FEATURED IN THIS ISSUE:

Package Virtual Platforms with real software to create functional mock-up units that can model and verify complex digital, analog, and mechanical systems like an automotive ADAS.

Examine a progression of increasingly complex UVM sequences to see that they’re really “just code” and see how Questa® can help visualize transactions to make debug easier.

Learn how Questa® inFact enables Portable Stimulus (PSS) so you can focus on what you want to verify instead of worrying about how to model it in SystemVerilog/UVM.

Identify good “fishing spots” to focus formal verification on leveraging simulation traces to explore interesting states that will help uncover deep-state bugs.

Learn about a complete clock-domain crossing methodology that overcomes the challenges you may encounter in verifying designs with multiple clock domains.

Discover how to define complex low-level sequences of register- and pin-manipulation operations and auto-generate correct-by-construction exec block definitions for a variety of target languages used in your PSS model.

Find practical advice that goes beyond the tutorial provided in the UVMF installation to illustrate how to extend the top-level tests generated by UVMF to allow additional customization, adding application-specific content to verify your design.

Is Your Verification in Jeopardy?

By Tom Fitzpatrick, Editor and Verification Technologist

I am a big fan of the television show Jeopardy! and, like many of you, I have been intrigued by James Holzhauer, who, as of this writing, is now the second-winningest contestant in the show’s history. During his current 22-game winning streak (Ken Jennings won 74 consecutive games in 2004), Holzhauer has amassed $1,691,008 in winnings (vs. $2,520,700 for Jennings), including the top 11 single-game totals in the show’s history. For those of you not familiar with the show, it is a general-knowledge quiz show consisting of two rounds in which players choose from six clue categories, each of which has five clues, which are assigned increasing dollar values from top to bottom. Players are awarded the dollar amount for a correct response, and given the next choice of clue, or they have that amount deducted for an incorrect response. The third round, called “Final Jeopardy,” consists of a single question on which contestants can wager as much as they wish.

Holzhauer has achieved his amazing success through two complementary approaches. The first is his analytics-based approach to the game. Instead of starting at the top (lowest-value) clue in a category and proceeding down through the same category, as most players tend to do, he starts with the bottom higher-value clue of a given category and proceeds across the bottom of the board to accumulate large amounts of money very quickly. Since the “Daily Double” squares (which allow the contestant to bet as much money as they wish) tend to be towards the bottom of the board, Holzhauer often is able to further enrich his total before his opponents have really gotten started. Of course, the other key to his success is that he has answered 97% of the questions correctly.

The only comparable Jeopardy! performance I can recall is watching Bill Murray in the movie Groundhog Day where he knows all the answers because he has seen the same episode over and over again and has obviously memorized the answers. I think both of these examples show that just doing things the way you’ve always done them is not the path to success, especially when your competition is changing the game.

So, with this thought in mind, we’ve assembled a great DAC edition of Verification Horizons for you, with each article showing how you can take a new approach to some aspect of functional verification.

We start this issue with “SystemC FMU for Verification of Advanced Driver Assistance Systems” from my Mentor colleagues, Keroles Khalil and Magdy A. El-Moursy, over in Egypt. They discuss how to package Virtual
Platforms with real software to create functional mock-up units that model pieces of a car. By putting these pieces together using a standard functional mock-up interface, you can assemble a system-of-systems to model and verify complex digital, analog, and mechanical systems, including things as complex as a complete ADAS system.

After that, Rich Edelman will show us how to have “Fun with UVM Sequences – Coding and Debugging.” In his own inimitable way, Rich walks us through a progression of increasingly complex (and previously intimidating) sequences and shows us that they’re all “just code” and nothing to be afraid of. He also shows us how Questa® can help you visualize your transactions to make debug easier.

Next we continue our Portable Stimulus series by Matthew Ballance with “Creating Test the PSS Way in SystemVerilog.” In it we learn how Questa® inFact enables Portable Stimulus (PSS) so you can focus on what you want to verify instead of worrying about how to model it in SystemVerilog/UVM. We’ve packaged some of the critical Portable Stimulus capabilities into PSS Apps that can read in existing SystemVerilog classes and covergroups to give you better verification productivity by getting the most out of your existing testbenches.

Next, we have a couple of articles from our formal verification team. In “Formal Bug Hunting with ‘River Fishing’ Techniques” from my Mentor colleagues Mark Eslinger and Ping Yeung, we see how to identify good “fishing spots” to focus formal verification to find more bugs. As designs get more complex, trying to uncover bugs by starting formal analysis at time 0 gets more difficult and less efficient, so instead we leverage simulation traces to explore interesting states that will help us uncover deep-state bugs.

We follow that with “Don’t Forget the Protocol! A CDC Protocol Methodology to Avoid Bugs in Silicon,” where we learn a complete clock-domain crossing methodology that overcomes the challenges you may encounter in verifying designs with multiple clock-domains.

The first article in our Partners’ Corner is from our friends at Agnisys. They discuss “Auto-Generating Implementation-Level Sequences for PSS.” If you’ve been following our recent Portable Stimulus articles, you’re familiar with the new PSS standard that lets you create an abstract activity graph to define the scheduling relationships between critical actions that you want to execute. The implementations of those actions are provided using what PSS calls exec blocks, which use either code templates or imported method calls in the target language. Agnisys has developed a tool that will let you define complex low-level sequences of register- and pin-manipulation operations and auto-generate correct-by-construction exec block definitions for a variety of target languages that you can use in your PSS model.

We wind up this time with a tutorial-level article from our new friends at DesignLinx Solutions, “UVMF, Beyond the ALU Generator Tutorial Extending Actual Test Control of the DUT Inputs.” The UVM Framework (UVMF) is a tool you can use to auto-generate a UVM verification environment, which you can then customize to add application-specific content to verify your design. This article provides some practical advice that goes beyond the tutorial provided in the UVMF installation to illustrate how to extend the top-level tests generated by UVMF to allow additional customization.
By the time you’re reading this, DAC will actually be here, and James Holzhauer may (or may not) still be winning at Jeopardy! But if you apply the lessons from these great articles, I know that you’ll be winning your verification battles soon. Please stop by the Mentor (#334) and Verification Academy (#617) booths at DAC and say hi. I’d love to hear what your verification challenges are and talk with you about how we might be able to get you out of “jeopardy.”

Respectfully submitted,
Tom Fitzpatrick
Editor, Verification Horizons

CONTENTS

Page 4: SystemC FMU for Verification of Advanced Driver Assistance Systems
by Keroles Khalil and Magdy A. El-Moursy – Mentor, A Siemens Business

Page 12: Fun with UVM Sequences
– Coding and Debugging
by Rich Edelman – Mentor, A Siemens Business

Page 21: Creating Tests
the PSS Way in SystemVerilog
by Matthew Ballance – Mentor, A Siemens Business

Page 27: Formal Bug Hunting
with “River Fishing” Techniques
by Mark Eslinger and Ping Yeung – Mentor, A Siemens Business

Page 34: Don’t Forget the Protocol! A CDC Protocol Methodology to Avoid Bugs in Silicon
by Sukriti Bisht, Sulabh Kumar Khare, and Ashish Hari, and Kurt Takara – Mentor, A Siemens Business

Partners’ Corner:

Page 42: Auto-Generating Implementation-Level Sequences for PSS
by Amanjyot Kaur and Louie De Luna – Agnisys

Page 50: UVMF, Beyond the ALU Generator Tutorial Extending Actual Test Control of the DUT Inputs
by George Stevens - DesignLinx Solutions
An integrated framework to simulate electronic systems (including digital and analog devices) with the mechanical parts of a heterogeneous automotive system is presented. The electronic system, consisting of many electronic control units (ECUs), is modeled to simulate the mechatronic system functionality. The recently developed functional mock-up interface standard approach is used to create a model for a complex cyber-physical automotive system. The framework simulates real systems, including the hardware (HW) and the software (SW) to run on the virtual ECUs. It allows co-development of the automotive system SW and HW while the mechanical system is in the loop. Hardware and software debugging is demonstrated using the developed methodology. The development cycle for the automotive mechatronic system can be greatly shortened using the proposed framework.

INTRODUCTION

Autonomous driving (AD) requires sophisticated electronic systems. The designs of today’s automotive electronic systems contain many system-on-chips (SoC)\(^1\),\(^2\). Deep verification is required to ensure that there are no bugs or risks of failure. The safety of advanced driver assistance systems (ADAS) can not be guaranteed without verification of the whole system, including thermal, mechanical,
simulation, such as SystemC/TLM2 [1]. In a hybrid automotive system, synergy among different tools and models crossing domains (system of systems), including virtual platforms, plant simulators, digital simulators, HW emulators and mechanical simulators, is required. Virtual platforms model ECUs (i.e., virtual ECU or VECU), including processors, HW peripherals, and controllers to run automotive software stacks. [8-12] ADAS VECUs have multiple sensors on board, of which the readings are combined to obtain a better perception of the actual situation. In this article the presented framework is used to verify both autonomous and non-autonomous VECUs. A VECU which acts as an ADAS ECU has AUTOSAR SW [13] that runs on top of it. The VECU is connected with many types of sensors and actuators through the functional mock-up interface (FMI) standard [14] as shown in Figure 1 on the opposite page.

SystemC/TLM VPs are generated using the Vista™ tool [15] and is connected with sensors/actuators created by MATLAB® [16] and TASS® Prescan. [17] TASS gives the capability to develop dangerous scenarios that can occur in the physical surrounding of a vehicle. These simulated scenarios allow engineers to have more time to focus on real system design issues at an early design stage. Using the proposed framework, the developer can verify real code on the VECU and deal with many sensors and actuators in different scenarios and study the effects of using ADAS in such scenarios. The presented methodology integrates VECU as a functional mock-up unit (FMU) using FMI.

The work in [1] presents an approach to create VPs as FMUs. The VP is shipped as an FMU shared object with an FMI model description XML in the FMU file. The FMI master loads the VP DLL. The proposed mechanism in this article is different in that it depends on creating an FMU wrapper which can be loaded by the FMI master with the VP through an IPC mechanism. The I/O interconnects of the VP are controlled by FMI V.2.0 co-simulation APIs triggered by the FMI master. I/O changes are propagated in an event-driven execution mechanism that is compatible with the nature of the SystemC/TLM based VP. With this mechanism, portability is high compared to the mechanism described in [1], as the VP can be modified and compiled without changing the FMU part. The remainder of the article is organized as follows: the design methodology using the proposed FMU is presented in section II; a case study of a full automotive system is provided in section III; and conclusions are presented in section IV.

**THE DEVELOPED METHODOLOGY**

In order to wrap a SystemC virtual platform as an FMU, two wrappers are used. One is inside the VP, the internal FMU control (IFC), and the other is outside, the external FMU wrapper (EFW). The IFC is the TLM SystemC model responsible for communicating with the EFW and synchronizing the VP with the other FMUs (could be electrical or mechanical). The IFC is connected with the EFW through an IPC mechanism. The EFW consists of XML and DLL. It can be parsed and loaded by the external FMI master inside another simulation tool. FMUs are assumed to be compliant with the FMI V.2.0 co-simulation standard. Also, it is assumed that the FMUs are provided as a pre-compiled shared library (*.DLL on Windows or *.so on Linux systems). Runtime library dependencies are assumed to be available on the same host on which the VP is running. The EFW can be configured to have any number of input and output pins. Analog ports are represented as REAL numbers and digital pins are represented as INTEGER numbers. Each pin has a unique reference value ($V_{\text{reference}}$). EFW configuration can be set in the XML file. IFC has the same configuration of EFW.

The EFW XML has a dynamic array of $S_{\text{FMU\_PORT}}$ structure for each pin, as shown in Figure 2 on the following page. The $S_{\text{FMU\_PORT}}$ structure defines the $\text{FMU\_Container\_Value}$, which represents the value to be sent/received from/to the VP to/from the FMI master. The $\text{FMU\_Container\_Type}$ represents the data type (REAL or INTEGER). $V_{\text{reference}}$ represents the reference value for the pin, which is defined in the XML file. If the pin value is updated by any side (master or slave), the updated value is set. The pin direction in the
The XML file identifies if the pin is input or output. C_socket represents the virtual socket, which is mapped to the pin, as shown in Figure 3, above. The default initialization value of \( V_{\text{reference}} \) is -1. Initialization is performed once the FMI master begins calling the FMU initialization APIs.

Once the FMI master sends an INTEGR value to the FMU, it calls the fmi2SetInteger API; then the EFW searches for all the S_FMU_PORTS that have the matched reference value with input direction. EFW sets the FMUContainer.Value with the value that is sent by other FMUs. If the XML \( V_{\text{reference}} \) does not exist, then the \( V_{\text{reference}} \) is set to -1. Finally, the value is sent to the corresponding virtual socket, as shown in Figure 3.

