Flattening the UVM Learning curve: Automated solutions for DSP filter Verification

Avinash Lakshminarayana, Eric Jackowski
avlakshm@silabs.com, eric.jackowski@silabs.com
Silicon Labs
400 W Cesar Chavez St
Austin, TX 78701
USA

Eric Cigan, Mark Lin
ecigan@mathworks.com, marklin@mathworks.com
MathWorks
1 Apple Hill Dr
Natick, MA 01701
USA

***Abstract*- By their nature, DSP filters are computationally intensive, predominantly structured as data paths with relatively simple control paths. Algorithm developers often develop filters using MATLAB®, with hardware engineers developing RTL to implement the MATLAB specification. Traditionally, the verification of filters at unit level has been limited, with the RTL designer developing a simple testbench comparing RTL outputs against pre-generated MATLAB golden vectors. That approach is now changing because the complexity of control paths is increasing, algorithms are being modified frequently to conform with evolving standards, and optimization is being done to minimize power consumption. These developments drive the need for more comprehensive verification methods.**

When considered as a standalone testbench, the DSP filter testbench is still quite small, but the number of such benches in a project is quite high. So, while we see the obvious benefits of implementing our testbenches in UVM, both the size and the number of testbenches make it prohibitive. This led us to explore "lite-UVM" testbenches with a high degree of automation. Our goal was to realize the benefits of UVM-based verification, but with minimal bring-up efforts.

In this paper, we document various solutions we explored to develop more robust verification, spanning from home-brewed solutions to off-the-shelf industry tools. The intended audience for this presentation is DV engineers looking for a light UVM framework that verifies DSP filters with complex data paths, but simple control logic.

We have classified this paper into four parts - Introduction, Related Work, Automation Strategies for DSP Filters, and Conclusion.

Keywords: DSP verification, UVM code generation. dpigen, uvmbuild

I. Introduction

The digital filters have come to occupy an important role in IoT designs. They are used to perform a series of mathematical operations on digital signals with the primary goal of transmitting and receiving signals efficiently and with minimal distortion. Typically, the systems team evaluates, develops, and tunes the entire filter chain in a MATLAB environment. Then an RTL designer breaks the model into smaller components and implements them in SystemVerilog/VHDL®. The systems team also provides a few golden test vectors in a text file format. These golden vectors are supposed to represent real-world use cases. The designer and/or verification engineer use these test vectors to compare against an RTL implementation. As the designs get more complex, we increasingly find a need to run the MATLAB model with new input patterns and compare the results against the RTL implementation. Such patterns may be meaningless in the context of a full system, but they will help with initial debug. Sometimes, a constrained random input pattern may also unearth a corner case bug in the RTL design. Further, a lot of control logic is being added into the DSP filter IPs — clock gating, pipelining, and interrupts to name a few. Another recent design choice includes a coprocessor implementation with a standard interface on one side of the filter and a glue logic that talks to the DSP core on the other side.

A narrow time-to-market in the IoT space means that bug escapes can be quite costly. At the same time, a quick bring-up and lean testbench gain precedence over complex environments with long design and maintenance cycles. DSP verification can obviously benefit from a UVM methodology. But deploying UVM comes with its own set of challenges. First off, the filters have a relatively small footprint to justify the initial bring-up time of a full-fledged UVM environment. Designing math-intensive scoreboards will be laborious and error-prone. Alternately, one will have to invoke the MATLAB model externally and post-process the data. Such efforts are easier said than done.

Finally, because it is an interdisciplinary field, we also want to increase the level of abstraction so that non-DV engineers can understand and change the testbench knobs easily.

On the bright side, vertical reuse is limited for filter testbench, thus simplifying many of our design choices. From a verification point of view, DSP filters have many shared characteristics. The input and output ports typically have I/Q data signals in fixed-point format. The data is processed usually in stream or block mode in a simple protocol. The input patterns like impulse, step, and ramp can be reused across many filters. DSP testbenches lend themselves well to automatic generation and templating. Keeping all these factors in mind, we developed various strategies to deploy UVM extensively in filter verification. This work documents those strategies, performs pros-and-cons analysis of each, and shares some guidelines on what worked and what did not work for our team.

II. Related Work

A variety of tools automating the creation of UVM environments have been developed and popularized since UVM was first approved as a standard by the Accellera Systems Initiative in 2011. In 2012 Mentor Graphics (now known as Siemens EDA) introduced UVM Express, later renamed UVM Framework or UVMF, to aid design teams adopting the UVM methodology [1]. The UVMF from Siemens EDA is an open-source package that provides a UVM methodology and code generator for rapid testbench generation [2].

More recently, researchers have explored use of open-source tools in conjunction with UVM. PyUVM is an implementation of UVM based on Python® rather than SystemVerilog [3]. PyUVM is integrated with Cocotb, a Python library for writing synchronous logic available as a plugin for RTL simulators [4]. Communication between the testbench and the RTL simulator is via VPI, FLI, and VHPI.

In this paper, we consider the use of UVM generation from design specifications developed in MATLAB and Simulink® as high-level modeling languages. However, UVM creation tools based on C/C++/SystemC™ design specifications have also been developed. In 2017, Mentor Graphics added UVM generation to the Catapult™ HLS product through integration with UVMF, with the HLS C models integrated as predictors [5].

