Building a Better Virtual Sequence with Portable Stimulus

by Matthew Ballance — Mentor, A Siemens Business

When using the Universal Verification Methodology (UVM), sequences are the primary mechanism by which stimulus is generated in the testbench. Sequences come in two flavors: simple sequences for driving a single interface, and virtual sequences that control more complex behavior. Simple sequences tend to work with a single sequence item, while virtual sequences often spawn off multiple sub-sequences to accomplish their intended task. Good virtual sequences are challenging to create, and even more challenging to reuse in a way not explicitly intended by the original author. Portable stimulus can make creating virtual sequences easier, increase the verification value achieved by running these virtual sequences, and enable more reuse of the description used to create the virtual sequence. This article will walk through an example showing how portable stimulus applies to creating scenarios for a DMA engine.

DMA ENGINE OVERVIEW

The DMA engine used in this article is a relatively simple 8-channel DMA engine. It has a register interface for programming DMA transfers, and two master interfaces for the DMA engine to use for transferring data. Each DMA channel can either perform direct memory-to-memory transfers, or can use peripheral handshake signals to transfer data from memory to a peripheral device, or from a peripheral device to memory.

SIMPLE SEQUENCES

A simple sequence for the DMA engine might look as simple as what is shown in Figure 2.

```vhdl
class wb_dma_rand_single_transfer_seq extends wb_dma_transfer_seq;

start_item(desc);
if (!desc.randomize())
    `uvm_fatal(get_name(), "Failed to randomize sequence item");

finish_item(desc);
endtask
```

Figure 2: Simple UVM Sequence
This simple sequence simply randomizes a sequence item that represents a DMA transfer, then sends that sequence item to the driver to program registers, etc. A simple sequence like this will use a sequence item, like that shown in Figure 3, containing random fields and constraints appropriate to describe a DMA transfer on a given channel.

```
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;
    rand bit[11:0] tot_sz;
    rand bit[2:0] trn_sz;
    rand bit[8:0] chk_sz;
    rand bit[31:0] transfer_sz;

    constraint trn_sz_c {
        trn_sz inside {1, 2, 4};
    }

endclass
```

However, our simple sequence starts to hit limitations when we want to test sequences of operation or operations in parallel. For example, what if we wanted to launch two transfers in parallel to exercise 2 DMA channels at the same time? Without some guidance, we would likely have a mess since more than one of our simple sequences would target the same DMA channel. We could, for example, use a queue to randomly assign channels to the transfers as we set them up, as shown in Figure 4.

```virtual task body();
    wb_dma_single_transfer_descriptor d1, d2;

    int channels[] = {0, 1, 2, 4, 5, 6, 7};

    repeat(10) begin
        channels.shuffle();

        d1 = wb_dma_single_transfer_descriptor::type_id::create("d1");
        d2 = wb_dma_single_transfer_descriptor::type_id::create("d2");

        d1.channel = channels[0]
        d2.channel = channels[1]

        fork
            begin
                start_item(d1);
                finish_item(d1);
            end
            begin
                start_item(d2);
                finish_item(d2);
            end

        join
    end
```

Even a simple sequence provides us significant verification value in comprehensively verifying some aspects of a design. Our simple DMA transfer sequence allows us to comprehensively verify the operation of a simple DMA engine channel. We can add in functional coverage to specify the key aspects of DMA-channel operation that we want to prioritize, and be able to confirm that we’ve verified them. Using Questa® inFact, we can achieve these transaction-level coverage goals much more efficiently than we could with pure random stimulus.

This approach is very typical of the directed-random style of virtual sequences. We use procedural methods to avoid conflicts between lower-level transactions that carry out our high-level test intent. This allows us to get some randomness in our scenarios, while following the rules of the system - in this case, the fact that parallel DMA channels must use different channels. While this directed-random approach solves our immediate problem, this code isn’t very reusable or scalable.
For example, we cannot create a new sequence that extends from this sequence but disables the use of certain channels. The declarative nature of portable stimulus allows us to do exactly this type of reuse.

CRAFTING A PORTABLE STIMULUS VIRTUAL SEQUENCE

Let’s take a look at how we can model a virtual sequence with Portable Stimulus for our DMA engine. Portable stimulus encapsulates behavior in actions. Our first task is to take a step back and consider which actions we need to create. Our DMA engine can perform three logical functions:

- Transferring data between two regions of memory
- Transferring data from memory to a peripheral device
- Transferring data from a peripheral device to memory

We will represent each of these operations as a PSS action that we can then use in a scenario.