On the other side, IFC handles the reception and transmission of the pin values by the mapper sub-block. EFW sends a string using the IPC virtual socket. The string consists of five data values separated by a semicolon (pin value, data type, variable reference, updated flag, and direction). The mapper inside the IFC parses the string and issues a transaction on the general purpose input output (GPIO) TLM pin, which matches the \( V_{\text{reference}} \) in the XML, as shown in Figure 4, opposite. Each TLM pin has a unique \( V_{\text{reference}} \).
Similar behavior happens when the GPIO issues a transaction to send data on the GPIO pin. The mapper sends the data value to the EFW on the corresponding virtual socket. Once the FMI master reads data from the FMU, it calls the `Fmi2getint` API, and the saved value is returned to the FMI master. The exchanged data between the FMI master external tool and the VP, which is wrapped as an FMU slave, is shown in Figure 5, above. A case study of many VECUs communicating together (with one of them, ADAS VECU, acting as an FMU connected to a Matlab FMI master) is presented in the following section.

CASE STUDY

An ADAS of an autonomous vehicle consisting of four VECUs connected to a CAN bus is used in the presented case study to demonstrate the use of VP as an FMU, as shown in Figure 6. A dashboard VECU is modeled using CANoe® [18] for packet generation and monitoring. Different inputs (such as pedal/angle, brake/angle, shift, and stop engine) and outputs (monitor speed/RPM) are stimulated and observed through the dashboard. The engine control VECU is modeled as a PowerPC Virtual Platform using Vista Architect® [15] to run the AUTOSAR stack/application. The engine control VECU reads input combinations from the dashboard and sends command/pedal/brake angle to the transmission controller VECU. The transmission controller VECU is modeled as a virtual SoC (based on the ARM® CortexM3) that runs a bare metal application. It reads pedal angle from the engine controller VECU and calculates the RPM/speed. The braking system is modeled using Simcenter Amesim® [19] with mechatronic components, exported as a slave FMU and connected to a Vista FMI master running on the host machine. The FMU input is a brake force which comes from the digital-to-analog converter (DAC) model inside the engine.

Figure 5: Data exchange between the VECU pins and the FMI master in an external tool

Figure 6: Advanced emergency braking system controlled by ADAS VECU (case study)
control VECU, and the output is the force on the 4-wheel calipers of the vehicle.

The scenario begins with the actor information receiver (AIR) sensor detecting a person moving across the road, modeled using PreScan®. The AIR sensor is connected to a virtual ADAS control VECU running the advanced emergency braking system (AEBS) bare-metal application and wrapped as an FMU. The ADAS VECU is integrated inside MATLAB Simulink® from MathWorks, as shown in Figure 7. The AIR sensor output is zero in case no object is detected. The analog-to-digital converter (ADC) of the VECU is connected to the AIR sensor. One driving scenario is to be chosen out of 10,000 for testing the ADAS. The simulated vehicle is driven normally on a simulated road. The AIR sensor in the car detects a person running across the road in front of the car at a distance of 20 meters. The ADAS VECU uses this sensor information and determines that a collision would occur with the person unless avoidance action is taken. The ADAS sends a signal to the braking system to stop the car.

Each VECU has a unique CAN ID. The communication between the VECUs is done by CAN frames, as shown in Figure 8, opposite.

The ADAS VECU is wrapped as an FMU and imported into MATLAB Simulink. It receives the speed/RPM CAN packet from the transmission controller VECU and controls the car animation in PreScan. Once a human appears in front of the car, the sensor detects the object and the ADAS VECU automatically sends a CAN packet (ID =0x70) with data containing the brake angle. The transmission controller VECU sends a force to the mechanical brake system in Simcenter Amesim and the car stops. The natural decreasing slope of the car speed is generated by road friction. Once the AIR sensor detects the object at a distance of 20 meters, the ADAS control VECU sends brake signal frames automatically, resulting in a sharp decrease in the car speed with the car stopping at a distance of 14 meters. The simulated scenario is shown in Figure 9, opposite. The car speed decreases from 20 km/s to 0 km/s during 1 sec.

In another scenario, the car’s speed decreases from 90 km/s to 40 km/s during 971 ms, as shown in Figure 10, on the following page. In this case the collision between the vehicle and the human will happen as a result of driving at high speed and suddenly encountering the pedestrian. Many scenarios can be verified with different speeds. AUTOSAR SW analysis for each scenario can be generated and verified, as shown in Figure 11, on the following page.
CONCLUSIONS

As shown in this article, simulating a full heterogeneous automotive system is becoming feasible using virtual platforms for the target VECUs and FMI standard. Not only electrical systems but also mechanical systems can be simulated and verified using the presented methodology. Virtualization is efficiently used to run AUTOSAR SW on a functional simulation. An ADAS, including electrical (digital and analog) and mechanical subsystems is modeled and simulated in a short time (with high performance). Integrating the heterogeneous parts of a system of systems is becoming essential for automotive system development. In this article, the FMI standard is used to model full SoC platforms as FMUs. FMI allows simulating digital, analog, and mechanical parts using an integrated framework. Real software runs efficiently on the modeled system, allowing the verification of the system operation. ADAS is validated on the developed virtual system. Debugging the software with the hardware is made easier with the presented methodology.

Figure 8: CAN frames data flow on virtual CAN bus

Figure 9: (a) The AIR sensor distance (m) and car speed (km/hr) (b) The ADAS VECU stops the car once it detects the human in front of it
REFERENCES


[13] AUTOSAR (AUTomotive Open System ARchitecture), Available at: https://www.AUTOSAR.org/


[17] TASS A Siemens Business, PreScan. Available at:https://tass.plm.automation.siemens.com/prescan


This article was previously presented as a paper at DVCon US 2019.
Fun with UVM Sequences – Coding and Debugging

by Rich Edelman – Mentor, A Siemens Business

In a SystemVerilog UVM testbench, most activity is generated from writing sequences. This article will outline how to build and write basic sequences, and then extend into more advanced usage. The reader will learn about sequences that generate sequence items; sequences that cause other sequences to occur and sequences that manage sequences on other sequencers. Sequences to generate out of order transactions will be investigated. Self-checking sequences will be written.

INTRODUCTION

A UVM sequence is a collection of SystemVerilog code which runs to cause “things to happen”. There are many things that can happen. A sequence most normally creates a transaction, randomizes it and sends it to a sequencer, and then on to a driver. In the driver, the generated transaction will normally cause some activity on the interface pins. For example a WRITE_READ_SEQUENCE could generate a random WRITE transaction and send it to the sequencer and driver. The driver will interpret the WRITE transaction payload and cause a write with the specified address and data.

CREATING A SEQUENCE

A UVM sequence is just a SystemVerilog object that is constructed by calling new. It can be constructed from many different places, but normally a test might construct sequences and then run them - they embody the test. For example a test might be pseudo-coded as:

| LOAD ALL MEMORY LOCATIONS |
| READ ALL MEMORY LOCATIONS, |
| CHECK THAT EXPECTED VALUES MATCH. |

There might be a sequence to write all memory locations from A to B. And another sequence to read all memory locations from A to B. Or something simpler: A WRITE_READ_SEQUENCE that first writes all the memory locations and then reads all the memory locations.

The test below creates a sequence inside a fork/join_none. There will be four sequences running in parallel. Each sequence has a LIMIT variable set and starts to run at the end of the fork/join_none. Once all of the forks are done, the test completes.

class test extends uvm_test;
  `uvm_component_utils(test)

  my_sequence seq;
  ...

  task run_phase(uvm_phase phase);
    phase.raise_objection(this);
    for (int i = 0; i < 4; i++) begin
      fork
        automatic int j = i;
        seq = new($sformatf("seq%0d", j));
        seq.LIMIT = 25 * (j+1);
        seq.start(sqr);
      join_none
    end
    phase.drop_objection(this);
  endtask
endclass

Figure 1
RUNNING A SEQUENCE – CREATING AND SENDING A SEQUENCE ITEM

The sequence below, ‘my_sequence’, is a simple sequence which creates transactions and sends them to the sequencer and driver. In the code below, the body () task is implemented. It is a simple for-loop which iterates through the loop LIMIT times. LIMIT is a variable in the sequence which can be set from outside.

Within the for-loop, a transaction object is constructed by calling new () or using the factory. Then start_item is called to begin the interaction with the sequencer. At this point the sequencer halts the execution of the sequence until the driver is ready. Once the driver is ready, the sequencer causes ‘start_item’ to return. Once start_item has returned, then this sequence has been granted permission to use the driver. Start_item should really be called “REQUEST_TO_SEND”. Now that the sequence has permission to use the driver, it randomizes the transaction, or sets the data values as needed. This is the so-called “LATE RANDOMIZATION” that is a desirable feature. The transactions should be randomized as close to executing as possible, that way they capture the most recent state information in any constraints.

After the transaction has been randomized, and the data values set, it is sent to the driver for processing using ‘finish_item’. Finish_item should really be called “EXECUTE_ITEM”. At this time, the driver gets the transaction handle and will execute it. Once the driver calls ‘item_done ()’, then finish_item will return and the transaction has been executed.

EXECUTING A SEQUENCE ITEM – THE DRIVER

The driver code is relatively simple. It derives from a uvm_driver and contains a run_phase. The run_phase is a thread started automatically by the UVM core. The run_phase is implemented as a forever begin-end loop. In the begin-end block the driver calls seq_item_port.get_next_item (t).

For this example, execution is simple - it prints a message using the transactions’ convert2string () call, and waits for an amount of time controlled by the transactions ‘duration’ class member variable.

Once that ‘execution’ is complete, the seq_item_port.item_done () call is made to signal back to the sequencer and in turn the sequence that the transaction has been executed.
CONTROLLING OTHER SEQUENCES

Sequences can have handles to other sequences; after all, a sequence is just a class object with data members, and a “task body()”, which will run as a thread.

Virtual Sequences

The so-called “virtual sequence” - a sequence which may not generate sequence items, but rather starts sequences on other sequencers. This is a convenient way to create parallel operations from one control point.

A virtual sequence simply has handles to other sequences and sequencers. It starts them or otherwise manages them. The virtual sequence may have acquired the sequencer handles by being assigned from above, or by using a configuration database lookup, or other means. It may have constructed the sequence objects, or have acquired them by similar other means. The virtual sequence is like a puppet master, controlling other sequences.

A virtual sequence might look like:

```plaintext
sequenceA_t sequenceA;
sequenceB_t sequenceB;
sequenceA sqrA;
sequenceB sqrB;

task body();
    sequenceA.start(sqrA);
    sequenceB.start(sqrB);
    ...
```

Related Sequences

In the code snippet below, there are two sequences, ping and pong. They each have a handle to each other. They are designed to take turns. First one sends five transactions, then the other, and so on.

```plaintext
seq_item_port.item_done();
end
endtask
endclass
```

See the appendix for the complete code.

First the handles are constructed and a run limit is set up.

```plaintext
ping_h = new("ping_h");
ping_h.LIMIT = 25;
pong_h = new("pong_h");
pong_h.LIMIT = 40;
```

Then the handles get their “partner” handle.

```plaintext
ping_h.pong_h = pong_h;
pong_h.ping_h = ping_h;
```

Finally the two sequences are started in parallel.

```plaintext
fork
    ping_h.start(sqr);
    pong_h.start(sqr);
join
```

WRITING A SELF-CHECKING SEQUENCE

A self-checking sequence is a sequence which causes some activity and then checks the results for proper behavior. The simplest self-checking sequence issues a WRITE at an address, then a READ from the same address. Now the data read is compared to the data written. In some ways the sequence becomes the GOLDEN model.

```plaintext
class write_read_sequence extends my_sequence;
    `uvm_object_utils(write_read_sequence)
...
task body();
    for (int i = 0; i < LIMIT; i++) begin
        t = new($sformatf("t%0d", i));
        start_item(t);
```
A video traffic generator can be written to generate a stream of background traffic that mimics or models the amount of data a video display might require. There is a minimum bandwidth that is required for video. That calculation will change with the load on the interface or bus, but a simplistic calculation is good enough for this example. The video traffic will generate a “screen” of data 60 times a second. Each screen will have 1920 by 1024 dots. Each dot is represented by a 32 bit word. Using these numbers, the traffic generator must create 471MB per second.

A more complete traffic generator would adjust the arrival rate based on the current conditions - as the other traffic goes up or down, the video traffic generation rate should be adjusted.

Sequences will be used to synchronize other sequences (so called virtual sequences). Often two sequences need to have a relationship between them formalized. For example they cannot enter their critical regions together - they must go single-file. Or they can only run after some common critical region has passed.