There has been a more focused development of tools that incorporate MATLAB and Simulink into UVM environments. In the early 2010s, Mentor Graphics developed tools and techniques enabling the use of MATLAB in predictors, checkers, and other testbench components in conjunction with the Questa SystemVerilog DPI or VHDL FLI using the MATLAB Engine interface, an API that allows MATLAB to be run from a shared library [6].

In 2013–14, MathWorks introduced automatic generation of SystemVerilog DPI from MATLAB code and Simulink models [7]. The SystemVerilog DPI component generator produces shared library from MATLAB functions or Simulink subsystems for specified arguments and data types. It generates a directory structure that includes DPI-C wrappers, header files, makefiles, SystemVerilog testbenches, and simulation scripts for several commercial HDL simulators. Mentor Graphics subsequently updated UVMF to automatically incorporate DPI-C components generated from MATLAB and Simulink by this method [8].

In 2019 MathWorks added the ability to generate complete UVM environments from Simulink subsystems for various design topologies with subsystems corresponding to UVM sequences, drivers, DUTs, monitors, scoreboards, etc. [9].

In a 2021 study, a verification engineer developing testbenches for a datapath-oriented signal processing design developed in MATLAB and Simulink found that the automated UVM environment generator provided a useful unit test environment, but that using the SystemVerilog DPI generator with UVMF was better suited to development of chip-level test environments [10].

In this paper, we made use of these DPI and UVM generation capabilities and applied them to automated testbench generation for signal processing designs.

III. Automation Strategies for DSP Filters

Now, we proceed to explain various methods we deployed to verify DSP filters. The serial order of the strategies presented might lead one to conclude that each method proposed is an evolution over the previous one. But this is not our intent. Rather, we found that each of the methods add unique value for our verification and each of them come with certain constraints and limitations. It is up to the user to pick the methods that best suit their needs.

First, we'll dive into the definition of a programmable FIR filter that can be used to evaluate each of the different methods. The FIR filter is symmetric and has 29 programmable TAPs. The user can also select a particular latency through the FIR filter by reducing the number of TAPs to 23, 17 or 11. We focused on verifying the datapath against bit accurate model for each of the latency selection values.

1. Creating DPI-C Based Verification Components

In our first iteration, we started with a MATLAB reference model and used MATLAB Compiler™ to convert it into a C-based Linux® shared object (.so) file [11]. We then build a DPI layer around a C code wrapper of the .so file, which can be called from an SV/UVM environment, at the start of test. For clarity, we divide this section into three parts as shown in Fig. 1 – the MATLAB layer, DPI layer, and SV Layer. Let's look at each of the layers in more detail.

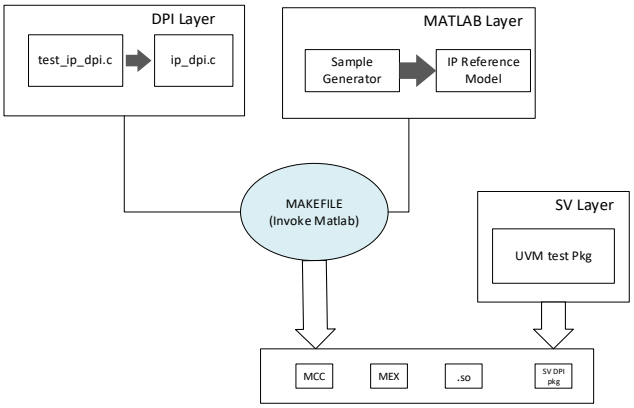

Figure 1. Three layers of MATLAB Compiler DPI-C based VC development. (Blue highlight specifies automated section.)

A. MATLAB Layer

The MATLAB layer has two components — a sample generator and the IP reference model. As the name implies, the code is completely in MATLAB. The sample generator takes the necessary configurable options as its input. Some of these options are passed directly to the reference model. Examples include filter coefficients, decimation rate, etc. The other options are consumed by the sample generator. Some options, such as number of samples and random seed, are used to populate the input samples for the reference model. We chose to generate input samples in the MATLAB layer instead of the SV layer because we can produce DSP-specific patterns like sine waves with just a few lines of code. Producing DSP patterns in SystemVerilog cannot be done with such ease. Additionally, MATLAB has random number generator functions that can be controlled through SV seed, enabling us to manage constrained random verification. Once the input samples are created, we invoke the reference model and collect and post-process the output data. Both the input and output samples are returned by the sample generator to the calling function. The UVM environment can then send input samples to the DUT with compliance to the protocol, and the output samples can be used by the scoreboard to compare with RTL output. Refer to Figure 2 for an example sample generator. The FIR Filter sample generator took an array of TAPs, a control value for the latency, the random seed, and a stimulus selection value. The type of stimulus is selectable between completely random, a sinusoidal tone, an Impulse, and maximum coefficient magnitude. The impulse was used as a simple turn on tests to see that all the coefficients ended up on the output. The sinusoidal tone was used to cover any frequencies of interest by the systems team. The maximum coefficient magnitude takes a random set of coefficients and maximizes the positive magnitude and negative magnitude. This stimulus and random stimulus exposed the most bugs with the accumulators and the associated rounding and saturation.

