Co-Simulating Matlab/Simulink Models in a UVM Environment

Neal Okumura
Paul Yue
Raytheon Company
2000 E El Segundo Blvd
El Segundo, CA 90245

Glenn Richards
Mentor Graphics Corporation
8005 SW Boeckman Rd
Wilsonville, OR 97070

I. Introduction

Linux based multi-core computing farms enable the execution of multiple test scenarios, and facilitate the reduction in simulation time. However, the process to verify a design with a behavioral model is still a serial process, and requires a separate comparison step between the two simulation results. The verification process requires the generation of test patterns and the expected results with bit accurate behavioral models like C/C++ and Matlab, then running the Register Transfer Level (RTL) simulation, and finally comparing the RTL results to the behavioral model.

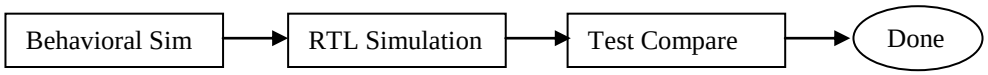

Figure 1. Verification process with behavioral and RTL simulations.

This becomes problematic when long run-time simulations are needed to fully test the functionality of the design or when the design size starts to grow large. The amount of data results to be compared can be massive and prone to human error. The verification process is segmented and serial, which may add unnecessary delays and inaccuracies.

To address the problem of RTL verification completeness and accuracy, the Universal Verification Methodology (UVM) RTL simulation environment can be coupled with the behavioral simulation environment. In this paper Mentor Graphics Questa and Matlab/Simulink are used as the simulation tools. By coupling the simulators the overall simulation time may be reduced, and the accuracy and time spent comparing the behavioral model to the RTL will be improved. UVM techniques can also then be used to reduce the number of test scenarios, the simulation effectiveness may be improved and provide a more thorough excitation of our designs, and the simulation will provide an automated method of collecting Verification Metrics.

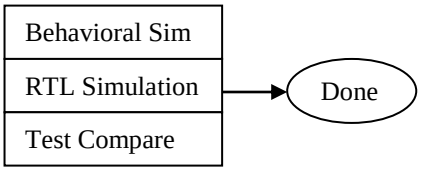

Figure 2. Improved verification process with behavioral and RTL simulations running together in one simulation.

II. FINDINGS

A. *Co-Simulation*

Matlab and Simulink models are run in the Matlab software independent from the design and the UVM environment. Tools that run the UVM environment cannot directly run the Matlab software. With the use of the Matlab engine and SystemVerilog DPI, the UVM environment can make a subroutine process call to bring up an embedded Matlab environment within UVM.

B. *Matlab Engine*

Matlab has a library of C functions called the Matlab engine library that are used to start and end processes, send data to and from, and send commands to be processed in Matlab. The functions in this API include opening and closing the Matlab engine, putting and getting variables, and sending strings to the Matlab console. These functions are contained in a file from Matlab called engine.h. The engine.h file should be included in the source file that calls out the Matlab engine functions [1]. An example of a Matlab engine function call in source code is shown below. The example shown is included with Matlab in its install at: ~/extern/examples/eng_mat/engdemo.c.

```
#include “engine.h”

main () {
Engine *ep;
ep = engOpen(‘’)
}
```

Figure 3. Creating Matlab Engine functions in C source code.

The designer will create a C source file that contains the desired Matlab functions. These functions will then be imported into UVM.

C. *SystemVerilog DPI*

SystemVerilog has a built in capability to interface to external languages by using an import function through the Direct Programming Interface (DPI). A function can be written in another language like C, and then imported into the SystemVerilog source through DPI. Once the function is imported, the SystemVerilog source can make a function call to it as normal. An example of a SystemVerilog source importing C function through DPI is seen below, in Fig 4:

```
import “DPI-C” function void engOpen();
```

Figure 4. Importing the Matlab Engine function to SystemVerilog through DPI, SystemVerilog code.