The code below declares two sequences which are to be synchronized (syncro_A_h and syncro_B_h). It also declares a synchronizer. There is nothing special about these classes - they have simply agreed to use some technique to be synchronized, as shown on the next page.
The synchronized sequences get a handle to the synchronizer and a starting address.

```python
s = new();
synchro_A_h = new("synchroA");
synchro_B_h = new("synchroB");
```

The synchronizer control is rather simple. It just says “GO” for 20 ticks and “STOP” for 100 ticks.

```python
fork
  forever begin
    #100;
    s.state = GO;
    #20;
    s.state = STOP;
  end
join_none
fork
  forever begin
    synchro_A_h.start(sqr);
  end
fork
  forever begin
    synchro_B_h.start(sqr);
  end
join_none
```

The synchronized sequences are started. They run to completion and then simply get restarted. They run forever.

```python
fork
  forever begin
    synchro_A_h.start(sqr);
  end
fork
  forever begin
    synchro_B_h.start(sqr);
  end
join_none
```

The simple synchronizer with two states – GO and STOP.

```python
typedef enum bit { STOP, GO } synchro_t;

class synchronizer;
  synchro_t state;
endclass
```

The class that uses a synchronizer to NOT execute until told to do so.

```python
class synchro extends my_sequence;
  `uvm_object_utils(synchro)
  bit[31:0] start_addr;
  bit[31:0] addr;
  synchronizer s;
  synchro_transaction t;

  task body();
    forever begin
      addr = start_addr;
      // Is it safe?
      while (s.state == STOP) begin
        #10;
        `uvm_info(get_type_name(), "Waiting...", UVM_MEDIUM)
      end
      t = new($sformatf("t%0d", addr));
      start_item(t);
      if (!t.randomize())
        `uvm_fatal(get_type_name(), "Randomize FAILED")
      t.rw = WRITE;
      t.addr = addr++;
      finish_item(t);
    end
endtask
endclass
```

In simulation, the sequence waits until the synchronizer is in the GO state. Once in the GO state, then the synchronized code generates a transaction using new, and then calls start_item/finish_item to execute it. After waiting for access to the driver and then executing, the synchronized sequence comes back to the top of the loop and checks the
synchronizer state. It will either GO again, or STOP/ WAIT as shown in the example above.

**IMPLEMENTING AN INTERRUPT SERVICE ROUTINE WITH SEQUENCES**

Sequences will be used to provide an “interrupt service routine”. Interrupt service routines “sleep” until needed. This is a unique kind of sequence. In this example implementation it creates an “interrupt service transaction” and does start_item and finish_item. In this way, it can send that ISR transaction handle to the driver. The driver is then going to hold that handle UNTIL there is an interrupt.

As part of the drivers’ job of handling the SystemVerilog Interface, it will handle the interrupts. In this case, handling the interrupt means that some data is put into the “held handle” and then the handle is marked done. Meanwhile, the interrupt service sequence has been waiting for the transaction to be marked DONE. In some parlance this is known as ITEM REALLY DONE. In the UVM, there are other mechanisms for handling this kind of thing, but they are unreliable and more complicated than this solution.

```verilog
// VH

class interrupt_transaction extends transaction;
  `uvm_object_utils(transaction)
  int VALUE;
  bit DONE;
endclass

class interrupt_sequence extends my_sequence;
  `uvm_object_utils(interrupt_sequence)

  interrupt_transaction t;

  task body();
    forever begin
      t = new("isr_transaction");
      start_item(t);
      finish_item(t);
      wait(t.DONE == 1);
    end
endclass

// INTRODUCTION TO SEQUENCES

class interrupt_sequence extends my_sequence;
  `uvm_object_utils(interrupt_sequence)

  interrupt_transaction t;

  task body();
    forever begin
      t = new("isr_transaction");
      start_item(t);
      finish_item(t);
      wait(t.DONE == 1);
    end
endclass

class driver extends uvm_driver#(transaction);
  `uvm_component_utils(driver)

  transaction t;
  bit done;
  int value;
  bit [31:0] mem[1920*1024];

  task interrupt_service_routine(interrupt_transaction isr_h);
    `uvm_info(get_type_name(), "Setting ISR", UVM_MEDIUM)
    done = 0;
    isr_h.DONE = 0;
    wait(done == 1);
    isr_h.VALUE = value;
    isr_h.DONE = 1;
    endtask

  task run_phase(uvm_phase phase);
    forever begin
      seq_item_port.get_next_item(t);
      if($cast(isr, t))
        begin
          fork
            interrupt_service_routine(isr);
            join_none
          end
        else begin
          // REGULAR driver processing
          ...
        end
      seq_item_port.item_done();
    end
  endtask
endclass

// Figure 2

Figure 2
```
SEQUENCES WITH “UTILITY LIBRARIES”

Sequence "utility libraries" will be created and used. Utility libraries are simple bits of code that are useful for the sequence writer – helper functions or other abstractions of the verification process.

The open_door sequence below does just as its name implies. It opens the door to the sequencer and driver. Outside calls can now be made at will using the sequence object handle (seq.read () and seq.write () for example).

```systemverilog
class open_door extends my_sequence;
  `uvm_object_utils(open_door)

  read_transaction r;
  write_transaction w;

  task read(input bit[31:0]addr, output bit[31:0]data);
    r = new("r");
    start_item(r);
    if (!r.randomize())
      `uvm_fatal(get_type_name(), "Randomize FAILED")
    r.rw = READ;
    r.addr = addr;
    finish_item(r);
    data = r.data;
  endtask

  task write(input bit[31:0]addr, input bit[31:0]data);
    w = new("w");
    start_item(w);
    if (!w.randomize())
      `uvm_fatal(get_type_name(), "Randomize FAILED")
    w.rw = WRITE;
    w.addr = addr;
    w.data = data;
    finish_item(w);
  endtask

  task body();
    `uvm_info(get_type_name(), "Starting", UVM_MEDIUM)
    wait(0);
    `uvm_info(get_type_name(), "Finished", UVM_MEDIUM)
  endtask
endclass

open_door open_door_h;
open_door_h = new("open_door");
fork
  open_door_h.start(sqr);
begin
  bit[31:0] rdata;
  for(int i = 0; i < 100; i++) begin
    open_door_h.write(i, i+1);
    open_door_h.read(i, rdata);
    if ( rdata != i+1 ) begin
      `uvm_info(get_type_name(), $sformatf("Error: Wrote \%d, Read \%d", i+1, rdata), UVM_MEDIUM)
      //`uvm_fatal(get_type_name(), "MISMATCH");
    end
  end
end
join_none
```

CALLING C CODE FROM SEQUENCES

Calling C code (using DPI-C) from sequences is easy, but there are a few limitations. DPI import and export statements cannot be placed inside a class – so they must be outside the class in file, global or package scope. As such, they have no design or class object scope.

```systemverilog
import "DPI-C" function void c_code_add(output int z, input int a, input int b);
export "DPI-C" function sv_code;
```

A DPI-C export function or task is just a SystemVerilog function or task that has been “exported” using the export command.

```systemverilog
function void sv_code(int z);
$display("sv_code(z=%d)", z);
endfunction
```
A DPI-C import function or task is a C function with a return value. For a task, the return value is an int (See the SystemVerilog LRM for details). For a function, the return value, is whatever the return value should be.

The simple void function c_code_add () is defined below. It has two inputs and “returns” a value in the pointer *z. This C function calls the exported SystemVerilog function 'sv_code ()'.

```c
#include "stdio.h"
#include "dpiheader.h"

void
c_code_add(int *z, int a, int b)
{
    *z = a + b;
    sv_code(*z);
}
```

The dpiheader.h is a handy way to check the API for DPI-C. In this example, the dpiheader.h (below) is very simple.

```c
void c_code_add(int*, int, int);
void sv_code(int);
```

This sequence does nothing particularly special, it generates transactions, but it does call a C function. (c_code_add RED line below). In terms of writing sequences that call C code, there is really nothing special to do in terms of sequences. The DPI-C code must be properly written and must be declared in a proper scope, as shown below and above, right.

```c
t.c:
    #include "stdio.h"
    #include "dpiheader.h"

    void
c_code_add(int *z, int a, int b)
    {
        *z = a + b;
        sv_code(*z);
    }
```

**CALLING SEQUENCES FROM C CODE**

Calling sequences from C code is harder than calling C code from sequences. This is because sequences are class objects. Class objects do not have “DPI-C Scope”, so in order to call into a sequence (or start a sequence), other means must be used. There are many other references on techniques to do this.
SEQUENCES AND TRANSACTIONS RECORDING

In the example code discussed throughout this article, each of the sequences is running in parallel – at the same time on the single sequencer. The sequences (and recorded streams are listed below as children of that sequencer).

Each row is a sequence executing. It is easy to see in the two screenshots above (figures 4 and 5) how the sequences each take turns sending and executing a transaction on the driver.

CONCLUSION

The reader of this article now knows that sequences are not mysterious or things to be afraid of, but rather that sequences are simply "code" – usually stimulus or test code. That code can be written to do many different things, from the original "random transaction generation" to synchronization to interrupt service routines. Sequences are just code – important code that causes stimulus generation and results checking.

REFERENCES


All source code is available from the author. Contact rich_edelman@mentor.com for access.

This article was previously presented at DVCon US 2019.
Portable Stimulus is one of the latest hot topics in the verification space. Mentor, and other vendors, have had tools in this space for some time, and Accellera just recently released the Portable Test and Stimulus Standard, a standard language that can be used to capture Portable Stimulus semantics.

From the name, one very obvious application of Portable Stimulus is to enable a test scenario to easily be reused across test-execution platforms or levels of verification. As the figure above shows, Portable Stimulus does allow test intent to be reused from block level to subsystem level to SoC level. It also enables a Portable Stimulus tool to create tests that are appropriate for the variety of test platforms on which that verification is carried out – typically SystemVerilog for block and subsystem level, and C tests for SoC level.

However, Portable Stimulus enables more than just test portability. Portable Stimulus enables a high degree of automation in the test creation process, and enables the user to describe tests at a far higher level of abstraction than is possible with techniques like SystemVerilog and UVM.

Portable Stimulus is a modeling language, not a programming language, and this enables much of its expressive power. One of my favorite ways to illustrate the distinction between a programming (implementation) language and a modeling language is to compare navigating with a paper map to navigating with a mapping program, as shown above.

With a paper map, we spend most of our time dealing with how to get from our starting location to the destination. This is just like a programming language, where we’re generally focused on implementing a given algorithm. What highways should we take? Are there tolls involved? What is traffic likely to be like when we drive through?

With a mapping program, we’re free to focus almost entirely on what: where are we starting, where are we going, and what preferences do we have. The mapping program is able to analyze multiple options based on the data it has available and provide one or more good options for routing. Portable Stimulus enables exactly this type of productivity boost over creating tests by hand by allowing a test writer to focus on modeling the rules that govern legal tests, then specify preferences and goals around the specific tests that need to be generated. Automation then takes over and, much like a mapping program, generates tests that adhere to the rules about what constitutes a legal test and satisfies the goals and preferences expressed by the user.
WHAT ABOUT UVM AND SYSTEMVERILOG?

Today, verification at the block and subsystem level is predominantly done in SystemVerilog with UVM, and there is always a need to improve productivity. Portable Stimulus can definitely be added to a UVM environment to more easily create more-complex virtual sequences. In addition, the PSS models used to create these virtual sequences can also be reused across design revisions and across verification. That’s great, but we need to learn a new verification language (PSS) and the methodology behind it in order to get these advantages.

The good news is that SystemVerilog already contains declarative descriptions that, with a little automation, allow us to focus on what stimulus we need to generate instead of how we generate the stimulus we need. The two key declarative specifications in SystemVerilog are the constrained-random stimulus models captured by random variables and constraints in classes, and the functional coverage model captured in covergroups.

Let’s consider coverage placed on a stimulus-generation (transaction or configuration data) class. When generating stimulus randomly, we need to have this coverage in order to know whether we’ve generated all the cases of interest. In most testbench environments today, the stimulus generation is quite decoupled from the coverage goals that we have for that stimulus. Conceptually, it looks something like this:

In this case, we have a SystemVerilog class with random variables and constraints. Our testbench uses the built-in randomize call provided by SystemVerilog to generate random values for the fields in the class, then applies the stimulus to the design using a UVM agent or bus-functional model. At the same time, our testbench samples our coverage model to track what stimulus has been generated. After running a series of simulations, we’ll look at the coverage report to see how our stimulus did in terms of satisfying our testing goals.

We’ll typically note that some coverage is making good progress and will complete if we just run a few (10s, 100s, or 1000s) more simulations. Other coverage may be making little progress toward our goals, and we’ll need to devise some new constraints and/or directed tests to help our testbench produce these critical cases. Next we’ll run more simulations, analyze the coverage results, and repeat the process of creating more-focused tests. Iterating in this way can take significant time in the verification process!

Now, going around this loop is unavoidable if the coverage that we’re collecting is tightly related to the deep internals of the design, and we’re spending time understanding how to provoke certain conditions deep in the design. But, spending time simply coercing
the random generator to do what we want seems wasteful. It’s a bit like asking a mapping program for directions without telling it where we’re going, and just periodically checking whether we’ve arrived at our destination!