B. DPI Layer

In this layer, we need a library of DPI calls to interact with MATLAB functions. The DPI implementation is in C, but it makes use of MEX functions [12]. MEX functions, or MATLAB executables, refer to programs that are automatically loaded and can be called like any MATLAB function. Using C-MEX file applications, we can write C programs that can be called from the MATLAB command line. Some of the DPI functions handle MATLAB bookkeeping activities like initializing and terminating the tool and setting up the libraries. A separate DPI function is needed to invoke the sample generator. This function also performs low-level memory management like allocation and clearing using mxArray. mxArray is a special array type to transfer data between C and MATLAB [13].

At this stage, we also recommend creating an optional C test wrapper that can call all the above functions and display the input and output samples generated. This step is a quick sanity check that ensures the DPI function calls are working as expected.

```
function [out, in] = fir_filter_sample_generator(sample_type, tone_in_hz, latency_ctrl, coefficients, random_seed,
max_rtl_samples)

  % Allows us to see the full values
  format longG;

  % Seed RNG
  fprintf('\nMatlab Seed=%u\n\n',random_seed)
  rng(random_seed);

  % Enumerate Selections
  RANDOM = 255;
  MAX_COEFF_MAG = 200;
  TONE  = 100;
  IMPULSE = 10;

  % Generate Input Samples
  switch sample_type

    case RANDOM
      % random samples
      rtl_i = randi([-2^18 ((2^18)-1)], max_rtl_samples, 1);
      rtl_q = randi([-2^18 ((2^18)-1)], max_rtl_samples, 1);

    case MAX_COEFF_MAG
      % max magnitued sum of coefficients check by aligning the largest
      % input with the coeffecient
      % Create Max positve Value on inphase, Max Negative value on quadrature
      max_pos = (2^18)-1;
      max_neg = -1*2^18;

      coef = ones(1,29);
      coef(1:15) = coefficients(1:15);
      coef(16:29) = fliplr(coefficients(1:14));

      len = length(coef);

      % Determine indexes with positive or negative coefficients
      pos_coef_idx = coef >= 0;
      neg_coef_idx = coef < 0;

      % Generate all postive values for inphase
      rtl_i = ones(length(coef),1);
      rtl_i(pos_coef_idx) = max_pos;
      rtl_i(neg_coef_idx) = max_neg;

      % Generate all negative values for quadraure
      rtl_q = ones(length(coef),1);
      rtl_q(neg_coef_idx) = max_pos;
      rtl_q(pos_coef_idx) = max_neg;

      % Add on some zeroes to flush the filter
      rtl_i = [rtl_i; zeros(len,1)];
      rtl_q = [rtl_q; zeros(len,1)];

    case TONE
      % output a tone for 1ms at Full Scale
      bits_full_scale = 19;
      tmp = tone_gen(tone_in_hz,bits_full_scale);
      tmp_length = length(tmp);
      if  tmp_length < max_rtl_samples
        stop_index = tmp_length;
      else
        stop_index = max_rtl_samples;
      end
      tmp = tmp(1:stop_index);
      rtl_i = real(tmp);
      rtl_q = imag(tmp);

    case IMPULSE
      % pulse high for one clock and send out the coeffs until 0 again
      rtl_i = [(2^18)-1; zeros(29,1)];
      rtl_q = [-2^18; zeros(29,1)];

    otherwise
      disp("Invalid Sample Type");
      disp(sample_type);
      exit();
  end

  % Create complex input
  in = complex( rtl_i(:), rtl_q(:) );

  % Call Bit Acurate model for output samples
  [out, ~] = channel_filter( in, latency_ctrl, coefficients );

end
```

Figure 2. Example sample generator code.

C. SystemVerilog Layer

We create an SV package file with a set of functions that call the DPI functions from the DPI layer. All the DPI-specific code is included here so the rest of the testbench components can reference these functions like any other SV calls. In the UVM test, we can set up a directed test with a specific input pattern and control. In the case of the FIR Filter, we randomly select a sample type, latency control, and coefficients, and then call appropriate package functions to run the model. From this step we get an input pattern generated by the sample generator and an expected output pattern from the MATLAB model. Note that this is a zero-simulation-time event, executed at the start of the simulation. During the run phase, we supply the input data at clocked intervals, honoring any protocol requirements. At the end of the test, we compare the RTL output against the output pattern from the MATLAB model.

Using this three-part technique, we can generate the outputs from any input pattern we want, without having to rewrite the reference model in SystemVerilog. We do this by calling the appropriate MATLAB and gcc commands using a makefile to compile, run, and check our MATLAB and DPI layer setup. Hence, this flow enables us to change the input patterns on the fly instead of relying just on the golden vector file supplied by the system engineer. It has the added benefit of not using a MATLAB license during regression runs.

