Results Checking Strategies with Portable Stimulus

Tom Fitzpatrick
Mentor, a Siemens Business
tom_fitzpatrick@mentor.com

Matthew Ballance
Mentor, a Siemens Business
matt_ballance@mentor.com

INTRODUCTION

Electronic design and verification automation have historically been driven by domain-specific languages. The Accellera Portable Stimulus Working Group has released the Portable Test and Stimulus Standard (PSS)[1] to enable capturing of high-level test behavior, including stimulus and coverage goals, to satisfy the requirements of users in multiple verification domains and that can be targeted to verification environments from UVM to tests running on the processors of an SoC. There are also existing Portable Stimulus tools in the industry that implement taking a high-level test specification and targeting that to multiple verification environments.

PSS allows the user to create a declarative specification of critical behaviors – called *actions* – that must be performed by the system-under-test, and the scheduling, resource and data constraints between them. From these specifications, automation may be applied to allow tools to generate multiple randomized scenarios from the original specification. In addition to scheduling, resource and data flow constraints, each leaf-level action in a PSS model also includes templated or procedural interface code – an *exec block* – that implements the abstract behavior on the desired target platform.

This idea of "scenario-level randomization" builds on strategies that UVM users have employed for years and applies similar concepts at the higher level of abstraction necessary to model system-level intent. However, whereas UVM deploys monitors and scoreboards to check functional correctness *procedurally*, Portable Stimulus poses some challenges for results checking precisely because the PSS model is abstract and *declarative*, while results checking is often platform-specific.

The key to results checking in Portable Stimulus is to understand that different levels of verification and validation environments have different needs when it comes to results checking. In a block-level environment, we might want detailed scoreboarding in addition to an overall per-operation pass/fail. In an SoC environment, we may only need an overall per-operation pass/fail. The original abstract specification may indicate where in the flow results checking should occur, but the abstract nature of the scenario specification itself necessitates that it rely on the target-specific implementation to actually perform the checks.

PSS MODELING BASICS

Before we can discuss results checking in a PSS model, let us first examine the key elements of a PSS model and how they interact to specify verification intent. As mentioned, a PSS model consists of a set of actions that represent behaviors the system, which includes both the design and the verification environment, must execute. This means that the actions will represent behaviors ultimately executed by some combination of hardware, embedded software and verification components (VIP). Consider a simple UART block under verification as shown in Figure 1.

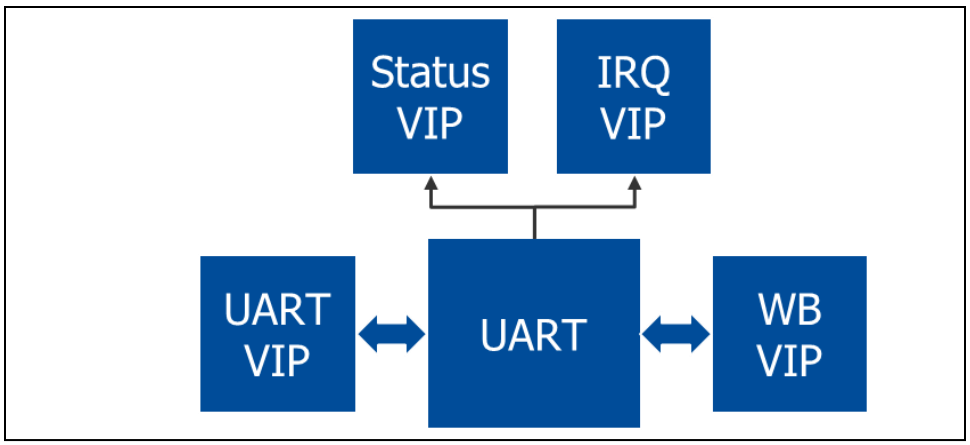

Figure 1: Block-level UART Verification Environment

In a typical block-level environment, the design-under-test (DUT) is surrounded by a set of VIP components at its interfaces to interact with the DUT. The goal in block-level verification is to exercise the block in all possible modes and configurations to ensure correct behavior. Thus, the VIP blocks must each be able to provide the full spectrum of protocol and data options on each interface. Later, when the UART is integrated into a subsystem or system, the other blocks to which it will be connected will, of necessity, conform to a subset of the protocol and data options that were previously verified.