BRINGING DECLARATIVE TEST CREATION TO SYSTEMVERILOG
What we’d like to have is a stimulus generator that is aware of our goals and is able to automatically generate the stimulus that efficiently achieves them. The good news is that we already have all the needed information to enable this! Our testbench now looks like the diagram shown below. Instead of using plain-random generation of our stimulus, we use a stimulus generator that is aware of the random variables and constraints in our stimulus class as well as the coverage goals defined in our covergroup, as illustrated in figure 5 below.

Because our stimulus generator is aware of our stimulus-coverage goals, it’s able to efficiently generate the transaction that will achieve our coverage goals. This allows us to achieve our coverage goals easily without “going around the loop” multiple times to analyze coverage holes and create new tests.

WHAT DOES IT REALLY LOOK LIKE?
Now, the diagram below looks simple and straightforward. But, how much work is really involved? To find out, let’s have a look at an example using Mentor’s Questa® inFact tool. Questa® inFact is a full Portable Stimulus tool that works with the Accellera PSS language, but it also provides features for SystemVerilog users that are built on top of the core engines that are designed for processing PSS models. These SystemVerilog-focused features allow the user to continue developing SystemVerilog constraints and covergroups, but get some of the test-creation automation benefits that PSS enables.

Consider a testbench for a DMA engine. Stimulus generation is coordinated by a UVM sequence that interacts with the register model to program the DMA channels, and reacts to interrupts from the DMA engine to detect the end of a transfer, as shown in this DMA transfer descriptor:

```verilog
class wb_dma_descriptor extends uvm_sequence_item;
    `uvm_object_utils(wb_dma_descriptor)
    rand bit[5:0] channel;
    rand bit mode;
    rand bit inc_src;
    rand bit inc_dst;
    rand bit src_sel;
    rand bit dst_sel;
    bit[31:0] src_addr;
    bit[31:0] dst_addr;
```
In this testbench, the DMA transfer descriptor covergroup, we capture the specific attributes of a DMA transfer in the wb_dma_descriptor UVM sequence item shown above. This captures the data required to configure a DMA transfer, as well as the constraints on those fields.

```
rand bit[11:0] tot_sz;
rand bit[2:0] trn_sz;
rand bit[8:0] chk_sz;

constraint trn_sz_c { trn_sz inside {1, 2, 4}; }
constraint channel_c { channel inside {[0:7]}; }
constraint tot_sz_c { tot_sz > 0; }
constraint chk_sz_c { chk_sz > 0; }
endclass
```

We also have a covergroup, shown above, to capture our testing goals with respect to DMA transfers. For example, we need to ensure that all possible combinations of source interface, destination interface, and address increment behavior is exercised. This requirement is captured by the src_dst_inc_cross coverpoint cross.

From looking at our ‘ideal testbench’ diagram, we know that somehow we need to create a stimulus generator that will use these two sources of information to generate goal-driven stimulus.

```
% qso wb_dma_env_pkg::wb_dma_descriptor –covergroup
wb_dma_coverage_pkg::single_desc_cg –o wb_dma_descriptor_gen.svh
```

The inFact command to create a goal-driven stimulus-generator class from the UVM sequence item and covergroup is shown above. QSO, in case you’re wondering, stands for Questa Stimulus Optimizer.

As you can see, the command to create the stimulus-generator class is quite simple. Next, let’s look at the process to integrate the stimulus generator into our testbench.

```
class wb_dma_rand_single_transfer_seq extends wb_dma_transfer_seq;
  `uvm_object_utils(wb_dma_rand_single_transfer_seq)
virtual task body();
  wb_dma_descriptor desc;
  repeat (600) begin
    desc = wb_dma_descriptor::type_id::create("desc");
    start_item(desc);
    if(!desc.randomize())
      `uvm_fatal(get_name(), "Failed to randomize sequence item");
    finish_item(desc);
  end
endtask
endclass
```
A simple pure-random sequence is shown below left. As is typical with a simple random UVM sequence, stimulus is generated in a loop that calls the `start_item` and `finish_item` tasks to drive stimulus, with a call to the built-in SystemVerilog `randomize` function in the middle.

```vhdl
class wb_dma_rand_single_transfer_seq extends wb_dma_transfer_seq;
    `uvm_object_utils(wb_dma_rand_single_transfer_seq)
    virtual task body();

`ifdef INFACT
    wb_dma_descriptor_gen desc_gen = new("desc_gen");
`endif
    wb_dma_single_transfer_descriptor desc;
    repeat (600) begin
        desc = wb_dma_descriptor::type_id::create("desc");
        start_item(desc);
        `ifdef INFACT
            desc_gen.ifc_fill(desc);
        `else
            if (!desc.randomize()) begin
                `uvm_fatal(get_name(), "Failed to randomize sequence item");
            end
        `endif
        finish_item(desc);
    end
endclass
```

The figure directly above, a goal-driven UVM sequence, shows our random sequence updated to support goal-driven stimulus with inFact. The number and scope of the changes is quite small, and highlighted in the code snippet above.

- The sequence needs to create an instance of the stimulus-generator class
- The sequence needs to call an API on the stimulus-generator class instead of calling the built-in `randomize` function.

And that’s it! Pretty simple, right? But what about results?

Well, here’s a side-by-side comparison of goal-driven stimulus generation (top) against pure-random stimulus generation (bottom). In both cases, we’re generating 600 DMA descriptors. inFact’s goal-driven stimulus easily achieves our coverage goals, while random stimulus does okay on some coverage goals but does quite poorly (2.63%) on the largest cross goal.

CONCLUSION

From the example above, it’s clear that using goal-driven stimulus generation can help to achieve our goals more quickly with much less work by the verification engineer. But there are other benefits too. Using goal-driven stimulus brings more predictability to the regression process,
since it becomes much easier to predict how much simulation will be required to achieve a given coverage goal. Goal-driven stimulus also gives us excellent tools for systematically focusing stimulus on the space around a design bug to help determine whether there are related bugs. And, finally, using goal-driven stimulus just helps us achieve larger coverage goals with no more simulation resources than we would normally use.

Using the new PSS language brings many benefits, including portability across verification levels, and greater test-creation productivity. PSS encourages us to think about what we need to test and let automation deal with how to create those tests, which allows us to make better use of test-creation automation. As we’ve seen in this article, we can get some of these same benefits in SystemVerilog at the transaction level by generating goal-driven stimulus. Questa inFact, from Mentor, helps you achieve both of these benefits by providing support for developing scenario-level models with PSS and by allowing you to reuse classes and covergroups from your SystemVerilog testbench and generate stimulus in a more-efficient manner – the PSS way.
Formal Bug Hunting with “River Fishing” Techniques

by Mark Eslinger and Ping Yeung – Mentor, A Siemens Business

Formal verification has been used successfully to verify today’s SoC designs. Traditional formal verification, which starts from time 0, is good for early design verification, but it is inefficient for hunting complex functional bugs. Based on our experience, complex bugs happen when there are multiple interactions of events happening under uncommon scenarios. Our methodology leverages functional simulation activity and starts formal verification from interesting “fishing spots” in the simulation traces. In this article, we are going to share the interesting fishing spots and explain how formal engine health is used to prioritize and guide the bug hunting process.

INTRODUCTION

Formal verification has been used successfully by a lot of companies to verify complex SoCs and safety-critical designs. The ABCs of formal have been used extensively as described in:

• Assurance: to prove and confirm the correctness of design behavior
• Bug hunting: to find known bugs or explore unknown bugs in the design
• Coverage closure: to determine if a coverage statement/bin/element is reachable or unreachable

Using formal verification to uncover new bugs is emerging as an efficient verification approach when functional simulation regression is stabilized and is not finding as many bugs as before. Traditional formal verification, which starts to explore the design from time 0, is good for early design verification. As more blocks are being integrated, the state space of the design increases exponentially, making traditional formal verification approaches inefficient for bug hunting. As we have observed, complex bugs happen under combinations of events and scenarios.

Sophisticated approaches such as waypoints, coverpoints and goal-posting have been proposed and used successfully to find bugs buried deep in the design, especially post-silicon bugs.

RIVER FISHING APPROACH

We have expanded these approaches further. “River fishing” is a good metaphor for our methodology: by identifying some good “fishing spots,” we can significantly increase the number of fish we catch. Instead of using one initial state to start formal verification, we pick out interesting states from functional simulation traces, as depicted in Figure 2, and start formal verification from those fishing spots.
The major difference between this approach and goal-posting is that instead of targeting the coverpoints or goals to explore deep into the design, we are leveraging simulation traces, as in Figure 3, to explore interesting initial states close to the final targets. If functional simulations have exercised the coverpoints or goals already, we have an initial state as good as goal-posting. Also, if some targets require specific pre-conditions to happen ahead of time, we can also take advantage of the different functional tests and identify different fishing spots for those targets. Effectively, the river fishing approach leverages what has already been performed by functional simulations and starts formal verification as close to the targets as possible.

To summarize, the “river fishing” formal bug hunting methodology consists of three major steps:

1. Identify and extract a set of good fishing spots from the simulation traces
2. Screen and prioritize the fruitfulness of these fishing spots using formal engine health
3. Launch multiple formal engines concurrently on a server farm environment

In the following sections, we describe each of these steps in more detail.

IDENTIFYING “FISHING SPOTS”

River fishing experts have suggested that the interesting fishing spots are special sections of the river where there are unusual activities. Picking interesting spots from simulation traces for formal verification is similar. Our approach is based on the patent, Selection of Initial States for Formal Verification, with some variations, illustrated by figure 5.

A quick and comprehensive two-tier approach is used to pick out the interesting fishing spots. The primary criteria are:

1. Interface interactions

Inter-module communication and standard protocol interfaces are infamous for design issues. As protocol monitors are commonly used to instrument these on-chip buses and common interfaces, we can capture interesting fishing spots based on activities on these interfaces. Functional simulation is already generating tons of bus transactions; formal verification can leverage them to explore new corner cases.
2. Control and interrupts
These are FSMs, bus controllers, memory controllers, flow charts, algorithmic controls, and so on representing critical control logic in a design. It may not be trivial for formal to traverse all the possible or even some of the deep state sequences. Hence, every new state or transition exercised by simulation are potentially interesting fishing spots for formal verification.

3. Concurrent events
These are the arbiters, schedulers, switches, multiplexing logics, etc., in a design. The timing and sequence of events are important. Functional simulation may exercise the same sequence of events repetitively. Instead, formal verification can leverage the setup, but change the ordering of the events to ensure the design is robust.

4. Feedback, loops, and counts
These are the FIFOs, timers, counters, and so forth handling the data transfers, bursting, and computations in a design. Functional simulation exercises the non-stressful situations well. By leveraging the simulation activities, formal verification can explore the stressful corner case scenarios; such as starting, stopping, overflow, underflow, and stalling.

5. User-defined assertions and coverage properties
The fan-in cones of the user-defined properties are great coverpoints and sub-goals for formal verification. If they have already been covered by simulation, they will be good fishing spots for formal verification to verify the target properties.

Secondary criteria are related to inclusion or exclusion of the primary ones as described in the patent.[6]

• Inclusion criteria: When gathering the primary fishing spots, we want to mark the high-value ones; such as states that satisfy multiple criteria, states with low frequency of change, and states with new activities
• Exclusion criteria: At the same time we want to filter out poor-value fishing spots; such as duplicate states, error states that lead to misleading results, and states during the power-up, reset, initialization, or configuration sequence

SCREENING WITH ENGINE HEALTH
Formal verification is different from simulation. Simulation runs are predictable and will finish. When running formal verification on a large set of assertions, it is difficult to predict when and if it will finish. Hence, it is important to understand
formal engine health and leverage it to measure the progress made by formal verification. Based on our experience, formal engine health is a good matrix to estimate, prioritize, and monitor formal verification runs.

The concept of engine health helps us determine if, and to what degree, formal engines are contributing to the progress made on a given set of targets. We define formal engine health with a set of parameters:

1. Formal targets concluded (proven/fired/covered/uncoverable)
2. Sequential depth explored or cone of influence analyzed
3. Formal knowledge and engine setting acquired

Targets concluded and sequential depth explored are two well-established metrics that have been used regularly to determine the overall progress of a formal run.

On the other hand, if we want to understand how formal is doing on an individual property or target, we have to go deeper to examine the cone of influence and the formal knowledge acquired on those targets. A formal setup may work extremely well on a few targets, but not so well on the others. Cone of influence provides more information about the depth explored per target. Cone of influence provides detailed information on the fan-in logic and their dependence concerning the targets. It highlights the “rocks” in the fan-in logic where formal gets hung up, spending a lot of time analyzing. These difficult design elements can be identified, and users can determine whether it is necessary to intervene. Adding cutpoints, abstracting the design elements, or enabling some special engines are a few approaches to enable the “rock” to be analyzed more efficiently.

Traditionally, formal engine health was used to monitor computational intensive formal runs. Until a recent project, it did not occur to us that it can also be a good approach to screen fishing spots. With a lot of formal runs to perform on limited compute resources (and licenses), we found ourselves constantly terminating runs with potentially unfruitful fishing spots. The formal engine health parameters (described above) were used to determine whether compute servers were allocated to explore the spots thoroughly, as indicated in figure 6 below.