We found that most of the code, particularly in the DPI layer, was repetitive. The low-level programming is also error prone. We recognized that further automation was possible. We will revisit this issue in later sections. However, one of the biggest challenges we faced with this flow was debuggability. When fully operational, we have three tools working in tandem — MATLAB, gcc, and SV simulator. The error messages from one tool were getting clobbered up by another. Some files worked well in one layer and not so well when integrated with other layers. Many failures in such cases were “null pointer” errors or memory access issues. We had very limited visibility to what was going on underneath. There is no mechanism to step through the code or introduce breakpoints across the tool/language boundaries. Worst case, we binary search the MATLAB and the SV by commenting out sections to identify the error. Finally, this approach only addresses some of our concerns — integrating the MATLAB reference model into an existing UVM testbench and producing arbitrary data samples.

```
//
// DPI C Functions
//

import "DPI-C" function int  firFilterDpiInitializeMatlab(string options[MAX_MATLAB_OPTIONS], int count);
import "DPI-C" function int  firFilterDpiInitializeLibrary();
import "DPI-C" function int  firFilterDpiRunSampleGenerator(input int unsigned  sample_type,
                                                            input int unsigned  tone_in_hz,
                                                            input int unsigned  latency_ctrl,
                                                            input real          coeff[NUM_OF_SYMETRIC_COEFFS],
                                                            input int unsigned  size_of_coeff,
                                                            input int unsigned  random_seed,
                                                            input int unsigned  max_rtl_samples,
                                                            output int unsigned size_of_in,
                                                            output int unsigned in_i [RADIO_MODEL_MAX_SAMPLES],
                                                            output int unsigned in_q [RADIO_MODEL_MAX_SAMPLES],
                                                            output int unsigned size_of_out,
                                                            output int          out_i [RADIO_MODEL_MAX_SAMPLES],
                                                            output int          out_q [RADIO_MODEL_MAX_SAMPLES]);
import "DPI-C" function int  firFilterDpiTerminateLibrary();
import "DPI-C" function int  firFilterDpiTerminateMatlab();



//
// SV Function calls wrapping DPI C funcitons
//
matlab_options[matlab_options_count++] = "-nosplash";
matlab_options[matlab_options_count++] = "-nodisplay";
matlab_options[matlab_options_count++] = "-nojvm";
fir_filter_initialize_matlab(matlab_options,matlab_options_count);

fir_filter_initialize_radio_model();

fir_filter_run_radio_model(sample_type,
                           tone_in_hz,
                           latency_ctrl,
                           coeff_for_matlab,
                           size_of_in,
                           in_i,
```

Figure 3. DPI function calls for FIR Filter.

2. Coprocessor Testbench Generation

Next, we attempted to reduce the initial bring-up time of the UVM testbench. The coprocessor architecture is gaining ground in DSP designs [14]. Multiple filters in the SoC will have similar I/O and latency requirements. They typically operate in the streaming mode, where they process one byte of data at a time and produce an output, or they operate in block mode, where they operate on a chunk of data at a time. The control paths also share commonality. Some filters have control signals like ready and valid to accommodate gaps while others operate on a steady stream of data. The main difference between the filters comes in the data path. It's compute-intensive and can be encapsulated inside a well-defined core. Clearly, the DSP filters render themselves well for coprocessor implementation. This also simplifies the verification effort. We decided such coprocessor filters will be ideal candidates for trying out automation.

We deployed a two-pronged strategy. First, we created a parametrized verification component (VC) that had all the common utilities we needed for verification. This includes a Register Access Layer to program registers, protocol layer to obey any I/O requirements, the clocking blocks, and the interrupt handlers. This VC could be instantiated in any testbench, connected, and configured to the filter being verified. Some configuration options included sending data to the filter as a stream of packets vs. one block at a time, allowing backpressure and bubble insertion if the design allowed. We used a layering approach for our UVM agents as discussed in [15]. We first came up with a generic DSP transaction that could be applied to any filter. The same transaction could be used to receive processed data from the filter and sent up to the scoreboard. Its variables include the data packet we wanted to process and a few control signals like delays and type of operations. Our coprocessor interfaces fell into three buckets — accept I/O data as an AHB transaction, a parametrizable FIFO interface, and a simple bypass mode with no control signals. We created a UVM agent for each interface and a corresponding translation sequence. We tried to parametrize the VCs wherever possible, especially the interfaces. We took advantage of harness files as described in [16]. We wrapped all this infrastructure inside a Coprocessor UVM environment that can be instantiated in any testbench. We also created a UVM sequence library for common utilities — send and receive DSP coprocessor transaction through the coprocessor environment, enable local clock gating, etc.

The second piece of this puzzle involved developing a scripting framework that automatically generated the testbench for us. The identical I/O ports and the communication protocol simplified our scripting. For each coprocessor, we needed a UVM testbench that instantiates and configures the coprocessor UVM environment. In addition, we needed a base test to set up and call the MATLAB APIs we discussed in the previous section and then call the sequences to send and receive data from the DSP filter. We created a template file for each of these components and built a configurable Python layer that automatically generated a basic testbench. It was also able to create a few other common tests like register and interrupt tests. Once generated, the DV engineer could then edit the individual files to customize or enhance the tests.

