How to leverage the power of MATLAB from Functional Verification Test Benches

Tom Richter - MathWorks
Agenda

• Verification Challenges
• Why MATLAB?
• Verification options
  • Co-simulation with MATLAB® on top
  • Co-simulation with HDL Simulator on top
  • SystemVerilog DPI component generation
  • UVM testbenches and components
• Questions
Verification Challenges

- The complex nature of the design
- Time and resource limitation
- Stimulus generation
- Golden reference model creation
- Verify that the deployed algorithm works the same as the reference
- Are all requirements implemented and tested?
Why MATLAB?

• Used in many domains such as signal processing, image processing and communications

• There, it is de facto standard language to explore, evaluate, and design architectures and algorithms

• MATLAB environment and its add-on products:
  • Intuitively learnable high-level programming language
  • Huge library of Functions, Classes, and System Objects
  • Diverse visualization options
  • Many apps, functions and objects for signal generation
Requirement Implementation and Testing

• MathWorks offers also tools to
  • associate requirements with MATLAB code and
  • plain-text external code, such as C code,

• This is achieved by
  • creating selection-based links with the Requirements Editor or
  • creating links programmatically at the MATLAB command line.

• Verify requirements with MATLAB code by creating links to MATLAB unit tests and running the tests

• View and edit links to code
Stimulus - Wireless Waveform Generator App

Certain MATLAB add-on come with rich features for waveform generation, full system simulation, and visualization.

- Waveform Generator App from 5G Toolbox offers:
  - Off-the-shelf waveforms: NR-TMs/FRCs
  - Custom downlink & uplink waveforms
  - CCDF measurement
  - Export waveform or generate code
Verification with MATLAB and Simulink

• Early verification and prototyping capabilities
  • FPGA-in-the-loop on FPGA and SoC boards
  • Automated insertion of probes into FPGA netlists for viewing and analysis
  • Hardware-based testing through read/write access to AXI registers and DDR

• Capabilities for functional verification
  • Co-simulation with HDL simulators from Siemens® EDA, Cadence®, and AMD®
  • SystemVerilog DPI components and testbenches from MATLAB and Simulink
  • Universal Verification Methodology (UVM) testbenches and components
Three ways to do Co-Simulation in MATLAB

There are three options to achieve co-simulation with MATLAB by using a MATLAB:

• System Object `hdlverifier.HDLCosimulation` or `hdlcosim`
• Callback function in combination with the instance `matlabtb`
• Callback function in combination with the instance `matlabcp`
MATLAB on Top - Workflow

Using a Co-Simulation System Object is considered to be a “MATLAB on top – workflow”:

• MATLAB will be the manager
  • Creates stimuli for the device under test (e.g.: manually written RTL)
  • Controls the compilation of the HDL files
  • Launches the HDL simulator tool (GUI or batch mode)
  • Sends stimuli to the HDL simulator continually (sample or frame based)
  • Receives simulated data from the HDL simulator
  • Compares, visualizes, or postprocesses the received data
Example – Verify a Viterbi Decoder (VVD)

The Viterbi decoding algorithm is still used in space communications, voice recognition, ..., and DNA sequencing. Assume:

• The algorithm was developed and tested in MATLAB
• VHDL code was manually written following the same specs
• You think about reusing the MATLAB code for verification
Example VVD – Reference Algorithm

- System design
- Write code (unstructured, floating point)
- Test algorithms
Example VVD – Refine Reference Algorithm

- Research on algorithm implementation
- Write code (unstructured, fixed-point)
- Test algorithm against reference
Example VVD – Streaming Algorithm

- Research on IP requirements
- Refine code (streaming, fixed-point)
- Test algorithm against unstructured fixed-point reference
Example VVD – Integration Decision

Depending on your final target (FPGA/ASIC/SoC) and preferences you may now decide how to resume in the integration process.

• Manually writing RTL code
• High-level synthesis
• Automatic code generation, e.g.: [HDL Coder™]:
  • Synthesizable VHDL, Verilog, and SystemVerilog code
  • Fully generic, target independent, readable, and traceable
  • Synthesizable SystemC code and testbench for Stratus™ HLS
  • Support for vertical products for AI, DSP, Comms, and Vision
Example VVD – Prepare Co-Simulation

The Cosimulation Wizard helps you with:

• Creating a MATLAB System Object or callback function from a template
• Preparing the tcl-commands for the HDL simulator

