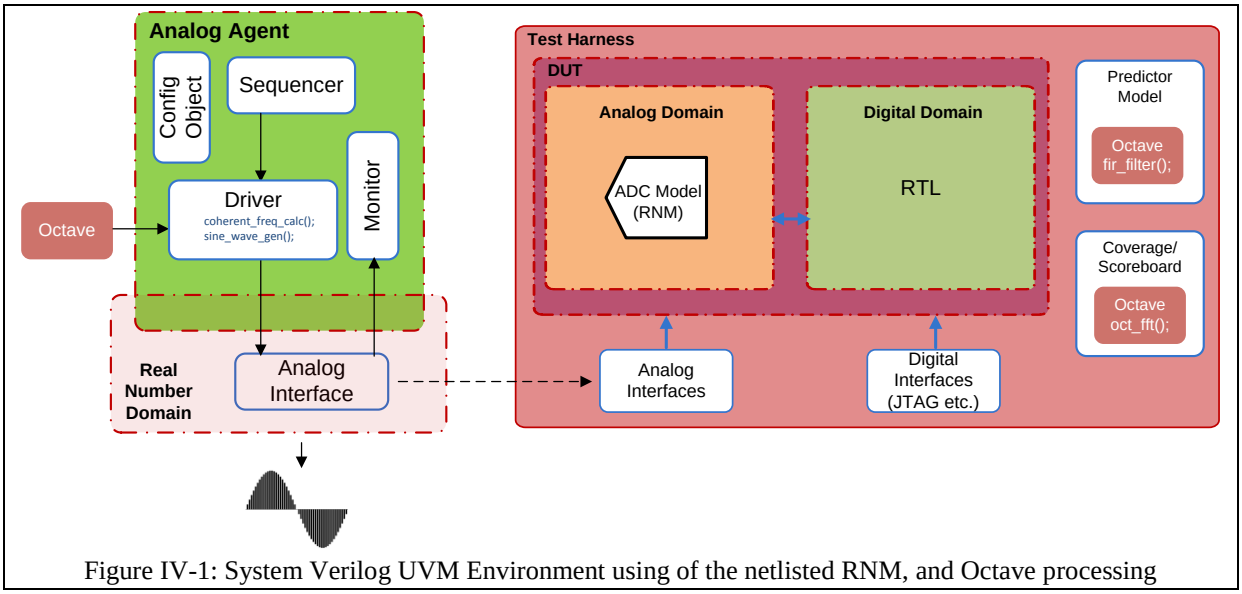

Figure IV-1: System Verilog UVM Environment using of the netlisted RNM, and Octave processing

A. *Analog Agent*

As the DUT in question here is a mixed-signal design, parts of the DUT and their corresponding SystemVerilog interfaces and verification components will need to support real numbers in order to fully verify the combined analog and digital functionality of the design. The signals on each interface in the harness are driven and observed by a corresponding UVM agent.

Each agent in the environment will typically use the standard UVM agent pattern: sequencer, driver, and monitor sub-blocks. Also, in order to configure agents to behave differently from one test to the next, each agent will have a configuration object which can be modified from the test level. Dealing with real numbers is a departure from the standard digital verification environment that is supported by UVM. Therefore the structure of the UVM agents that will be interacting with real ports will be architected to handle real numbers. Figure IV-1 presents a proposed architecture of such an agent.

It is envisaged that the transactions generated by the sequencer will encode the real number value using integers. Integers can be randomized and coverage collected on then. The integer value is converted to a real number by the driver, which applies it via the interface to the real number port on the DUT. The analog transaction will be a sine wave of N Points and frequency F. The driver will call Octave functions in order to generate the analog

stimulus for the DUT. The monitor collects the transactions via the same interface, and converts the real number back to its integer equivalent to be sent to various subscribers such as the scoreboard and coverage collector.

B. Predictor Model & Scoreboard

In order to fully exploit the power of constrained random stimulus generation, the verification environment needs to be able to automatically predict the state of the DUT and its outputs. Therefore a predictor model, as shown in Figure IV-1, is required.

The predictor model will generate output transactions in response to the same input transaction that are applied to the DUT. This model also calls on Octave functions to generate the expected output.

The scoreboard will compare the DUT's responses against the predictor model's output transactions in order to determine the functional correctness of the DUT. Again the scoreboard will call on Octave functions to implement the comparison.

C. Re-use

With ever more complex designs combining analog and digital verification productivity is a major issue. Productivity can be greatly improved through the re-use of verification components. One of the main benefits of using a UVM environment is that it facilitates this re-use of multiple verification elements in the form of SystemVerilog classes and other components. In addition, having the flexibility to configure individual components and import elements such as Octave greatly expands the verification space.

D. Application to Co-Simulation

Co-Simulation between SystemVerilog and schematic (Spice/Spectre) is a key requirement for AMS sign-off. However, porting/developing test-cases that can deal with the non-ideal behaviour of the analog circuit vs the model can be difficult. These methods, which allows the SV-UVM testbench to re-use the typical analog analysis functions (such as the FFT show here) allows not only for direct portability between the RNM and the Co-Simulation, but also allows the actual analog performance to be evaluated and reported accurately.

V. CONCLUSION

In this paper we have outlined a comprehensive AMS verification environment that enables the power of UVM to be extended to include AMS verification. The standard UVM structure is combined with Real Number Modelling and Octave to enable high-quality analog modelling, realistic stimulus generation, and results analysis. The combination of these methods allows functional coverage in the analog domain to be included as part of the UVM environment, with allows much greater coverage of analog use-cases, modes and interactions via regression and constrained-random based testing. The methods outlined are fully compatible with the major SystemVerilog simulators. Finally, the environment ensures low-overhead via automation and re-use, while at the same time providing high-performance in terms of simulation run-time and system level verification coverage.

REFERENCES

[1] A. Freitas and R. Santonja. "UVM Ready: Transitioning Mixed-Signal Verification Environments to Universal Verification Methodology" DVCON Europe (2014).

[2] J. Qiu, and K. Schwartz. "NVVM: A Netlist-based Verilog Verification Methodology for Mixed-Signal Design" DVCON Europe (2014).

[3] Online Resources from http://www.cadence.com/rl/Resources/datasheets/Virtuoso_MS_Behavioral_Modeling_DS.pdf.

[4] 1800-2012 - IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language, ISBN: 978-0-7381-8110-3.

[5] GNU/Octave: http://www.gnu.org/software/octave/

[6] Embedding Octave: https://www.gnu.org/software/octave/doc/interpreter/Standalone-Programs.html

[7] Online Resources from http://www.accellera.org/downloads/standards/uvm.