When it comes to modeling the verification intent in PSS, we think not about the specific components in the UVM environment (since we want to reuse the intent in other environments), but rather about the critical operations that must occur to verify the UART:

- The UART VIP will transmit data to be received by the UART
- The UART VIP will receive data sent from the UART
- The UART will receive the data and store the data in memory
- The UART will retrieve data from memory and send the data out

For this particular example, the data transmitted and received by the UART may be transferred using either programmed I/O (PIO) or handshake-based mechanisms. The data protocol on the serial side of the UART demands that the UART VIP actions and the corresponding UART action must execute in parallel. We can represent the UART receive operation graphically as in Figure 2.

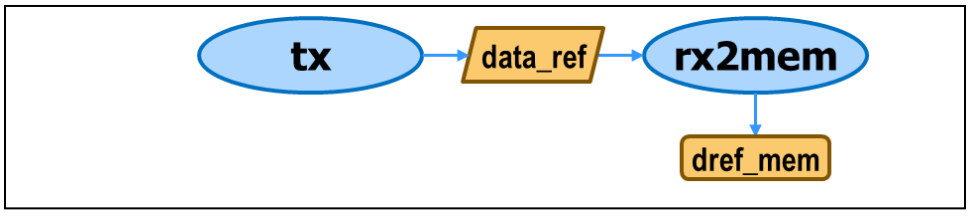

Figure 2: Block-level UART Receive Operations

In PSS parlance, **tx** and **rx2mem** are actions, and **data_ref** and **dref_mem** are *data flow objects*. Specifically, **data_ref** is a *stream* object, which carries the scheduling constraint that its producer (**tx** in this case) and its consumer (**rx2mem**) must execute in parallel, while **dref_mem** is a *buffer* object, which implies "storage" in the system and carries the scheduling constraint that its consumer may not begin execution until the producer completes. Code snippets for the relevant parts of the PSS model are shown starting with Example 1.

> [NOTE: To simplify the graphics, the **data_ref** and **dref_mem** objects in the graphics in Figure 2 correspond to the **data_ref_s** and **data_ref_mem_b** objects, respectively, in the code examples, such as Example 1.]

```
component wb_uart_c : pvm_dev_c {
    action rx2mem_a : pvm_dev_a {
        input data_ref_s          rx_info;
        output data_ref_mem_b     dat_o;

        constraint dat_o.sz > 0;
        constraint dat_o.sz <= 64;
        constraint dat_o.sz == rx_info.sz;
    }
}
```

Example 1: UART Component with Receive-to-Memory Action

The **component** element of PSS is used to encapsulate actions and other elements that correspond to specific parts of the system to facilitate reuse as the corresponding system element is reused in other contexts. In this example, the **wb_uart_c** component is an extension of the base type **pvm_dev_c**, which provides a little infrastructure that will help later, as shown in Example 2.

```
component pvm_dev_c {
    action pvm_dev_a {
        rand bit[31:0]            devid;
    }
}
```

*Example 2: Base Component Type with **devid** field*

The **wb_uart_c** PSS component represents functionality that is ultimately performed by the UART block itself. The important part initially in the PSS model is the data flow, so the UART block will receive the data stream as an input and will store data as an output. We use constraints in the action definition to specify information about the data objects and the relationships between the input and output data.

```
component uart_agent_c : pvm_dev_c {
    action tx_a : pvm_dev_a {
        output data_ref_s   tx_info;
    }
}
```

Example 3: UART VIP Component Transmit Action

In Example 3, we define the transmit action, **tx_a**, that represents functionality to be executed by the UART VIP in Figure 1.