Adapt the generated files for your needs
Example VVD – Launch HDL Simulator

• Modify the System Object code if necessary

```plaintext
hDec = hdlcosim( 'InputSignals', {'/viterbi_block/In1','/viterbi_block/In2'}, ... 'OutputSignals', {'/viterbi_block/Out1'}, ... 'OutputSigned', false, ... 'OutputFractionLengths', 0, ... 'TCLPostSimulationCommand', 'echo "done";';, ... 'PreRunTime', {10, 'ns'}, ... 'Connection', {'SharedMemory'}, ... 'SampleTime', {10, 'ns'});

switch Simulator
    case 'ModelSim'
        hDec.TCLPreSimulationCommand = ... % Pre-simulation tcl-commands for HDL simulator
            'force /viterbi_block/clk_enable 1 0; force /viterbi_block/clk 0 0 ns, 1 5 ns -repeat 10 ns; force /viterbi_block/reset 1 0 ns, 0 0 ns;';
        case 'Xcellum'
        hDec.TCLPreSimulationCommand = ... % Pre-simulation tcl-commands for HDL simulator
            'force clk 0 0 -after 0ns 0'1" -after 5ns -repeat 10ns; force reset 0'1" -after 0ns 0'0" -after 0ns; force :clk_enable 0'1" -after 0ns';
    end

switch Simulator
    case 'ModelSim'
        vsim('tclstart','viterbi_cosimulation_tclcmds(’vsimmatlabsysobj’)),"runmode","Batch";
    case 'Xcellum'
        nclaunch('tclstart','viterbi_cosimulation_tclcmds(’hdlsimmatlabsysobj’));
    end
```

• Launch the HDL Simulator (that compiles the code and sets up the tool)
Example VVD – Run Co-Simulation

• Create a loop to run the whole transceiver chain

```matlab
receiveDelay = 58; % TracebackDepth + Pipeline Delay
hError = comm.ErrorRate('ComputationDelay',100,'ReceiveDelay',receiveDelay);
hScope.reset;
rng(1)
for counter = 1:20480/FrameSize
    data = randi([0 1],FrameSize,1);
    encodedData = hConEnc(data);
    modSignal = hMod(encodedData);
    receivedSignal = hChan(modSignal);
    demodSignalSD = hDemod(receivedSignal);
    quantizedValue = fi(4-demodSignalSD,0,3,0);
    input2 = quantizedValue(1:2:2*FrameSize);
    % Input Preparation
    receivedBits = hDec(input1,input2);
    errors = hError(data, double(receivedBits)); % Calculate Bit Error Rate
    hScope(data, receivedBits) % Show transmitted and received random sequence in a Scope
    hConstDist(modSignal, receivedSignal) % Show constellation of modulated transmitted and received signal
end
```

• Compare the results with the simulation reference

```matlab
fprintf('Bit Error Rate is %d
', errors(1))
```

Bit Error Rate is 3.493751e-03
HDL Simulator on Top - Workflow

Using callback functions in MATLAB and instances like `matlabtb` or `matlabcp` made known to the HDL Simulator we call the workflow “HDL Simulator on top”:

- MATLAB just works in the background
  - It needs to run the HDL Link MATLAB server using shared memory inter-process communication

- HDL Simulator controls the whole simulation
  - Triggers the callback function in MATLAB
  - Sends/requests data to/from MATLAB
Component vs. Testbench

<table>
<thead>
<tr>
<th>matlabcp</th>
<th>matlabtb</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test your HDL code in an HDL Simulator but a certain component which does not yet exists in HDL will be simulated through MATLAB. Testbench exists in EDA tool.</td>
<td>Drive stimuli from MATLAB to an HDL component and send the output back to MATLAB. Testbench exists in MATLAB but is controlled (called) from the HDL Simulator.</td>
</tr>
<tr>
<td>It sends output from HDL Simulator to MATLAB and receives input to HDL from Simulator MATLAB</td>
<td>Acquire input to HDL Simulator from MATLAB and send back output from HDL Simulator to MATLAB</td>
</tr>
</tbody>
</table>

Implement Filter Component of Oscillator in MATLAB

Cosimulation for Testing Filter Component Using MATLAB Test Bench
Example – MATLAB Filter Component (MFC)

The following steps are required to create and test a MATLAB component that can be used from an HDL Simulator:

• Generate a Callback Template
• Integrate your MATLAB function into the template files
• Modify the generated tcl-file
• Start the HDL Link MATLAB server
• Run existing testbench in HDL Simulator
Example (MFC) – Cosimulation Wizard

The Cosimulation Wizard helps again with:

• Creating a callback function from a template
• Preparing the tcl-commands
Example (MFC) – Callback Template

Two options for MATLAB component function writing:

- Using the HDL Instance Object (used for the template)
  - MATLAB function prototype: `function matlabFuncName(obj)`
  - Object field examples:
    - `obj.portvalues`, `obj.tnow`,
      `obj.userdata`, `obj.simstatus`,
    - `obj.argument`, `obj.portinfo`
  - TCL command: `matlabcp hdlInstanceName -mfunc matlabFuncName -use_instance_obj`

- Using Port Information
  - MATLAB function prototype:
    `function [oport, tnext] = matlabFuncName(iport, tnow, portinfo)`
  - TCL command: `matlabcp hdlInstanceName -mfunc matlabFuncName`
Example (MFC) – Run Co-Simulation

Run the HDL Link MATLAB server for communication between the HDL simulator and MATLAB:

• In an open MATLAB session run:
  >> hdldaemon

• You can also open a MATLAB session in batch mode:
  • Windows cmd: `matlab -nodesktop -r "hdldaemon"`
  • Linux shell: `xterm -e "matlab -nodesktop -r "hdldaemon"" &`

• Start the cosimulation
  • Windows cmd: `vsim -do qcommands_osc_w.tcl`
  • Linux shell: `vsim -do qcommands_osc_l.tcl`

(Here ModelSim was used)
SystemVerilog – DPI Component Generation

- Export a MATLAB function as a component with a direct programming interface (DPI) for use in a SystemVerilog simulation
  - MATLAB Coder is used for generating C code with a DPI wrapper
  - The DPI wrapper communicates with a SystemVerilog interface function
  - The SystemVerilog component can be used within a SystemVerilog testbench

- Get DPI component shared library for:
  - Linux (.so) or Windows (.dll)

- Templates to influence component
  - Sequential – sequential design, with registers
  - Combinational – with no registers

- Different choices for port data types
Example – Sine Wave Generator (SWG)

The following steps are required to generate a SystemVerilog DPI component:

• Prepare a MATLAB function for code generation
• Create a MATLAB testbench file (optional)
• Define and set a configuration object (optional)
• Generate the component using `dpigen`
• Verify your component with a generated testbench (optional)
Example (SWG) – Prepare Function

Before generating the component from MATLAB code ensure to:

- Initialize variables and define them on all execution paths
- Define static variables as persistent (states, registers)
- Avoid dynamic memory allocation for efficiency
  - Rather use fixed-size arrays and variable-size arrays (size < threshold)
- To identify issues:
  - Use `%#codegen` pragma to instruct the MATLAB Code Analyzer to help with finding and fixing violations
  - Run the Code Screener
Example (SWG) – MATLAB Testbench

A MATLAB testbench is an optional file. However, it helps with:

• Defining port
  • data types,
  • sizes, and
  • complexity
• Testing and debugging the MATLAB function
• Verifying the generated component
Example (SWG) – Configuration

• Create a `svdpiConfiguration` object
  • The default configuration points to the templates for a sequential module (component and testbench template)
  • Change component and testbench name, and SystemVerilog port types

```python
svcfg = svdpiConfiguration

svcfg =
svdipiConfiguration with properties:

  ComponentKind: 'sequential-module'
  CoderConfiguration: [xl coder.EmbeddedCodeConfig]
  ComponentTypeName: ''
  TestBenchTypeName: ''
  TemplateDictionary: []
  PortGroups: []
  ComponentTemplateFiles:
    ['/mathworks/devel/bat/Bdoc23a/build/matlab/toolbox/hd1verifier/dpigenerator/rtw/SequentialModuleML.svt']
  TestBenchTemplateFiles:
    ['/mathworks/devel/bat/Bdoc23a/build/matlab/toolbox/hd1verifier/dpigenerator/rtw/SequentialTestBenchML.svt']
```
Example (SWG) – Component Generation

Use the `dpigen` function to generate the component artefacts. You can:

• Define the MATLAB function and optionally the MATLAB testbench
• Provide sample data for input arguments
• Set a configuration object or separately:
  • Custom include files
  • Compiler options
  • SystemVerilog Port data types
  • Component templates
• Generate and launch a code generation report

```
dpigen sineWaveGen -testbench sineWaveGen_tb -args {0,0} -config svcfg
```
Example (SWG) – Package Required Files

With the **packNGO** function you can zip all the required Files.

- Easily share the generated artefacts
- Get only files necessary

```
packNGO(buildInfo)
```
Example (SWG) – Verify Component

When generating the component including testbench you can directly verify it with an HDL simulator

• A *_tb.sv file is generated together with data files for stimuli and expected output
• tcl-scripts provided for compiling and running the testbench in:
  • ModelSim™/QuestaSim™
  • Cadence® Xcelium™
  • Synopsys® VCS™
  • AMD® Vivado™
UVM Component Generation from MATLAB

In addition of generating a general SystemVerilog DPI component, further template files enable also the generation of UVM components.