The process goes like this. As the goal is to screen and prioritize the effectiveness of the fishing spots, a quick run with a set of explorative formal engines is performed. An initial fishing spot is used to establish the initial values of the formal engine health parameters, as in (Proof, Depth, Knowledge):

$$F(Spot_0) = (P_0, D_0, K_0)$$
$$\Delta F(Spot_n) = (P_n, D_n, K_n) - (P_0, D_0, K_0) = \Delta(P_n, D_n, K_n)$$

Then the subsequent fishing spot is compared with the initial one. If the delta, $\Delta F$, is low, we drop the spot from further investigation. By doing this we can screen the fishing spots accordingly and launch formal runs with only the differentiating and fruitful spots. At the same time, we also cache the formal knowledge during the screening process, so CPU resources are not wasted. Subsequent formal runs will be faster.
MONITORING ENGINE HEALTH

With a prioritized list of fishing spots, extensive formal runs can be launched concurrently on a server farm environment. A master server will manage and monitor all the formal processes. It will collect the formal engine health parameters from each of the formal runs continuously.

Figure 7 above shows the distribution of the formal targets at the beginning of a formal run. It lets users know how much progress has been made. Initially most of the 33 targets were inconclusive (I). With multi-cores running concurrently, formal verification gradually verified the targets into one of the following catalogs: firing (F), vacuous (V), un-coverable (U), covered (C), and proof (P).

A comprehensive formal tool, such as,(7) has a set of formal engines to handle designs with different structures. As the formal run progresses, we can examine the status of the engines to understand which ones are finding results and which ones are not being productive. If an engine is not contributing to any of the current set of results, we can swap out that engine to save resources and focus the other engines on the task at hand. Figure 8 shows a snapshot of the engines in the middle of Portable Stimulus a distributed formal run.

In figure 8 below, the “Proven/Unsatisfiable” columns show which engines solve the safety/liveness/vacuity/cover type checks. The “Fired/Satisfied” columns show which engines generate the counterexamples. Engine 7 is very productive in finding a lot of proofs and firings. Engine 0 (the housekeeping engine) and Engine 10 have found some proofs. On the other hand, Engines 12 and 17 (designed for some difficult problems) haven’t been contributing to the results although they have been working on the problems. The “Inconclusive Targets” columns (Good, Fair, Poor) show the individual engine health for the work-in-process targets. Engine 12 is working on 75 targets that have good health; in other words, it is making good progress. There are three targets with fair and poor health. Similarly, Engine 10 is working well on 51 properties but is not effective on more targets.

We have captured two representative bug situations below. They were successful only after we started formal verification from fishing spots deep in the simulation traces. Formal engine health screening was also used to eliminate a lot of potential unsuccessful formal runs from the beginning or during the process.

RESULTS

We have captured two representative bug situations below. They were successful only after we started formal verification from fishing spots deep in the simulation traces. Formal engine health screening
was also used to eliminate a lot of potential unsuccessful formal runs from the beginning or during the process.

**Case Study 1 (figure 9):** To save power, today’s designs have a lot of ratio synchronous clocks that run each component as slowly as possible. As a result, ratio synchronous interfaces are common. At one of these interfaces, the fast clock-domain was designed to sample the data at the end of a slow clock period when the data valid condition was asserted. However, under some corner case situations (unknown to the design team initially), the data was sampled even when the valid condition was not asserted. As a result, corrupted data was registered and passed on within the system. The fishing spots were determined based on activities on the two interfaces, counters, and control logic. Several fishing spots were picked after the design had been initialized and also after data had been flowing correctly for hundreds of cycles between these two interfaces. Formal verification exposed the incomplete handshake between these two interfaces which caused the data to be sampled invalidly.

**Case Study 2 (figure 10):** In this data transfer controller, dynamic transfer channels were set up to handle data packets with different priorities. In this case, when more than one channel finished the transfer at the same time, one set of address pointers was de-allocated twice while the other set was not de-allocated, causing memory leaks and data corruption. The fishing spots were determined based on activities on the concurrent events, counters, and complex control logic. In this type of design, it is very difficult for formal verification to set up these data transfers. However, they are readily available in the simulation regression. By leveraging multiple fishing spots deep in the simulation runs and prioritizing them based on engine health, formal verification was able to expose the weakness in the channel de-allocated logic and synchronized two channels to complete the transfer at the same time.

**SUMMARY**

We firmly believe that simulation and formal methodologies can be used together to accelerate the verification of intricate designs. Some companies have already made organizational changes to facilitate this approach. The idea is to leverage what has been learned or achieved in one methodology as stepping stones for the other methodology. In this article, we have described a river fishing technique. It leverages the functional simulation activities and starts formal verification from interesting fishing spots in the simulation traces. As described, it consists of three major steps:

1. Identify and extract a set of good fishing spots from the simulation traces
2. Screen and prioritize the fishing spots using formal engine health
3. Launch and monitor multiple formal runs on the computing servers

To identify a set of fishing spots, we have highlighted several criteria, including interface interactions, control and interrupts, concurrent events, feedback loops and counts, user-defined assertions, and coverage properties. Then the fishing spots are screened and prioritized based on the formal engine health. We define formal engine health with parameters consisting of formal targets proven or fired, sequential depth explored, and formal knowledge acquired. Then the set of fishing spots are used to initialize multiple formal runs while a centralized database is aggregating all the results together. Based on the results presented and the complex bugs found, we can conclude that the river fishing technique does help improve the quality of the results in a formal regression environment.

REFERENCES
Clock-Domain Crossing (CDC) issues are the second most common reason for silicon re-spins. Most modern day designs have more than one clock, many of which are asynchronous. Signals that pass between logic clocked by different asynchronous clocks are called clock-domain crossing (CDC) signals. Due to the asynchronous nature of clocks, there is a possibility of CDC signals going metastable and propagating incorrect logic downstream resulting in functional failure of the design. To mitigate these problems, synchronizers are used on CDC paths.

Each synchronizer is dependent on a set of assumptions or protocols, which when violated can make the transfer unreliable. To avoid such issues, it is crucial to validate each synchronizer for its reliability. Engineers have been using assertion-based verification methods to verify synchronizer protocols in Formal and Simulation environments, but have fallen short of addressing the issue.

In this article, we will present the most prominent challenges faced with the existing methodology currently being used to verify synchronizer protocols and propose a new methodology to overcome them. Following are the key challenges:

- Significant effort and time is required to setup the design before starting verification using Formal and Simulation. The setup also includes translation of design configurations and settings to both environments which requires technical expertise
- Considerable debug effort required to review firings in Formal and Simulation
- Missing correlation between protocol assertion checks in Formal, Simulation and associated CDC paths
- Lack of re-utilization of benefits and efforts of both Formal, Simulation for faster design closure

The proposed methodology automates the progression of setup, constraint and results from CDC to Formal to Simulation. The methodology also addresses the important issue of correlating Formal and Simulation results to CDC results by using a new technique. In this technique, we add hooks to Formal and Simulation that enable us to extract important information and associate them to the CDC results. The proposed approach has also been automated for the seamless adoption of this methodology.

The article concludes with a demonstration of the proposed methodology on a set of real life designs. The demonstration proves that the methodology significantly reduces design and verification engineers’ effort and helps in achieving faster design closure.

INTRODUCTION

Most designs now-a-days have multiple asynchronous clocks. Logic clocked by each asynchronous clock forms the clock-domain for that clock. Signals that cross the boundaries of these clock-domains are called Clock-Domain Crossing (CDC) signals. Asynchronous nature of these CDC signals may cause setup or hold time violation of flip flops resulting in metastability. To ensure that these signals are properly synchronized, synchronizers are used. Synchronizers help protect the receiver domain from sampling metastable values and prevent data loss or corruption.

However, adding a synchronizer is just part of the solution. Despite the presence of synchronizers on CDC paths, there is still a possibility of data loss or corruption due to incorrect usage or interfacing of synchronizers. Reliability of every synchronizer depends on a set of assumptions or protocols. If the logic associated to the synchronizer does not interact as per the synchronizer’s protocol, the reliability of the synchronizer might break. For example data loss can occur if the input to a two DFF synchronizer is not stable for at least two clock cycles, or if the request and acknowledge signals in a handshake synchronizer
do not occur in a particular order. The figure below illustrates these scenarios.

These functional failures can further result in increased iterations and even silicon re-spins. To ensure such issues are not missed, designers and verification engineers verify synchronizer protocols. This Dynamic CDC Protocol Verification is done by extracting assertions for synchronizer protocols and verifying them using Formal and Simulation methods.

EXISTING DYNAMIC CDC PROTOCOL VERIFICATION METHODOLOGY

Figure 2 illustrates the existing approach for Dynamic CDC Protocol Verification. First step is to perform Static CDC analysis to ensure the presence of synchronizing logic on CDC paths. Post this step, assertions for synchronizer protocols are extracted based on CDC results. These assertions are then verified to ensure that the synchronizers present in the design are reliable.

To verify these protocol assertions, Simulation and/ or Formal tools are used. The setup for each of these tools is different and has to be done manually, which increases the verification time and effort. Once the setup is complete, engineers perform protocol assertion verification using the Formal, Simulation tools and their results are debugged separately. While this methodology promises to identify CDC protocol bugs, it suffers from many issues such as substantial effort required for setup and debug, lack of correlation with CDC results, among others.

The issues of data loss or corruption due to synchronizer protocol violation must be identified and addressed early in the design cycle, otherwise they can lead to functional failures in later stages.

Figure 1: Data loss in two DFF and Handshake synchronizers due to protocol violation

Figure 2: Existing methodology flowchart
Figure 3: False two DFF firing in Formal due to constants missing in setup

Figure 4: False 2DFF Formal firing due to unspecified clock frequency
CHALLENGES FACED WITH EXISTING METHODOLOGY

In the existing methodology, engineers use Simulation and Formal methods separately to generate and verify assertions for synchronizer protocols. However, the methodology entails the following common challenges:

- **Setup:** Significant effort and time is required to set up the design for Formal and Simulation runs. Design constraints and directives that were already specified in the CDC step, need to be translated and propagated to both the Formal and Simulation tools separately. For example, constant values, clock frequencies of design and so on. Issues at this step can result in unexpected design behavior in dynamic environment. Figure 3 shows an example where constant specified on input signal at the time of Static CDC was not propagated to Formal setup resulting in false two DFF protocol firing.

- **Debug:** Analyzing and debugging Formal and Simulation results requires expert knowledge about both the environments. Issues in setup or inaccurate assertions can result in large number of false firings, which further increase the problem. For example, incorrect clock frequency or domain specified on ports can result in unexpected behavior. Figure 4 shows an example of a false two DFF stability check firing in Formal where unspecified clock frequency in Formal setup resulted in clock incorrectly changing as per fastest clock’s frequency.

- **Correlation to CDC:** The existing methodology suffers from lack of relation between CDC, Formal and Simulation results. The debug environments of Formal and Simulation are very different from CDC. Hence, establishing a relation between CDC, Formal and Simulation results for the purpose of coverage and review of CDC paths is cumbersome and time consuming. For example, complex synchronizers such as FIFO or handshake have multiple assertions that are treated as separate entities by both Formal and Simulation, but relate back to a single CDC path. Issues or errors during correlating results can lead to missed bugs. Figure 5 shows various handshake synchronizer protocol assertions that relate back to the same crossing. If not correlated properly, some uncovered or fired assertions may be missed resulting in data loss or corruption in the design.

- **Lack of effort re-utilization:** Due to limitations of both Formal and Simulation methods, to save time and to ensure sufficient coverage, verification engineers may have to change from one method to another, or adopt both simultaneously.

![Figure 5: Multiple handshake protocol assertions](image-url)
In such scenarios, the challenge faced is in terms of redundancy and lack of verification effort re-utilization. For example, if Formal step is followed by Simulation step, assertions that were already proven by Formal will have to be re-verified in Simulation.

Apart from these common challenges, both Formal and Simulation methods have their own advantages and disadvantages. While Simulation is more intuitive to understand, it runs into coverage issues. Bugs can go undetected if the quality of testbench is not good. On the other hand, Formal offers exhaustive proofs but runs into infrastructural and capacity issues.

PROPOSED METHODOLOGY FOR DYNAMIC CDC PROTOCOL VERIFICATION

In this article, we propose a methodology that not only helps overcome the common challenges of Formal and Simulation verification, but also utilizes the advantages and efforts of both the methods. Figure 6 illustrates the proposed methodology.

With this methodology, we propose the automated propagation of setup from CDC to Formal and Simulation tools to help reduce the time and manual effort required in setting the design. This propagated setup includes design settings like clocks, resets, constants, port clock-domains, amongst others that are easily available at the CDC step. This automation ensures easy and correct propagation of setup without the hurdles of technical expertise required on the engineer’s part to setup each tool’s environment.