At compile time, the imported DPI functions must be compiled into a DPI header file. In Questasim this is done by adding the –dpiheader flag to the file being compiled. All files for the supporting functions should be contained within a SystemVerilog package. At compile time, the –dpiheader flag is added to the compile command [2]:

```
vlog example_engine_pkg.sv –dpiheader <filename> -sv
```

Figure 5. Example command in Questasim to create the DPI header file. Example package file containing all supporting function files is example_engine_pkg.sv

An engine class can then be declared to create an engine object that contains all supporting functions imported through DPI. This engine class file should be included in the package in Fig 5. An example of declaring an engine class is shown below:

```
class engine_example;
        function void engOpen ();
endclass : engine_example
```

Figure 6. Creating a Matlab engine class within SystemVerilog code

The Matlab engine object can now be created in the module that will use the engine. The SystemVerilog source that will use the Matlab Engine needs to import the engine package file that was compiled with the DPI header. Usually the top level module of the environment should contain the Matlab commands and import the engine package that was compiled:

```
import example_engine_pkg::*;
```

Figure 7. Importing the compiled Matlab Engine into a SystemVerilog module, SystemVerilog code.

With the imported Matlab Engine package declared, the Matlab engine functions can now be called within the module. The module should open a Matlab engine, run the necessary Matlab functions, and then close the engine when complete.

```
engine_example eng;

initial begin
        eng = new();
        eng.engOpen();
end
```

Figure 8. Calling Matlab engine functions within SystemVerilog code.

D. Simulink Model

Simulink is an application integrated in Matlab. Once a Matlab engine has been opened in UVM, a Simulink model can be loaded. The Matlab engine cannot directly command Simulink, but by using commands in the Matlab console a Simulink model can be loaded and started [3].

The function engEvalString() is a function included in the Matlab engine library that sends a command directly to the Matlab console. When this command is imported using DPI, Matlab console commands can be made from UVM. Using the Matlab console command to open a Simulink model, open_system('<Simulink model>'), in the engEvalString() function can be used to open the model:

engEvalString("open_system('<Simulink model>');");

Figure 9. Using the Matlab Engine string function, engEvalString(), to make a call to the Matlab console, SystemVerilog code.

This will open the Simulink model. To start the Simulink simulation, again use the engEvalString() function to send a command to the Matlab console to start the Simulink model.

Through the imported Matlab Engine string function, we can now make direct commands to the Matlab console from UVM and control our Simulink models.

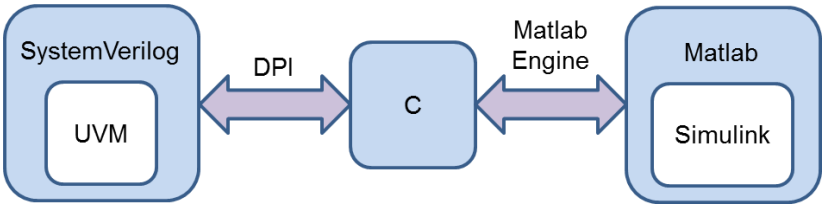

Figure 10. Relation between SystemVerilog and Matlab.

Due to the fact that the Matlab engine does not directly interface with Simulink models data cannot be directly transferred between the model and the UVM environment. To do this the Simulink model needs some modification. In the Simulink model, monitors need to be added to the points that will be compared to the DUT. This will allow bringing arrays of data to the Matlab workspace. Once the data is in the Matlab workspace this can now be transferred to the UVM environment. The Matlab engine function engGetVariable() gets a variable from the Matlab workspace. When this command is imported using DPI, the UVM environment can get the Simulink monitor data from the Matlab workspace.

Likewise by using this same method the UVM environment can send data to the Simulink model. When UVM generates the stimulus for the simulation constrained random variables can be used. The UVM environment generates transaction packets with constrained random variables for stimulus. These same packets can be sent to the Matlab engine, and then into the Simulink model. This will also provide further testing on the Simulink model that Matlab alone could not do.