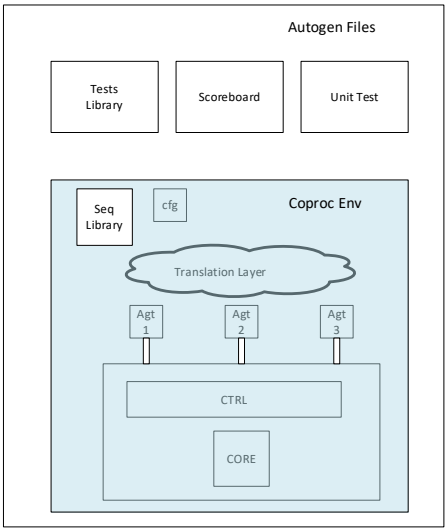

Figure 4. Testbench structure for Coprocessor based filter.

The home-brewed solutions were great but posed an abundance of maintenance challenges. Each time we had a bug in the framework, we had to regenerate every testbench. Some testbenches needed manual code changes. There was no easy way to reconcile the manual changes with the regenerated code. Besides, we were unable to meet our abstraction requirements. Non-DV engineers still could not take part in the process.

3. Automated Flow

In this section, we describe our experience using off-the-shelf tools available to accelerate UVM testbench development for DSP filters. The DV team could choose to model the data path using different solutions: C, Python, Octave, MATLAB, or SV. We chose to use MATLAB as that is our systems team's preferred modeling language, and we can increase the return on investment of their effort in creating the MATLAB model. If we had not done this, the burden of modeling each DSP block in some other language would be put on the DV team. One additional benefit in moving to a bit-accurate MATLAB check is that the DV engineer doesn't need to rely on parametric checkers found in DSP such as SNR. Now, we explore two MATLAB utilities — dpigen and uvmbuild. The dpigen utility helps build DPI functions from MATLAB specifications that can be called from an existing UVM environment. The uvmbuild utility can produce a complete UVM testbench from both a MATLAB and Simulink specification.

A. DPI Component Generation

In an earlier section, we discussed the difficulties in creating and debugging DPI files. The MATLAB dpigen command helps to alleviate this problem [17]. This command generates SystemVerilog DPI components from MATLAB functions. In the dpigen flow, the testbench development can be viewed under three sections — data source, algorithm, and analysis. The data source is the stimulus generator, algorithm is our reference model, and analysis refers to the checker. We can develop all three components in MATLAB and then invoke dpigen on each component to generate the DPI-C version. The SV test will first call the data source DPI function to get the input samples. The input samples are sent to the DUT and to the algorithm DPI function. The output from the algorithm and DUT are then passed to the Analysis DPI function to check if the outputs correlate (Fig. 4). The data source and checker MATLAB models can be generic and reused for different filters.

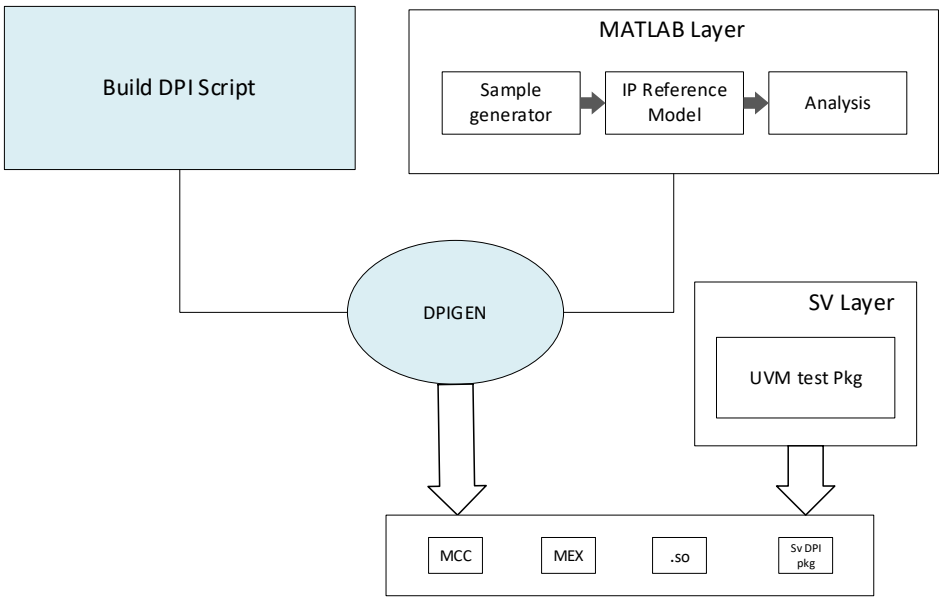

Figure 5. Automating DPI code generation using dpigen. (Blue highlight specifies automated section.)

Also note that both MATLAB and Simulink models can be used in this process. For MATLAB functions, the first step was to review and prepare the function for code generation. This included use of persistent variables to hold state within the generated C model and the use of fixed-point data types for all the variables. The input or output argument of the MATLAB function can be vector or scalar. MATLAB wrapper functions were created to address the situation where modification to the original function was not desirable or prohibitive.

MATLAB functions by nature are untimed. From the testbench perspective, C model operation completes in zero time. Thus, for cycle accuracy, functionality was captured in MATLAB and integrated into Simulink, where timing

aspects could be modeled. This was done through the MATLAB function block in Simulink. Then, the C code was generated from the Simulink model. Visibility into the generated C model was available to the RTL testbench through use of Simulink test points [18]. Each test point in the Simulink model resulted in a DPI-C function that the RTL testbench can call and sample the state of that point. This helped improve debuggability.