The protocol assertion generation step inserts hooks into Formal and Simulation to get insights into the status of each assertion. These insights help to correlate the Formal and Simulation results back to the associated CDC paths. This helps ensure that design and verification engineers can view the dynamic verification status of each CDC path.

The methodology aims to not only save but also re-utilize designer and verification engineer’s effort for faster design closure. Ensuring automated propagation of setup and barring the propagation of Formal proven assertions to Simulation helps in achieving the objective.

The proposed methodology has the following workflow:

1. **Static CDC Analysis**: Perform static CDC analysis on the design to ensure that all the relevant CDC paths are synchronized using proper synchronizers.

2. **Protocol Assertion Generation**: Generate assertions for protocols of synchronizers present in the design. Based on static CDC analysis, each synchronizer on a CDC path is analyzed and protocol assertions are generated depending upon the type of synchronizer and its connections. Simultaneously, the design settings and configurations passed to the CDC step are translated to Formal and Simulation language, generating the required setup for running both environments. This step also includes inserting hooks in both environments, which helps in correlating Formal and Simulation results back to the CDC environment.

Figure 6: Proposed methodology flowchart
3. Formal Analysis: Verify the generated synchronizer protocol assertions in Formal environment using the setup generated. The automated setup reduces significant effort required to setup the design and also avoids the possibility of false firings due to incomplete or incorrect setup.

4. Formal Debug: Debug the assertion firings generated by Formal tool by analyzing the counter example. Design constraints, if any, can be added to eliminate false scenarios and the design can be re-run incrementally with the added constraints.

5. Simulation: Run Simulation on the design using the auto-generated setup to verify the protocol assertions that are non-proven by the Formal tool. After Formal analysis, the Simulation setup is updated incrementally to ensure only non-proven assertions are promoted to Simulation environment. This helps ensure that the effort put in to get conclusive results in Formal is re-utilized resulting in faster design closure.

6. Review CDCs: Review the combined results of Formal and Simulation along with CDC results. The hooks inserted in Formal and Simulation are used to extract status of various protocol assertions associated to a crossing and correlate them back to CDC environment. This helps in ensuring that bugs are not missed and protocols for synchronizers on CDC paths are not violated. Figure 7 shows the view of a combined result set along with the related CDC paths, for a unit design.

CASE STUDY
Both existing and proposed methodologies were compared by using them for Dynamic CDC Protocol Verification of real life designs. The comparisons were performed on a set of real life designs, ranging from 1 to 30 million gates. The setup time required and results obtained with both the methodologies were recorded and analyzed.

A. Static CDC Analysis, Protocol Assertion Generation
The designs first underwent Static CDC analysis to ensure the presence of synchronizers on all relevant CDC paths. Post Static CDC analysis, assertions for synchronizer protocols were generated and verified in Dynamic environment as per each methodology.

B. Formal Verification
In the existing methodology, the protocol assertions were validated first in a Formal environment. The design settings and constraints like clocks, resets, constants, and so on were translated from Static CDC environment and re-specified in the Formal tool's language manually. This required significant effort and expertise. Once the Formal tool environment setup was complete, the design was verified formally and the assertion firings were debugged.

Similarly, in the proposed methodology, the first step was protocol assertion verification in Formal environment. The design configurations and constraints for Formal were auto-generated by using the proposed methodology. This auto-generated setup was reviewed and used to verify design formally. The setup automation saved a lot of
manual effort and expertise that was earlier required to port the setup from CDC to Formal.

C. Simulation

Post Formal verification, the engineer decided to take the Simulation path. With the existing methodology, protocol assertions that were proven formally were re-verified in Simulation environment. The design, testbench were compiled and setup for Simulation. On completion of Simulation, the results were analyzed and firings were debugged.

In the proposed methodology, only the assertions that were non-proven in the Formal environment, were verified in Simulation. The auto-generated setup along with testbench, was used to run Simulation. This was followed by debugging the Simulation firings. Significant reduction in debug effort and time was observed due to running Simulation only for Formal non-proven assertions.

Tables 1 and 2 summarize the comparison between existing and proposed methodology performed on a set of real life designs, ranging from 1 to 30 million gates. Significant reduction in debug effort and time was observed. Setup time for Simulation was also reduced. Firings in Formal also reduced for each design due to reduction in false firings caused by setup issues. Assertions passed to Simulation step also reduced due to exclusion of Formal proven assertions thereby reducing the verification effort required.

An overall reduction in effort and time was observed due to automated propagation of setup. The debug effort was also reduced due to minimization of incomplete and incorrect setup. The correlation of Formal, Simulation results to CDC paths helped in debug and easier identification of associated CDC paths.

### Table 1: Results Using Existing Methodology

<table>
<thead>
<tr>
<th>Design</th>
<th>Setup Time</th>
<th>Assertions</th>
<th>Run Time</th>
<th>Formal Coverage*</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Total</td>
<td>Failed</td>
<td>Proven</td>
</tr>
<tr>
<td>A</td>
<td>1w 3d</td>
<td>170</td>
<td>79</td>
<td>91</td>
</tr>
<tr>
<td>B</td>
<td>2w 2d</td>
<td>800</td>
<td>304</td>
<td>437</td>
</tr>
<tr>
<td>C</td>
<td>5w 4d</td>
<td>8552</td>
<td>5673</td>
<td>877</td>
</tr>
</tbody>
</table>

### Table 2: Results Using Recommended Methodology

<table>
<thead>
<tr>
<th>Design</th>
<th>Setup Time</th>
<th>Assertions</th>
<th>Run Time</th>
<th>Simulation Tool Coverage**</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Total</td>
<td>Failed</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>10 min</td>
<td>170</td>
<td>14</td>
<td>20 min</td>
</tr>
<tr>
<td>B</td>
<td>17 min</td>
<td>800</td>
<td>83</td>
<td>35 min</td>
</tr>
<tr>
<td>C</td>
<td>30 min</td>
<td>8552</td>
<td>127</td>
<td>1 hr</td>
</tr>
</tbody>
</table>

### Table 3: CDC Protocol Verification with Formal

<table>
<thead>
<tr>
<th>Design</th>
<th>Setup Time</th>
<th>Assertions</th>
<th>Run Time</th>
<th>Formal Coverage*</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Total</td>
<td>Failed</td>
<td>Proven</td>
</tr>
<tr>
<td>A</td>
<td>2d</td>
<td>170</td>
<td>32</td>
<td>138</td>
</tr>
<tr>
<td>B</td>
<td>5d</td>
<td>800</td>
<td>119</td>
<td>654</td>
</tr>
<tr>
<td>C</td>
<td>1w</td>
<td>8552</td>
<td>1825</td>
<td>4907</td>
</tr>
</tbody>
</table>

### Table 4: CDC Protocol Verification with Simulation (after Formal)

<table>
<thead>
<tr>
<th>Design</th>
<th>Setup Time</th>
<th>Assertions</th>
<th>Run Time</th>
<th>Simulation Tool Coverage**</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Total</td>
<td>Failed</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>1 min</td>
<td>170</td>
<td>6</td>
<td>20 min</td>
</tr>
<tr>
<td>B</td>
<td>1 min</td>
<td>800</td>
<td>34</td>
<td>35 min</td>
</tr>
<tr>
<td>C</td>
<td>5 min</td>
<td>8552</td>
<td>76</td>
<td>1 hr</td>
</tr>
</tbody>
</table>

* Formal Coverage = ((Failed Assertions + Proven Assertions) / Total Assertions) * 100

** Simulation tool coverage = Coverage metric of Simulation tool = ((Failed Assertions + Covered non-failed Assertions) / Total Assertions) * 100
CONCLUSION
The proposed methodology significantly reduces the time and effort required for Dynamic CDC Protocol Verification closure of designs. Not only does this approach help design and verification engineers overcome the challenges faced during Formal and Simulation verification but also enable them to reap the benefits of both the methods. The methodology is seamless to adopt and saves both time and effort to achieve faster closure.

REFERENCES

This article was previously presented as a paper at DVCon US 2019.
Auto-Generating Implementation-Level Sequences for PSS

by Amanjyot Kaur and Louie De Luna - Agnisys

INTRODUCTION
The Portable Test and Stimulus Standard (PSS) v1.0a aims to help the user describe the high-level test intent and create code for any downstream verification platform. This article starts out with a quick introduction to PSS; it then shows how to use Questa® inFact to create a PSS description and generate C & UVM code. We will then show how you can use ISequenceSpec™ to describe the Hardware/Software Interface (HSI) layer to create truly portable tests.

THE HSI LAYER
The HSI layer or Hardware Abstraction Layer (HAL) is used invariably in all modern digital systems. It is a way to abstract out the details of the hardware from the higher layers of the system, like the device driver, firmware, and software stack. For example, a power up sequence or an initialization sequence could have a list of specific steps that need to be carried out in a certain order. These steps could include register field writes, waits for a specific period of time or events, or set up of clock PLLs by trimming the values of a variable stored in a field.

Typically, an SoC consists of several subsystems and each subsystem could contain a multitude of blocks or IPs. Each IP could have its own HSI layer or a set of sequences that abstract out the core functionality of the IP. A subsystem could use these IP-level sequences and build its own hierarchical sequences. Then at the top, SoC level, there could be sequences built using the subsystem sequences. These hierarchical sequences are analogous to the register level hierarchy (or RTL hierarchy).

These set of sequences need to run on a variety of verification platforms used for system development; such as simulation, emulation, prototyping, and post-silicon validation. Even though the core algorithm of the sequence is the same or very similar for various platforms, the way the sequence is coded can be quite different.

PSS: WHAT IT IS
AND WHAT IT’S NOT
PSS enables users to create the high-level test intent description, including the data flow, scheduling, and resource constraints of the target device under test (DUT) and the target test platform. A PSS processing tool, such as Mentor’s Questa® inFact, can create implementations of the scenarios that match the test intent while satisfying the scheduling and resource constraints of the abstract model.
PSS incorporates the concept of a unit of behavior that is called an action. Actions can represent operations performed by either a device or validation environment component. Examples of actions are the read or write operations that a CPU core performs or a power transition that is performed by the power management unit.

The following code shows the syntax of an action which defines a write operation.

```pss
action write {
    output data_buf data;
    rand int size;
    //implementation details
    ......
}
```

*Code 1: Action block for write operation*

A PSS model interacts with external foreign-language code using “exec” blocks that provide a mechanism for declaring runtime functionality associated with a component or action. The foreign language may be SystemVerilog (SV)/UVM sequences or sequences defined in C.

PSS variables are referenced using mustache notation: `{{expression}}`. A reference is to an expression involving variables declared in the scope in which the exec block is declared.

The code below is an example of an exec block mapped to C sequences.

```pss
extend action mem_Desc: memWrite {
    exec body C = "
        REG32_WRITE ({{reg1.addr}}, {{reg1.data}});
    "
}
```

*Code 2: An “exec” block mapped to C sequences*

The PSS language, by design, avoids implementation details. The implementation is specific to the verification platform on which the test will run. If the implementation details were made part of PSS, the language would have lost its purpose of being portable, reusable, and adaptable.

**LOW-LEVEL SEQUENCES**

In functional verification, firmware, and validation, the low-level sequences play a vital role. In general, there are two types of sequences – those associated with the functionality of the DUT and those associated with its testing and validation. The functional sequences create a certain behavior in the DUT while the test sequences verify that the logic design meets the specification. Test sequences model the stimulus that can be used for direct and constrained random tests required to test complex IPs, subsystems, and SoCs.

The higher level of functional and test sequences calls these low-level sequences. Together they form the implementation layer.

**THE PROBLEM: LACK OF IMPLEMENTATION DETAIL**

Since PSS intentionally does not contain implementation level detail, it is up to the user to create the low-level sequences in some way and “stitch” it with the output generated by the PSS processing tool.

Many users create a test intent using PSS and generate the C and UVM models from it. However, the hardware/software interface layer is not part of this description.

It's not always easy for users to manually write long sequences that deal with the registers and pin manipulation commands. It becomes highly difficult to keep each design team member aligned with the updated sequences as the project progresses. Moreover, each member of the team is writing the same sequence, perhaps in different contexts and different languages, but otherwise executing redundant efforts. Hence, due to the absence of a common set of sequences which can be shared across the teams, the same algorithm needs to be implemented to target various platforms on which the PSS generated test scenarios need to run. Often users need to write sequences for thousands of
registers at the block level manually and, for mapping the sequences with exec blocks of PSS, map each sequence with its corresponding exec block. This is clearly an inefficient process that is a highly error prone and time-consuming task.

So, there is a need for a solution that not only addresses challenges such as consumption of time and resources and increased effort due to the non-availability of a common platform between different teams, but also enables the user to focus on high-level challenges while it automatically handles all the low-level sequences and design functionality.

**THE SOLUTION: CREATE A GOLDEN SPECIFICATION FOR SEQUENCES**

If the implementation-level sequences (also known as test realization code) are captured in a golden specification, just as the test intent is captured in PSS, then all target outputs can be created from these specifications and a fully portable solution can be achieved.