The PSS language allows actions to declare input and output ports, which allow the action to specify what data must be present when it executes, and what data it provides for the use of other actions. One of the first things we need to consider when modeling behavior as PSS actions is what inputs the behavior requires and what outputs it provides.

The memory-to-memory action reads data from a source location in memory and writes it to a destination location. This indicates that we should have an input to represent the source location and an output to represent the destination location. The mem2dev and dev2mem actions will, respectively, read from a region of memory and write to a region of memory. This indicates that they should, respectively, have an input and an output. But, how should we represent the device that these actions interact with? We could represent the device address using an input and output as well, but in this case we will not because we know that the device address is closely linked with the DMA channel used by the mem2dev and dev2mem actions. The device address will be a function of the system address map and the DMA channel in use, not something the test-scenario writer will manipulate directly.

Figure 5 shows our DMA action primitives in diagram form.

```plaintext
Figure 5: DMA Primitive Actions
```

Figure 6 shows the PSS description of our memory-to-memory action. The device-to-memory and memory-to-device actions look very similar, and will not be shown.

```plaintext
/** *
 * Memory-to-memory transfer
 */
action mem2mem_a : dma_dev_a {
  input data_ref_mem_b  dat_i;
  output data_ref_mem_b  dat_o;

  constraint {
    dat_i.sz <= 4096;
    (dat_i.sz % trn_sz) == 0; // Size must be a multiple of the transfer size
    (dat_i.addr % trn_sz) == 0; // Source and dest addresses must be aligned
    (dat_o.addr % trn_sz) == 0;
    dat_i.sz == dat_o.sz;
    dat_i.ref == dat_o.ref;
  }
}
```

Figure 6: Memory-to-memory transfer action

In addition to declaring the inputs and outputs of the action, we must specify the constraints that govern its operation. Specifically, we must specify that the size
of data transferred must be 4k or less – a constraint imposed by the DMA engine. The address of both the source and destination address must be aligned to the DMA’s transfer size as well. Note that the action declaration doesn’t specify anything about how the DMA will be programmed. The action only contains the high-level rules on a DMA transfer. We will add in the mapping to our UVM environment later.

CREATING THE SCENARIO

Now that we have low-level actions that represent the core operations the DMA engine can perform, we can assemble those actions into a compound action that carries out a scenario. The scenario we started with was running two DMA transfers in parallel on different channels. An equivalent scenario is shown in Figure 7 below.

```pss
component parallel_xfer_c {
    // Ensure parallel operations have unique channels
    constraint unique {
        m2m_1.channel, m2m_2.channel,
        d2m_1.channel, d2m_2.channel,
        m2d_1.channel, m2d_2.channel
    }

    activity {
        bind m2m_1.dat_i alloc.a0;
        bind m2m_2.dat_i alloc.a1;
        bind m2d_1.dat_i alloc.a2;
        bind m2d_2.dat_i alloc.a3;

        repeat (10) {}
        parallel {
            select {
                m2m_1;
                d2m_1;
                m2d_1;
            }
        }
    }

    select {
        m2m_2;
        d2m_2;
        m2d_2;
    }
}
```

Figure 7: PSS Parallel-Transfer Scenario

The first portion of the description creates instances of our low-level actions for performing DMA transfers. After creating action instances, we create a constraint to ensure that the channel selected for the DMA operations that eventually run will be different. Next, the activity block composes a scenario from the action instances. The parallel construct in PSS makes it just as easy to describe parallel behavior as in SystemVerilog, and the declarative nature of a PSS description enables us to write constraints that apply across the procedure of our test scenario.

Our PSS test scenario weighs in at 40 lines of code, certainly on-par with the roughly 50 lines required for our SystemVerilog sequence, showing that PSS provides a succinct way to capture scenarios.

COVERAGE

One challenge with directed-random sequences is that it’s difficult to tell whether we’ve generated the stimulus combinations we really care about. Portable stimulus provides a covergroup construct, just like SystemVerilog does. We can add a covergroup to our portable stimulus model to ensure, for example, that we generate all possible scenarios across our parallel scenarios in our virtual sequence (Figure 8 on the following page).
Figure 8: Adding Coverage

PSS currently supports data-centric coverage, so we still need to represent the scenarios we need to cover as a data relationship. However, here again, the declarative nature of PSS makes it much simpler to capture the coverage relationships we care about. We add in two enumerated-type variables (scen1 and scen2) to represent the scenario being executed, and add constraints to the activity to force these variables to the appropriate value given the scenario being executed. This adds a few lines to our scenario, but will allow us to ensure that we exercise all the scenarios.