By integrating MATLAB function into Simulink, it also enables runtime configurability of DPI-C functions. In the case of the Sine function, options such as amplitude, frequency or phase are passed via MATLAB function input arguments. These arguments are then driven by Simulink Parameter variable. Like Simulink test points, Simulink Parameter variable will result in a DPI-C function that RTL testbench can call to set the value of that variable during runtime. This enable reusability and randomization of RTL testbench.

Code generation was performed in the same environment as the RTL testbench (i.e., Linux). It is possible to do the same in Windows. A compilation makefile was created to call the dpigen with appropriate arguments. This process provided:

1. C code of MATLAB or Simulink model
2. SystemVerilog wrapper for the C code
3. SystemVerilog package containing DPI-C functions that represents the model
4. SystemVerilog module using the generated DPI-C function
5. Shared object of DPI-C function for the EDA Tool

B. UVM Testbench Generation

The uvmbuild command enables us to create individual UVM components — either through a drag-and-drop toolbox in Simulink or through a MATLAB function [19]. While creating the component, we can focus on the algorithm and dataflow and not worry about the UVM semantics. We can add as many or as few details into our model as needed.

In Figure 6, we show composition of the Sequence block in Simulink. It translates to a uvm_sequence upon code generation. We create a subsystem with three input ports — sample_type, random_seed, and max_value. These values will be declared as variables in the generated sequence. They can be controlled by the uvm_test. We have two instances of a random number generator that produce real and imaginary components of the input sample. The random number generator comes with certain configurable options like data type and maximum and minimum values. The converter block combines the two real values to produce a complex signal. The Sequence subsystem has two output ports producing identical values.

In one test, the sine function was used in the sequence block, where its parameter's upper and lower limits could be defined by using Simulink Parameter's min and max field. Uvmbuild will then automatically generated a random constraint block for each parameter using the defined limits. Before the test starts, the DPI-C of the sine function in the sequence is configured via the generated DPI parameter set function in start_of_simulation_phase(). If required, additional randomization could be done by extending the generated sequence class and replacing it via the factory at uvm_test.

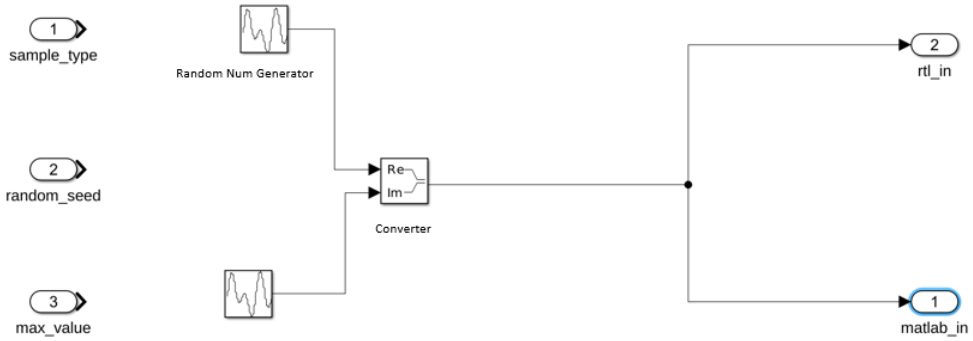

Figure 6. Sequence block in Simulink.

The subsystem can be implemented in a MATLAB function if the user is more familiar with the MATLAB environment. Typically, the Reference subsystem will be a MATLAB model obtained from the systems team. The subsystem can also be a dummy placeholder. For example, the DUT can be an empty block that will be replaced with RTL model after code generation. These subsystems can be saved, shared, and reused as standalone units.

Typically, we use the subsystems together to compose a full testbench and then use uvmbuild to generate an executable UVM testbench. The general structure of the testbench should reflect Figure 7. The tool expects that we create some components compulsorily for successful code generation. This includes the sequence, DUT, and scoreboard. The shaded subsystems are, however, optional.

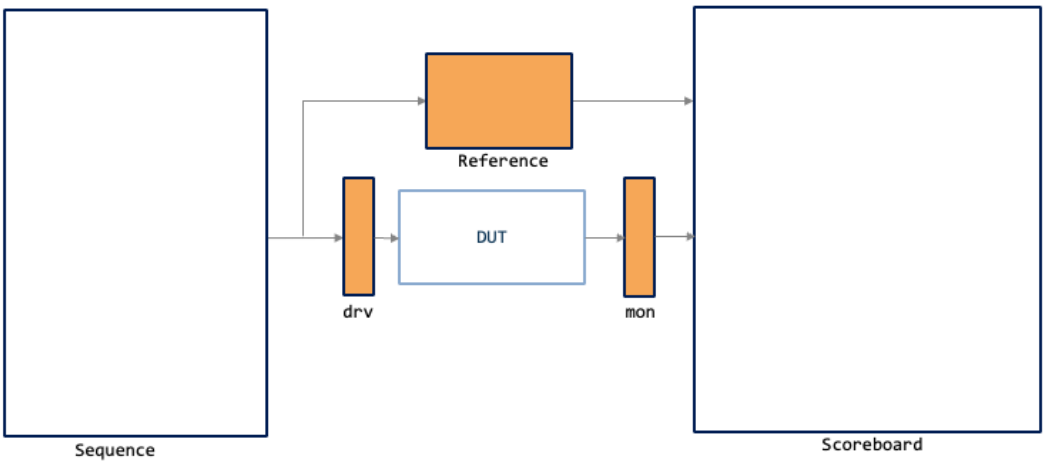

Figure 7. UVM testbench composition in Simulink.