A scheme for capturing the golden specification for implementation-level sequences will need to have the following capabilities:

- High level of abstraction devoid of implementation detail
- Control flow
- Access to hierarchical register data for SoC, subsystem, and IP
- Access to pins, signals, and interfaces
- High-level execution of arbitrary transactions
- Deal with timing differently based on the target
- Hierarchy of sequences and base address of the DUT
- Parallelism between:
  - Device and target environment
  - Subsystem and IP level
  - Several interfaces at IP level

In addition to these core features, certain meta information is also required so that these sequences can be written in a concise way without any duplication.

These features are:

- Arguments
- Parameters
- Enum/define/macros
- Structures
- Look up tables

The system must have a variety of input formats that users are already comfortable with; such as Excel, Word, and plain ASCII text. It must have some mechanism to abstract out the actual code generation using some form of template. This is a requirement because it helps in abstraction and keeps implementation detail away from the sequence specification.

The target platforms also impose certain requirements on this scheme. Certain concepts make sense in one platform and not in the other. For example, the concept of front door and back door register read/writes are clear to verification engineers, while speed of writes and reads is important for firmware developers; consequently they worry about consolidated read/writes and read-modify-writes. Similarly, a validation engineer may be interested in optimizing the throughput of the chip-tester by simultaneously testing four or eight chips.

ISequenceSpec, was designed from the ground up to address these concerns. Here are some examples to show how that’s done.

Figure 2: inFact and ISequenceSpec used together to create end-to-end portability
Figure 2 shows the interface between ISequenceSpec and inFact. Register Data is block-level IP that can be in standard formats such as SystemRDL or IP-XACT. Users can write the sequences in the ISequenceSpec spreadsheet or text environment, which in turn generates the output sequence (SystemVerilog or C) and the corresponding PSS exec blocks.

The sequences can be simple or complex, involving conditional expressions (if/else), array of registers, loops (for/while), wait, arguments and in-line functions. These exec blocks connect sequences generated by ISequenceSpec with the behavior and scenarios of block IP generated by inFact; i.e., actions and activity blocks.

The exec blocks are the connecting layer between the behavior in PSS and the external language sequences. They also have the capability to include a register map independent of PSS. Users can capture both hardware functional specifications and addressable register specifications in a single place and create an executable specification. To access the definition of a register while creating sequences, users can either create their own IP or can import existing register IP. This register data can be imported from a variety of formats, such as:

- XML
- RALF
- IP-XACT
- SystemRDL
- Custom XML
- Custom CSV

Importing the register data will automatically create a document consisting of the entire register map.

In figure 3 below is an example of the ISequenceSpec input format for sequences:

Regarding long and complex sequences that are common in today’s SoCs, Figure 4 shows the level of complexity that users can easily capture in ISequenceSpec. The example depicted below shows the use of structure as an argument, conditional statements, loops, sub-sequence call, external function call, and wait statement.

![Figure 3: A sample sequence in ISequenceSpec Excel version](image-url)
The following is C code generated for this complex sequence:

```c
int initial_seq(order_struct *os, int size, int clear_value)
unsigned int consolidated_temp_value = 0;
static const int base_addr = 0x0; /*---------- base address ----------*/
static const int set_val = 1;      /*---------- Set value ----------*/
static const int clear_val = 0;      /*---------- Clear value ----------*/
static const int size = 2; /*---------- Size of external sources ----------*/
static const int edge_detect = 1; /*---------- Edge detection ----------*/
int i;
int j;
……
consolidated_temp_value = ((os->word0_bf.priority_order) & PIV_MPICCFG_PRIORD_MASK) | (~((PIV_MPICCFG_PRIORD_MASK) & consolidated_temp_value));
REG_WRITE(PIV_mpiccfg_ADDRESS,consolidated_temp_value);
for (i = 0; i < size;i++) {
    ……
    dim_wr = PIV_meigwclr_ADDRESS+PIV_meigwclr_SIZE*i;
    REG_WRITE(dim_wr,0);
    consolidated_temp_value = ((0x00000000) & PIV_MEIVT_BASE_MASK) | (~((PIV_MEIVT_BASE_MASK) & consolidated_temp_value));
    REG_WRITE(PIV_meivt_ADDRESS,consolidated_temp_value);
    if( external_src[i] == 1){
        // Including Sequence :: operation_seq
        ……
        exintsrc_req = 1;
        if( exintsrc_req == 1){
            intr_req = 1;
        }
    }
}
```

---

**Figure 4: An example of a complex sequence**

<table>
<thead>
<tr>
<th>sequence name</th>
<th>ip</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial_seq</td>
<td>PIV</td>
<td></td>
</tr>
<tr>
<td>arguments</td>
<td>value</td>
<td>description</td>
</tr>
<tr>
<td>order_struct</td>
<td>1</td>
<td>Structure passed as an argument to the sequence</td>
</tr>
<tr>
<td>size</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>clear_value</td>
<td>0</td>
<td>clear value</td>
</tr>
<tr>
<td>constants</td>
<td>value</td>
<td>description</td>
</tr>
<tr>
<td>base_addr</td>
<td>0x0</td>
<td>base address</td>
</tr>
<tr>
<td>external_src[0]</td>
<td>(0,0)</td>
<td>external source</td>
</tr>
<tr>
<td>exintsrc_req</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>interrupt_srcs</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

---

**command**

- **write**: mpi cfg.prior order = os->priority_order
- **write**: meigwctrl[i].polarity = 1
- **write**: meigwctrl[i].type1 = 0
- **write**: meivt.base = base_addr
- **write**: operation_seq = Subsequence call = operation 1
- **write**: meii[i].priority1 = 1
- **write**: meii[i].inten = 1

---

**sequence name**

- **intial_seq**: (order_struct *os, int size, int clear_value)

---

**function**

- **unsigned int consolidated_temp_value = 0**
- **static const int base_addr = 0x0**
- **static const int set_val = 1**
- **static const int clear_val = 0**
- **static const int size = 2**
- **static const int edge_detect = 1**

---

**variables**

- **int i, j**

---

**constants**

- **base_addr**: 0x0
- **set_val**: 1
- **clear_val**: 0
- **size**: 2
- **edge_detect**: 1

---

**functions**

- **write**: mpi cfg.prior order = os->priority_order
- **write**: meigwctrl[i].polarity = 1
- **write**: meigwctrl[i].type1 = 0
- **write**: meivt.base = base_addr
- **write**: operation_seq = Subsequence call = operation 1
- **write**: meii[i].priority1 = 1
- **write**: meii[i].inten = 1
Replicating the same scenario on various platforms (such as UVM, System Verilog, MATLAB®, etc.) without ISequenceSpec can prove to be a very tedious task. Users will have to manually write long sequences, which can be very time consuming. So, having a single specification format can ensure synchronization between teams during various stages of development, thus saving time and improving efficiency.

Moreover, for ease-of-use, various validation checks are performed before the generation of outputs. This enables the user to debug the specification before implementing it in inFact. Several syntax or format problems can occur during the development of a sequence plan in the spreadsheet. These issues are reported as errors and warnings while creating specifications.

The entire spreadsheet is validated and a report containing a complete list of all issues opens in a window for debugging purposes. It guides the user to the row containing a problem, where the user can fix the issue and run the validation again. The following checks are carried out in the validation process:

- **Register Validation** – Guides user to the row containing the register data issue
- **Syntax Validation** – Sequence step validation is performed, which checks existence of variables, arguments, in-line functions, etc.

Below are some examples of the sequence code and the generated exec blocks:

**SV sequences:**

```sv
task memTx (int address, int Data);
    uvm_reg my_reg;
    uvm_status_e uvm_status;

    my_reg = rm.default_map.get_reg_by_offset(address);
    my_reg.write(uvm_status, Data);
    // UVM trnxn Function made by IDesignSpec
endtask:

task memRx (int address);
    uvm_reg my_reg;
    uvm_status_e uvm_status;
    uvm_reg_data_t read_data;

    my_reg = rm.default_map.get_reg_by_offset(address);
    my_reg.read(uvm_status, read_data);
    // UVM trnxn Function made by IDS
endtask:
```

Code 3: The above code is a sample of SV sequences generated by ISequenceSpec
ISSUES WITH THE CURRENT SOLUTION

As per the PSS 1.0a standard, there are a few issues with the exec block interface. These are:

- Lack of return value – Currently there is no way to return a value from the exec block back into a PSS action. Sometimes the flow of the PSS can change based on the execution of the exec block.
- Data types – Only scalar (numeric/enumerated/Boolean) and string variables can be referenced in a target-template exec block. This is a problem because often large structures may need to be passed to the exec blocks.
- Parameterizationenums/defines – Sharing of parameters, enums, and defines can prove useful for exec block sequences.

Considering the huge stride the industry has made with the PSS and golden specification for sequences, these issues are minor and will hopefully be fixed in the next version of PSS.
CONCLUSION

PSS is useful for SoC high-level test scenario creation; the IP level details are currently handled using exec blocks. Currently, users have to manually write long and complex sequences that deal with the registers and pin manipulation commands.

Interfacing inFact with ISequenceSpec can be beneficial for designers, as it truly makes the test intent portable not just at a high level but all the way to implementation. This end-to-end automation helps increase the efficiency and reduce the time for writing the tests. By using inFact and ISequenceSpec together, users can achieve the following benefits:

• Capture sequences in a golden spec that synchronizes the entire SoC team, making the process much more efficient

• Based on the target platform, generate the appropriate test realization code or implementation-level sequences in the target language (C, UVM, or SV) that enable register R/W and pin manipulation commands

REFERENCES


The basis of this article was derived from practical experience. The scenario was this:

“Here is a DUT specification, we have no UVM environment for you to start with as a template, so go and find out how to generate one with Mentor’s UVM Framework (UVMF) template generation methodology.”

The Mentor UVMF documentation and examples provided great direction on how to generate a UVMF framework from scratch via, in this case, Python scripting. And then, boom, in fairly short order and from an architectural standpoint, you are suddenly presented with a UVM framework that has hundreds of files and a large directory structure consisting of tests, sequences, transactions, drivers, monitors, predictors, scoreboards, configuration, and coverage. All connected, compile-able, and simulate-able… and a bit overwhelming.

So, what do you do next? The generated tests have no idea what is in your DUT or DVE (Design Verification Environment) functional specifications, nor does it know how to get stimulus all the way to the DUT from the top test. Also, your predictors do not know what to expect and are not ready to support score-boarding. Of course, there are other customization requirements such as synchronizing the drivers and monitors to the DUT and the framework itself which is quite well covered in the existing Mentor documentation AND all good subjects for white papers none-the-less. This article will concentrate on one of these areas: how to customize the front end Test Control structure. It will utilize, specifically, the primary Mentor UVMF tutorial called “Generator Tutorial” as the base example. This article is really a “how to” guide vs. a technical dissertation, and should be immediately useful to verification folks with limited UVM background engaged in something similar to the stated scenario at the beginning of this text. Additionally, these techniques should be directly applicable to any UVM template generated by Mentor’s UVMF template generator methodology.

**INTRODUCTION**

Considering the UVMF automated scripting,

“The python scripts provided automates the creation of the files, infrastructure and interconnect for interface packages, environment packages and project benches. Once generated, developers can promptly focus on adding functionality specific to the design and interfaces used”\[1\]

The latter part of this quote becomes the central point of interest here: customizing the generated UVM Framework.

Once the generator tutorial guide is digested and the example is working, the substance of this article can be readily adapted thereon. In particular, the task becomes how to customize the front end of the environment in order to have manual control of the ALU inputs at the test level. In the published Generator Tutorial, all of the ALU inputs are randomized at the input sequence to the ALU in driver. This article will demonstrate how to extend that stimulus control out to top level tests, via adding/extending sequences/tests between the top test and the ALU in driver.

Therefore, this article will consist of a step-by-step process that will include the addition of new files to the tutorial UVMF file set, as well as modifications to existing UVMF files. All changes/additions were integrated and successfully simulated. Figure ALU DUT Block Diagram, on the opposite page, shows the simple ALU DUT with three inputs and one output. The modifications/additions to the DVE include manual control of the ALU “A” and “B” inputs only, while the “Operator” input was left randomized.

Section 8 features a list of affected files with their paths included, and can be obtained at TBD if desired. However, there is enough detail in this article to perform the edits manually.
BACKGROUND

The focus of this article starts where the Mentor document “ALU UVM Framework Step By Step Guide”[2] ends. The guide does an excellent job of detailing the UVMF ALU example, the basis for the Generator Tutorial.[3] The tutorial demonstrates, to a user with a limited amount of UVM experience, how to successfully create a UVMF verification environment basically from scratch. This creation of a UVM environment, via Python or YAML scripting, is remarkable given 1) the vast amount of UVM factory files created to support a complete UVM environment, and 2) the mere fact that most UVM work has been traditionally based on an [existing] architected UVM template, of some sort created by somebody, to start with. Instead, we are starting with no testbench code of any type.