• Can integrate into a full UVM testbench
• Templates for component and testbench:
  • Predictor
  • Sequence
  • Scoreboard
UVM – Predictor component

The predictor template generates a UVM predictor module that has:

• an export that inputs a predictor transaction, and
• an analysis port that outputs a scoreboard transaction.

• The predictor template includes these variables:
  • ComponentTypeName
  • TestBenchTypeName
  • ComponentPackageTypeName
  • InputTransTypeName
  • OutputTransTypeName
UVM – Sequence component

The sequence template generates a UVM sequence module.

• It includes these variables:
  • ComponentTypeName
  • TestBenchTypeName
  • ComponentPackageTypeName
  • SequenceTransTypeName
  • SequencerTypeName
  • SequenceCount
  • SequenceFlushCount
UVM – Scoreboard component

The scoreboard template generates a UVM scoreboard module that has two exports that input a scoreboard transaction.

• The scoreboard template includes these variables:
  • ComponentTypeName
  • TestBenchTypeName
  • ComponentPackageTypeName
  • InputTransTypeName
  • OutputTransTypeName
  • ConfigObjTypeName

• Map ports to port groups by using the addPortGroup object function.
Example – Sine Wave Check (SWC)

The following steps are required to generate a UVM component:

• Prepare a MATLAB scoreboard function
• Create a MATLAB testbench file (optional)
• Define a configuration object (required)
• Provide UVM settings to the `cfgobj`
• Specify which port belongs to which port group
• Generate the component using `dpigen`
• Verify your component with a generated testbench (optional)
Example (SWC) – Scoreboard Function

Before generating the component from MATLAB code do the preparation steps necessary for SystemVerilog DPI components.

• Write a scoreboard function that compares
  • output of the DUT with
  • output of a golden reference

• Inputs are from Monitor and Predictor. Configurations are inputs, too.

```matlab
function [mismatch1, mismatch2] = sineWaveCheck(in1FromMon, in2FromMon, inFromPred, amp1, amp2)
    %inFromMon: sine wave from DUT
    %inFromPred: normalized sine wave from golden reference (predictor)
    %amp: amplitude of the sine wave from DUT
    mismatch1 = (in1FromMon ~= inFromPred*amp1);
    mismatch2 = (in2FromMon ~= inFromPred*amp2);
end
```
Example (SWC) – Configuration for UVM

- Create a `svdipiConfiguration` object and then change:
  - Change the kind of the component, and
  - Optionally the component name
- Check and optionally change through the Template Dictionary
  - Name of component package file
  - Name of input/output transition
  - Name of configuration object
- Specify which port belongs to which port group

```plaintext
clear svcfg;
svcfg=svdipiConfiguration;
svcfg.ComponentKind = 'uvm-scoreboard'
```
```plaintext
svcfg.TemplateDictionary = { 'InputTransTypeName', 'sineWaveTrans' };
addPortGroup(svcfg, 'MONITOR_INPUTS', { 'in1FromMon', 'in2FromMon' });
addPortGroup(svcfg, 'PREDICTOR_INPUTS', { 'inFromPred' });
addPortGroup(svcfg, 'CONFIG_OBJECT_INPUTS', { 'amp1', 'amp2' });
```
Example (SWC) – Generate and Verify

Use the **dpigen** function to generate the component artefacts.

```plaintext
dpigen sineWaveCheck -testbench sineWaveCheck_tb -args {zeros(1,100),zeros(1,100),zeros(1,100),0,0} -config svcfg
```

Test the UVM component using the generated SystemVerilog testbench and tcl-file.
FPGA/ASIC/SoC Workflow Adoption

Modelling Adoption

MATLAB
Floating-point, Unstructured

SV Export, or MATLAB Cosim*

MATLAB
Fixed-point, Unstructured

SV Export, or MATLAB Cosim*

MATLAB & Simulink
Fixed-point, Streaming IP Architecture

SV Export, or Simulink Cosim*

Rapid Prototype on FPGA

MATLAB & Simulink
Fixed-point, Streaming System Architecture

SV Export, or Simulink Cosim*

HDL for FPGA, SoC, or ASIC

MATLAB & Simulink
Model V&V in Simulink
Fixed-point Streaming System Architecture

SV Export, or Simulink Cosim*

HDL for High Integrity FPGA and ASIC

EDA Verification Integration Adoption

* Cosim with EDA simulators (Siemens, Cadence) or FPGA-in-the-Loop (FIL)

Code Generation Adoption
Questions