CONNECTING TO THE TESTBENCH

Thus far, we have focused on test intent - the high-level view of what we want to test. In order to actually run traffic in our testbench environment, we need to run sequences or call APIs in SystemVerilog. PSS provides the exec construct to connect the high-level test intent described in PSS to the low-level test realization described in SystemVerilog, C, or any number of other implementation languages.

Figure 9: Connecting PSS to SystemVerilog

The exec block shown in Figure 9 provides a mapping between the fields of the mem2mem_a action and a task in our SystemVerilog virtual sequence named wb_dma_dev_mem2mem. This task is responsible for programming the DMA engine to carry out a transfer on the selected channel.

CUSTOMIZING OUR VIRTUAL SEQUENCE

Now, let’s look at the original challenge we faced with our hand-coded virtual sequence: disabling the use of one of the channels. PSS provides us several ways to customize a scenario. The simplest might be to just create a new top-level scenario that inherits from our existing action, and add constraints to force the base action to not use a specific channel (Figure 10). This allows us to reuse our existing scenario, while customizing its behavior.

Figure 10: Customizing the Scenario with Inheritance

```cpp
eenum scenario_e { m2m, d2m, m2d }

val action parallel_xfer_a {
    // ...
    action scenario_e scen1, scen2;

covergroup {
    scen1_cp : coverpoint scen1;
    scen2_cp : coverpoint scen2;

    scen_1x2 : cross scen1_cp, scen2_cp;
} // ...

activity {
    // ...
    repeat(10) {} parallel {
        sequence {
            select {
                m2m_1; constraint scen1==m2m;
                d2m_1; constraint scen1==d2m;
                m2d_1; constraint scen1==m2d;
            }
            scen1;
        }
    }
} // ...
}

extend action wb_dma_c::mem2mem_a {
    exec body SV = """"
    wb_dma_dev_mem2mem({devid}), {{channel}}, {{dat_i.addr}},
    {{dat_o.addr}}, {{dat_i.sz}}, {{trn_sz}});
    """
}

val action parallel_xfer_no_ch0_a : parallel_xfer_a {
    constraint {
        m2m_1.channel !0; m2m_2.channel !0;
        d2m_1.channel !0; d2m_2.channel !0;
        m2d_1.channel !0; m2d_2.channel !0;
    }
}
```
The approach we just showed for customizing the scenario targets a specific instance of the scenario. What if we needed to ensure that all instances of the dev2mem action, wherever they appeared in the scenario, never used channel 5? PSS provides a type-extension mechanism that allows us to layer in constraints that will apply to all instances of a given type, as shown in Figure 11.

```extend action wb_dma_c::dev2mem_a {
  constraint channel != 5;
}
```

*Figure 11: Customizing the Scenario with Extension*

The advantage (and the disadvantage) of this approach is that the new constraint will apply to all instances of the specified type. In some cases, this is exactly what we want. In other cases, we want to be more targeted. PSS also provides a factory-like mechanism that allows us to override types in very specific contexts.

**REUSE BEYOND THE BLOCK LEVEL**

The atomic actions that we created, such as mem2mem and dev2mem, are necessary to support our block-level scenarios. However, there’s nothing tethering them to our block-level UVM environment except the UVM-specific implementation. PSS makes that simple enough to change with another exec block that maps our test intent to a test realization implemented in C (Figure 12).

```extend action wb_dma_c::mem2mem_a {
  exec body C = ""
  wb_dma_dev_mem2mem({{devid}}, {{channel}}, {{dat_i.addr}},
  {{dat_o.addr}}, {{dat_i.sz}}, {{trn_sz}});
  "\n",
}
```

*Figure 12: Mapping to Test Realization in C*

The declarative nature of PSS, coupled with appropriate language-specific mapping, makes it very easy to reuse test intent from block level when creating SoC-level scenarios. It doesn’t make sense to reuse all the PSS content we create at block level, since many of the scenarios will test functionality that is specific to block level. However, there are certainly opportunities for reuse at SoC level – infinitely more than with pure SystemVerilog UVM sequences!

**CONCLUSION**

Portable stimulus is often seen as being specific to SoC-level testing. But, as this article has shown, portable stimulus can be very helpful in block-level testing to make scenario creation with UVM virtual sequences simpler, more effective, and makes the scenarios we create more reusable. And, we can get these benefits with roughly the same number of lines of code as are required to capture much simpler UVM sequences.
VERIFICATION ACADEMY
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
- PowerAware 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