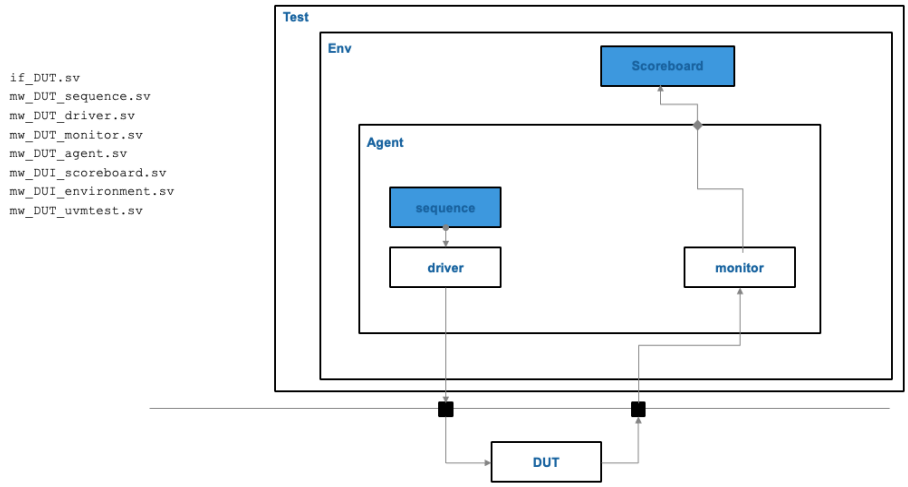

Figure 8. The generated testbench. The items on the left are the SV files created by the tool.

Figure 8 shows a testbench structure generated by the uvmbuild command. By default, uvmbuild will create a driver and monitor that are simply pass-through components. These components are project specific and are meant to be overridden via the UVM Factory. The pin-level activity is better implemented in SystemVerilog instead of MATLAB. The log message and its UVM verbosity can be controlled using HDL Verifier™. But we didn't pursue that in this exercise. Once the testbench met all our requirements, we save it in a template file. Simulink provides an option to export and save the testbench model in 'templatename.sltx' format [20].

The starting point for any new filter testbench development will be this template file. It will have the default sequence block and checker. The engineer can click through each component and customize it according to the needs of each individual filter testbench. One subsystem that needs to be changed always will be the Reference subsystem. It will call the system model of the filter being verified. In some cases, the scoreboard subsystem may need

modification. For example, we encountered some filters where output sample mismatch was tolerable for first output sample.

The auto generated UVM code was easily readable. So, we will have the option to make further changes to the SV code, if needed. We do not recommend using this option. The changes can't be reused. More importantly, if the testbench must be regenerated, all the manual edits will have to be redone.

The tool suite was very beneficial and elegant. After the initial hurdles, creating testbench for new filters was easy. The bring-up time is in the order of days instead of weeks. If needed, we have a path for non-DV engineers to take part in the verification process.

Our experience with tool was very good, though we encountered certain challenges. Many of our reference models were inherited from previous projects, and as we used some of these, we found that the MATLAB code generation capability does not support all MATLAB functions. This meant that we needed to recode some of our legacy MATLAB code to support the code generators used to produce SV DPI or UVM components. In addition, because the MATLAB language allows dynamic resizing of arrays and dynamic casting from complex types to real types, we had to revise the MATLAB code so that it would correctly generate C code. In our experience, we found that almost every existing file needed minor modifications, which in some cases could be time consuming. Here are some of the most common code changes that were required to for our legacy MATLAB code:

1. Persistent variables were initialized before use.
2. Dynamic resizing is not supported and sometimes causes unpredictable behavior. Predefine all array sizes.
3. All the unused variables were removed from the MATLAB code.
4. Implicit casting between complex and real values is not allowed. Make use of "convert" utility in toolbox to explicitly switch types.
5. Operations like Num2Str and Transpose are not supported, so they must be replaced with alternate implementation.

Another pain point for us was the lack of customization support. The tool comes with certain default customization options like changing the timescale and location of the automatically generated files. We needed additional customization. For example, we want all our UVM components to be derived from our internal base components. The tool-generated code derives the components from their corresponding UVM base classes. We had to jump through multiple hoops to get what we wanted. Hopefully, in the later version of the tool, such issues will be addressed.

IV. Conclusion

After careful consideration, we decided to deploy a hybrid approach when it came to methodology choices. For legacy blocks, we used MATLAB compiler to produce DPI-C models, so we don't have to change the existing code. We lose the ease-of-use that we get from advanced methods, but we were fine with it as we do not anticipate lot of changes in those blocks. For the new blocks in development, we made recommendations to systems team to provide models that are compatible with code-generation tools. So, our new blocks can use fully automated UVM testbench generation.