We can then create a PSS scenario in a top-level component to perform the receive operation as in Example 4.

```
component wb_uart_ip_scenarios_c {
    action test_rx_data {
        wb_uart_c::rx2mem_a  uart_rx;
        uart_agent_c::tx_a   uart_agent_tx;

        activity {
            bind uart_rx.rx_info uart_agent_tx.tx_info;
            parallel {
                uart_rx;
                uart_agent_tx;
            }
        }
    }
}
```

Example 4: Compound Action to test UART Receive action

The **activity** statement in the **test_rx_data** action defines the set of critical actions we want to execute and their explicit scheduling. This particular action represents the test intent captured in Figure 2, in which the **uart_rx** (the UART component's **rx2mem_a**) and **uart_agent_tx** (UART VIP's **tx_a**) actions operate in parallel. The **bind** statement shows that the input of the **uart_rx** action will be supplied by the **tx_info** output of the **uart_agent_tx** action. In PSS, constraints are combined across bindings, so the constraints defined in the **rx2mem_a** action in

Example 1 will also apply to the **tx_a.tx_info** output. The first two constraints define the **sz** field of the **dat_o** output to have a value from 1 to 64. The third constraint defines that the **sz** field of the **rx_info** input object is the same as **dat_o.sz**. The **bind** statement in Example 4 enforces the same set of constraints on the **uart_agent_tx.tx_info** output. Ultimately, the scenario defined by **test_rx_data** is for the UART VIP to send a data object of from 1 to 64 bytes in length to the UART and have the UART store it somewhere.

We can, of course, define additional testing actions, including a scenario-level action that may choose between many lower-level actions, as seen in Example 5.

```
action config_transfer_a {
        test_rx_data                test_rx;
        test_tx_data                test_tx;
        action bit[7:0] in [2..5]   n_ops;

        rand scenario_e     scen;
        activity {
                n_ops;              // randomize n_ops

                repeat (n_ops) {
                        scen;       // randomize the scenario
                        select {
                                (scen == scenario_rx): test_rx;
                                (scen == scenario_tx): test_tx;
                                (scen == scenario_rxtx): parallel {
                                        test_rx;
                                        test_tx;
                                }
                        }
                        do end_scenario_a;
                }
        }
}

action entry {
        config_transfer_a           uart_scenario;

        activity {
                repeat (10) {
                        uart_scenario;
                }
        }
}
```

Example 5: Additional Scenario-Level Test Actions

Graphically, this scenario, as represented by the **entry** action, is shown in Figure 3,

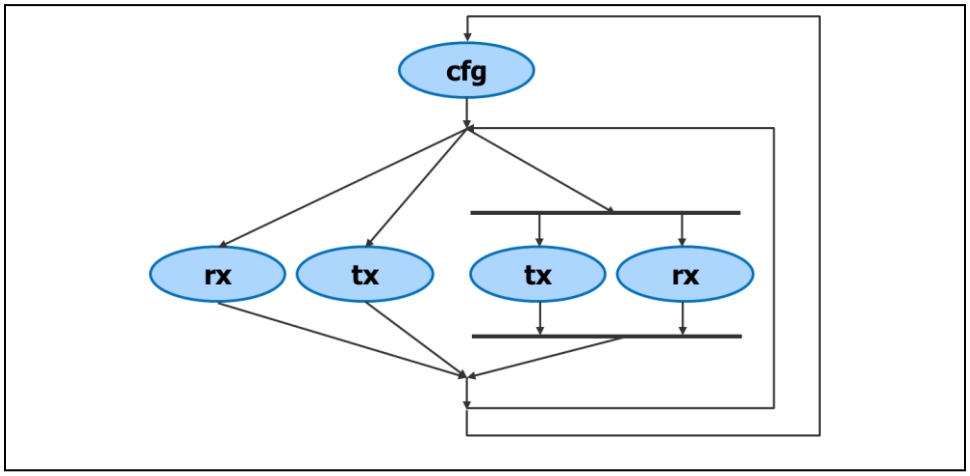

Figure 3: Top-Level Scenario Test Graph

where the test will loop 10 times (in the **entry** action), each iteration (in the **config_transfer_a** action) configuring the UART (action not shown) and then performing a combination of from 2 to 5 randomly-chosen operations, which are either **test_rx_data**, **test_tx_data,** or both **test_tx_data** and **test_rx_data** in parallel. This particular intent specification defines