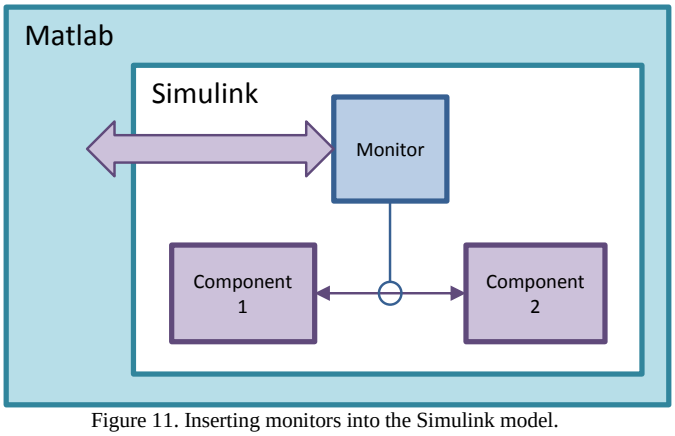

Figure 11. Inserting monitors into the Simulink model.

E. UVM Environment

The Matlab engine object and commands are contained in the top level testbench module. The Matlab engine is opened, closed, and commanded from the top module. From there the data is ported to the UVM environment for comparison and scoreboarding.

F. Coverage Based Verification

This setup allows a single iteration of the Simulink model. The UVM environment starts the Matlab engine as a subprocess and the Simulink simulation runs and transfers data to the Matlab workspace where it is transferred to the UVM environment as a variable. From there the UVM verification can run and compare against the Matlab results. Likewise, the UVM environment can run for a set amount of time, and then transfer the stimulus to the Simulink model to run. This allows for a test case that runs for a set amount of time, and can be directed or random. Coverage goals can be applied for coverage-based verification. With coverage goals applied, the test cases can be ranked and applied to the verification plan as necessary.

G. Closing Goals on Coverage Based Verification

To run a test case until all coverage goals have been met the UVM environment runs indefinitely. When a Matlab engine is embedded in the environment the model also needs to be able to run indefinitely so that it can continue to run until the coverage goals have been met. Additional modification to the Simulink model is required to make this happen. A timing correlation needs to be made between the Simulink model and the DUT in the UVM environment. A timing counter in the Simulink model needs to match a time window in UVM. This time window will have a given amount of data to exchange between the 2 environments and should be designed accordingly. When the timing counter in Simulink reaches a given time it should trigger an assertion to pause the Simulink model. Using the Matlab console command upon an assertion will pause the Simulink simulation.

Once the Simulink simulation is paused, the data needs to be ported from the Simulink monitors to the Matalab workspace. This can be done in the Matlab console via a DPI function with a pause. The data can then be transferred to the UVM environment and compared against the DUT after running for the same time period as the Simulink model. Once this is done the Simulink model can then be resumed with a console command to continue the Simulink simulation.

This iteration can be run over and over indefinitely which allows the UVM environment to run until the coverage goals have been completed.

H. Assertion Based Verification

By embedding the Matlab/Simulink model in the UVM environment, the model data can be directly compared to the DUT data and used as a golden model. Assertions can also be created within the UVM environment to further verify the proper functionality of the design.

I. Benefits

There are many benefits to creating a co-simulation with Matlab and UVM. The co-simulation does a direct data comparison between the Matlab model and the design. This saves time from analyzing data from both environments and doing a manual comparison and assessment.

In algorithmic designs, sometimes stimulus can be created more easily in Matlab than in UVM. By co-simulating them together, the same Matlab stimulus can be used for the DUT in UVM. This saves time creating stimulus for UVM.

The stimulus created for Matlab is a known pattern. With constrained random variables UVM has the capability of creating random stimulus with coverage goals. This creates a testcase of random and repeatable stimulus.

III. Conclusion

By importing the Matlab Engine into UVM using DPI, an embedded Matlab environment can be created within UVM. This allows a co-simulation of a Matlab/Simulink model and the DUT within UVM. This co-simulation gives the user many benefits and adds capabilities that are not available with the two environments by themselves. Co-simulation also saves on the overall design and verification time, and improves the accuracy of comparing the behavioral model to the RTL code.

References

[1] The MathWorks, Inc., *engdemo.c.* (Version 1.8.4.4) [Computer program]. Available in Matlab software at *matlabroot*\extern\examples\eng_mat (Accessed Matlab version 2012a)

[2] Mentor Graphics. *Questasim.* (Version 10.1b) [Computer software]. N.p., 26 Apr. 2012.

[3] The MathWorks, Inc., *Matlab.* (Version 2012a) [Computer software]. N.p., 9 Feb. 2012.