The verification of DSP filters in a UVM environment lends itself well for using an automated MATLAB-based verification flow. Once the flow is established, people who are less experienced in UVM but well-versed in MATLAB and Simulink – namely system designers – can take part in the testbench development. It also provides a common language/platform for DSP and verification engineers to speak with each other, putting this method ahead of other automatic code generation frameworks. We were able to reuse the MATLAB models as our golden reference in the UVM environment. One obvious advantage here is that we did not have to produce SV version of reference models and thus we saved time.

Future work involves engaging with designers to utilize the Simulink schematic to automatically generate RTL for the DUT using HDL Coder™. This RTL can be used for prototyping in a FPGA, emulation box, or hybrid virtual prototype environment. Another interest is to leverage SystemC for control of the datapath and add the SystemC control path to the Simulink model. Additionally, we want to expand this work to other DSP blocks like CORDICs, FFTs, integrators, and the Viterbi algorithm. Lastly, we want to explore use of cosimulation to make the MATLAB model run lockstep with the design to enable feedback from the RTL into the MATLAB.

Acknowledgment

Avinash Lakshminarayana and Eric Jackowski would like to thank Alvin Tung and Yufei Zhang for their contribution to the code generation experiments.

REFERENCES

[1] Mentor Graphics Corporation. (2012 February 22). *Mentor Graphics Drives Broader Adoption of UVM* [Press release]. https://verificationacademy.com/news/mentor-graphics-drives-broader-adoption-uvm.

[2] Baird, Mike and Oden, Bob. (2016 March 1). *Slaying the UVM Reuse Dragon: Issues and Strategies for Achieving UVM Reuse* [Paper presentation]. 2016 Design and Verification Conference US, San Jose, CA, United States. https://verificationacademy.com/resources/technical-papers/slaying-the-uvm-reuse-dragon

[3] Salemi, Ray and Fitzpatrick, Tom. (2021 March 1-4). *Verification Learns a New Language – An IEEE 1800.2 Implementation* [Virtual conference presentation]. 2021 Design and Verification Conference US. https://verificationacademy.com/conferences/dvcon-2021/verification-learns-a-new-language_an-ieee-18002-python-implementation.

[4] Wagner, Philipp. (2019 June 11-14). Cocotb: Python-powered hardware verification [Conference presentation]. WOSH (Week of Open Source Hardware), Zürich, Switzerland https://www.fossi-foundation.org/wosh/#event-abstract-4.

[5] Mentor Graphics. (2017 June 6). *Mentor users in New Era of C++ Verification Signoff with New Catapult Tools and Solutions* [Press release]. https://www.plm.automation.siemens.com/global/en/our-story/newsroom/mentor-ushers-new-era-c-verification-signoff-new-cataplult/91677.

[6] Mentor Graphics Corporation. **Matlab/Integration**. Siemens Verification Academy. https://verificationacademy.com/cookbook/matlab/integration.

[7] Jia, Tao and Erickson, Jack. (2015 June). Reuse MATLAB Functions and Simulink Models in UVM Environments with Automatic SystemVerilog DPI Component Generation. Verification Horizons. https://verificationacademy.com/verification-horizons/june-2015-volume-11-issue-2/Reuse-MATLAB-Functions-and-Simulink-Models-in-UVM-Environments-with-Automatic-SystemVerilog-DPI-Component-Generation.

[8] UVMF MathWorks Integration Users Guide, Version 2021.3, retrieved from https://verificationacademy.com/topics/verification-methodology/uvm-framework.

[9] MathWorks, HDL Verifier documentation, https://www.mathworks.com/help/hdlverifier/uvm-generation.html

[10] Rishi, Pedram. (2021, May 26). UVM Simulation of MathWorks Designs at Block, Subsystem, and Chip Level [Virtual conference presentation]. Siemens Realize LIVE + User2User 2021. https://verificationacademy.com/sessions/uvm-simulation-of-mathworks-designs-at-block-subsystem-and-chip-level.

[11] MATLAB Coder Reference, https://www.mathworks.com/help/coder/

[12] MEX functions Reference. https://www.mathworks.com/help/matlab/call-mex-file-functions.html

[13] mxArray Reference. https://www.mathworks.com/help/matlab/apiref/mxarray.html

[14] Minxue Liang and Jie Chen, "A DSP-coprocessor architecture for image/video applications [coding/decoding]," Proceedings. 7th International Conference on Solid-State and Integrated Circuits Technology, 2004., 2004, pp. 1601-1604 vol.3, doi: 10.1109/ICSICT.2004.1435135.

[15] Layering in UVM, UVM Cookbook, http://verificationhorizons.verificationacademy.com/volume-7_issue-3/articles/stream/layering-in-uvm_vh-v7-i3.pdf

[16] Jeff Vance, Jeff Montesano, Kevin Johnston, “Verification Prowess with the UVM Harness”, Interface Techniques for Advanced Verification Strategies

[17] dpigen Reference. https://www.mathworks.com/help/hdlverifier/dpi-c-generation-for-matlab-code.html

[18] Simulink Test Point Reference. https://www.mathworks.com/help/simulink/ug/working-with-test-points.html

[19] uvmbuild Reference. https://www.mathworks.com/help/hdlverifier/uvm-generation.html

[20] Simulink.exportToTemplate https://www.mathworks.com/help/simulink/slref/simulink.exporttotemplate.html