So, to go from scratch to an executable UVM environment with UVMF automated scripting allows possibly 80% of the bulk of the UVMF code to be in place in short time. However, getting the rest of the custom code in place for your DUT to see any activity takes much more time, as the “smarts” that gets contextual stimulus to the DUT pins must be understood and integrated. For example, the default stimulus provided by the generator tutorial, initially, results in a not quite complete path to the top test control that would allow customized stimulus insertion. Instead, the stimulus is generated further downstream via UVM randomization, which works absolutely fine for this simple ALU example.

If your DUT requires specific (non-random) stimulus patterns you must either create this at the BFM driver site (not good UVM practice), or allow data manipulation from somewhere upstream of the BFM within the UVM architected test, sequences, and/or transactions. Again the focus of this article will be that of inserting manual stimulus control at the top test control level.

TOP TEST CONTROL STRUCTURE

The Figure, Top Test Control Structure on the following page, depicts the customized implementation for manual manipulation of the tutorial’s ALU A and B input stimulus from the top test. The top green block, alu_manual_test, is the new extended test with SystemVerilog assignments to the stimulus, implying manual top test control. The connected blue blocks, alu_manual_sequence and alu_in_manual_sequence, are the two new sequences required to get the manual stimulus from the top test to the existing alu_in_transaction. Thereon, this transaction terminates at the target ALU Driver Agent, in particular the “alu_driver” which then allows a proxy access to the “alu_driver_bfm”, within the same agent, to get stimulus to the DUT I/O.

Of particular note:

- Since there is no DUT clock or data strobe of any sort, a finite “wait” delay is inserted in the alu_manual_sequence which provides a mechanism for data synchronization

- It requires two contiguous sequences to get stimulus from the top level alu_manual_test, through both manual sequences, onto the alu_in_transaction, and finally to the ALU Driver Agent. This is primarily because the UVMF inherent architecture, which is python template generated as UVMF, consists of two separate distinctions of “project_bench” and “verification_ip” containers. This is readily evident in the
directory structure naming convention as observed @ generator_tutorial/uvmf_template_output/project_benches/verification_ip

CREATING THE NEW TEST

Creating a new UVMF test actually becomes quite easy by “extending” the top test. The basics of doing this allows the original test class constructs and attributes to be extended automatically to the new test. The technique being demonstrated here will involve copying/modifying an existing test that, itself, has already been extended from top_test. The following is the method to create our new test:

1. Copy alu_random_test.svh to alu_manual_test.svh and make the following changes to the new test.

2. Change the text “random” to “manual” throughout the file.

3. Create the manual sequence handle. Later you will create the alu_manual_sequence, however here you must create a handle to it to use as a pointer within the test run_phase.

```verilog
class alu_manual_test extends test_tip;

`uvm_object_utils (alu_manual_test);
//Create manual sequence handle
alu_manual_sequence alu_manual_seq;

virtual function void build_base (uvm_phase phase);
alu_bench_sequence_base::type_id::set_type_override(alu_manual_sequence #(8)::type_id::get_type(0);
super.build_phase (phase);
endfunction // build_phase
```

4. Ensure that the alu_bench_sequence_base has been over-ridden with the alu_manual_sequence in the build phase.

5. Add the task run_phase. The task run_phase must first create the alu_manual_sequence which can then be used to assign manual stimulus to the “a” and “b” inputs of the ALU. In this example, a simple Verilog “repeat” statement, with an incrementing stimulus and UVM start command allows the run_phase of the test to start, run, and complete via the UVM raise_objection and drop_objection constructs.

```verilog
virtual function void build_base (uvm_phase phase);
alu_bench_sequence_base::type_id::set_type_override(alu_manual_sequence #(8)::type_id::get_type(0);
super.build_phase (phase);
endfunction // build_phase
```

```verilog
virtual function void run_phase (uvm_phase phase);

phase.raise_objection (this, "Top Test");
`uvm_info(get_name (), "Starting alu_manual_test", UVM_LOW)

// Create the top manual sequence to alu_manual_sequence which drives the ALU a & b inputs.
alu_man_seq = alu_manual_sequence #(8)::type_id::create ("alu_manual_seq", this);
```

//Add the UVM run phase to allow manual manipulations of ALU a & b inputs
virtual task run_phase (uvm_phase phase);

phase.raise_objection (this, "Top Test");
`uvm_info(get_name (), "Starting alu_manual_test", UVM_LOW)

// Create the top manual sequence to alu_manual_sequence which drives the ALU a & b inputs.
alu_man_seq = alu_manual_sequence #(8)::type_id::create ("alu_manual_seq", this);
6. Add the new file name to `alu_test_pkg.sv` file. This is a simple `include` statement of the file name so it gets included in the compile.

CREATING THE NEW BENCH SEQUENCE

Creating a new UVMF project bench sequence, similar in process to the new test created in the previous section, is quite easy by “extending” the top bench sequence. The basics of doing this allows the original project bench class constructs and attributes to be extended automatically to the new sequence. The technique being demonstrated here will involve copying/modifying an existing project bench sequence that, itself, has already been extended from `alu_bench_sequence_base`. The following is the method to create our new project bench sequence:

1. Copy `alu_random_sequence.svh` to `alu_manual_sequence.svh` and make the following changes to the new sequence.
2. Change the text “random” to “manual” throughout the file. Importantly this results in creating the `alu_in_agent_manual_seq` at the top of the task body.
3. Add the logic “a” and “b” declarations for this sequence prior to the task body.

```vhdl
// Initialise the ALU a & b inputs
alu_manual_seq.a = 0;
alu_manual_seq.b = 1;

// Manually manipulate values for a & b start the sequence
// The op input remains randomized downstream
in alu_manual_sequence
repeat (100)
begin
  alu_manual_seq.a = (alu_manual_seq.a) + 1;
  alu_manual_seq.b = (alu_manual_seq.b) + 1;
  alu_manual_seq.start (null);
end
phase.drop_objection(this, "alu_manual_sequence");

// Assign the ALU a & b inputs
alu_in_agent_manual_seq.a = a;
alu_in_agent_manual_seq.b = b;

// Default to only one start per sequence
repeat (1) alu_in_agent_manual_seq.start
(alu_in_agent_sequencer);
```

4. Add the ALU input logic assignments to the new sequence and default to only one start per sequence. This allows the repeat statement in `alu_manual_test.svh` to control the length of the simulation.

```vhdl
// Add logic for passing ALU data in
logic [ALU_IN_OP_WIDTH-1:0] a=0;
logic [ALU_IN_OP_WIDTH-1:0] b=0;
```

5. Add the new file name to `alu_sequences_pkg.sv` file. This is a simple `include` statement of the file name so it gets included in the compile.

CREATING THE NEW VERIFICATION IP SEQUENCE

Creating a new UVMF verification IP sequence, similar in process to the new bench sequence created in the previous section, is quite easy by “extending” the top verification IP sequence. The basics of doing this allows the original verification IP class constructs and attributes to be extended automatically to the new sequence. The technique being demonstrated here will involve copying/modifying an existing verification IP bench sequence that, itself, has already been extended from `alu_in_sequence_base`. The following is the method to create our new project bench sequence:

1. Copy `alu_in_random_sequence.svh` to `alu_in_manual_sequence.svh` and make the following changes to the new sequence.
2. Change the text “random” to “manual” throughout the file. Notice that you do not have to create another sequence since the downstream UVMF element becomes the existing `alu_in_transaction`.
3. Add the logic “a” and “b” declarations for this sequence prior to the task body.

```vhdl
// Initialise the ALU a & b inputs
alu_in_agent_manual_seq.a  =  a;
alu_in_agent_manual_seq.b  =  b;
```

```vhdl
// Add logic for the sequence variables, inputs to the ALU
logic [ALU_IN_OP_WIDTH-1:0] a=0;
logic [ALU_IN_OP_WIDTH-1:0] b=0;
logic allow_reset=1’b1;
```
4. Add the ALU input logic assignments to the `alu_in_transaction` to get the manual stimulus to the `alu_in_driver`.

```vhdl
// Assign the manual data, passed from the top alu_manual_sequence // This assigns the a & b data to the alu_in_driver areq.a = a; areq.b = b;
```

5. Add this sequence to the `alu_bench_sequence_base.svh` file (as it is used by the project bench sequences upstream) so they are available via extension of the project bench base sequence.

```vhdl
// add verification IP manual sequence typedef alu_in_manual_sequence alu_in_agent_manual_seq_t; alu_in_agent_manual_seq_t alu_in_agent_manual_seq;
```

6. Add the new file name to `alu_in_pkg.sv` file.
   This is a simple `include` statement of the file name so it gets included in the compile.

---

# MODIFYING THE SIMULATION MAKEFILE AND RUN

The following modifications are required, of the sim/Makefile, in order to run Make to build and execute the UVMF simulation with the new `alu_manual_test` in your area.

1. Define the path to your QuestaSim UVMF home area. Note this requires the versions, **minimally**, that are shown for both Questa (10.7b) and UVMF Framework (3.6g). Note that the original code (commented out) searched for the UVMF area with wildcards, but does not specify a UVMF version, thus the export of a hard path to ensure version compatibility.

```makefile
# UVMF library directory: # This variable points to the UVMF release where uvmf_base_pkg_directory resides. # This variable points to the release code that is not user modified. # This variable allows for the UVMF release directories to reside independent of project related verification_ip and project_benches directories. # This code below looks "upward" for directory starting with UVMF_* and returns first match for use with the release examples. #/UVMF_HOME = $(firstword $(wildcard $(addsuffix / UVMF_*,. .. ../.. ../../.. ../../../.. ../../../../..))) export UVMF_HOME = C:/questasim64_107b/examples/ UVM_Framework/UVMF_3.6g
```

2. Set the TEST_NAME variable to the new test.

```makefile
# Set test case specific Variables TEST_NAME   ?= alu_manual_test
```

This completes the modifications and now the simulation can be run by executing, in the sim directory:

```
“make cli” -run with out Questa GUI
“make debug” -run with Questa GUI
```

# MODIFIED/NEW FILE LIST

The following is a list of the generator_tutorial files that were either modified or added to complete the top test control functionality.

A. Modified Files:

- `generator_tutorial/uvmf_template_output/project_benches/alu/tb/tests/alu_test_pkg.sv`
- `generator_tutorial/uvmf_template_output/project_benches/alu/tb/sequences/src/alu_bench_sequence_base.svh`
B. Added (New) Files:

• generator_tutorial/uvmf_template_output/project_benches/alu/tb/sequences/src/alu_manual_sequence.svh

CONCLUSION

Becoming proficient at architecting a UVM (i.e. no “F” appended to this acronym) environment requires an extremely steep learning curve by all accounts, even for those folks who have a strong background in traditional SystemVerilog testbench design. With Mentor’s UVMF template generation methodology, a UVM environment is readily achievable for Verification Engineers that may have little to no experience in UVM technology, but possess the architectural mindedness for good testbench design from the onset. At this point, it becomes a reasonable and manageable task to put in place the “smarts” to turn a UVMF factory implementation into a working DVE which has bonified UVM structure.

The “front end” customization presented in this article can be directly applied to completing the other noted parts of the UVMF environment, with help from Mentor’s examples and documentation. This may take some further experimentation (simulation breakpoints and inserting experimental code cannot be emphasized enough here!), as was necessary in the given scenario that resulted in the basis of this empirical article. It is surprising to hear of large, established corporations investing incredible time and money into attempts at creating UVM by hand, and ultimately failing.

UVMF can get you 80% there (well maybe the code base anyways, and from there it is truly within the grasp of the Verification Engineer to create a customized and useful UVMF environment.

Go UVMF!!!!!!

REFERENCES

[1] “UVM_Framework_Users_Guide.pdf”, Rev 3.6g.0, Mentor, A Siemens Business. @questasim64_10.7b/examples/UVM_Framework/UVMF_3.6g/docs

[2] “ALU_UVMF_Step_By_Step_Guide.pdf”, August 2017, Mentor, A Siemens Business. @questasim64_10.7b/examples/UVM_Framework/UVMF_3.6g/docs/generator_tutorial

[3] “Generator Tutorial”, a Mentor UVMF example demonstrated in Reference 2, located at: @questasim64_10.7b/examples/UVM_Framework/UVMF_3.6g/docs/generator_tutorial
The Most Comprehensive Resource for Verification Training

33 Video Courses Available Covering
- DO-254
- UVM Framework
- UVM Debug
- Portable Stimulus Basics
- SystemVerilog OOP
- Formal Verification
- Metrics in SoC Verification
- Verification Planning
- Introductory, Basic, and Advanced UVM
- Assertion-Based Verification
- FPGA Verification
- Testbench Acceleration
- Power Aware Verification
- Analog Mixed-Signal Verification

UVM and Coverage Online Methodology Cookbooks
Discussion Forum with more than 10,000 questions asked
Verification Patterns Library

www.verificationacademy.com