$$N = 3^{(10 \times n\_ops)}$$

separate scenarios, given the different sequences of actions that may be produced by unrolling the inner loop. Assuming that a generator tool will generate a sufficient number of tests to cover all possible values for the **cfg** action, the number of scenarios would actually be

$$N = 3^{(10 \times C \times n\_ops)}$$

Test Realization in PSS

It is easy enough to understand the test intent conveyed by a PSS model such as the one shown here. One value of PSS is that this abstract model may be mapped to multiple target implementations by extending the PSS model to provide **exec** blocks that specify the detailed implementation of each leaf-level action in the scenario and having a tool stitch these implementations together such that they meet scheduling and data constraints specified in the model. However, when results checking is considered, we must take into account the fact that different test implementations may handle checking in different ways.

Consider first a UVM implementation of the above test, as specified by the **entry** action. It may be implemented in UVM as a virtual sequence that is initiated by the UVM test as shown in Example 6.

```
class wb_uart_pss_vseq_test extends wb_uart_pss_test_base;
        `uvm_component_utils(wb_uart_pss_vseq_test)
        task run_phase(uvm_phase phase);
                wb_uart_pss_vseq_base vseq =
                        wb_uart_pss_vseq_base::type_id::create("vseq");

                super.run_phase(phase);
                phase.raise_objection(this, "Main");
                vseq.start(null);
                phase.drop_objection(this, "Main");
        endtask
endclass
```

Example 6: Top-Level Scenario Test in UVM

where the **wb_uart_pss_vseq_base** type is overridden on the command line by:

```
+uvm_set_type_override=wb_uart_pss_vseq_base,wb_uart_pss_rxtx_transfer_seq
```

The **wb_uart_pss_rxtx_transfer_seq** virtual sequence in UVM thus coordinates the activity in the UVM test environment by invoking sequences in the UVM components. This would typically involve the UART VIP executing a sequence that will randomly generate a set of transactions to the UART and a scoreboard component that can capture the data as it comes out of the UART and compare it to the data sent by the UART VIP. We could model similar functionality in the PSS model, but unfortunately, when we get to a subsystem or system-level environment, particularly a system which may be testing large amounts of data, the idea of maintaining copies of the data throughout the verification environment is infeasible (and often impossible). Rather, an optimal results-checking strategy in Portable Stimulus relies on the underlying implementation to generate and check the data while receiving minimal guidance from the PSS model.

In effect, we must "begin with the end in mind"[2] and take into account the kind of checking that may be available at the system-level, and incorporate that into our test realization strategy. It is almost never a good idea to model the complete data streams in PSS. Instead, we model the data objects used by the UART as in Example 7.

```
package pvm_types_pkg {
	struct data_mem_t {
		rand bit[31:0]		addr;
		rand bit[31:0]		sz;
	}

	buffer data_ref_mem_b : data_mem_t {
		rand bit[31:0]		ref;
	}

	stream data_ref_mem_s : data_mem_t {
		rand bit[31:0]		ref;
	}
}
```

Example 7: Modeling Data in PSS

The most significant aspect of this data model is that there is no explicit "data" field in the PSS model. Instead, each component will use the **ref** field as a key to generate or check data in the underlying realization. For example, instead of just randomizing transaction items, as in Example 8,

```
class uart_serial_tx_seq extends uart_serial_seq_base;
	task body();
		uart_serial_seq_item item =
			uart_serial_seq_item::type_id::create("item");

		for (int i=0; i<data.size(); i++) begin
			start_item(item);
			item.randomize();
			finish_item(item);
		end
	endtask
```

Example 8: Random Sequence in UVM

the UVM sequence in the UART VIP would rely on the controlling virtual sequence to generate a repeatable random stream of data as in Example 9,

```
class uart_serial_tx_seq extends uart_serial_seq_base;
	byte unsigned 		data[$];
	task body();
		uart_serial_seq_item item =
			uart_serial_seq_item::type_id::create("item");

		for (int i=0; i<data.size(); i++) begin
			item.data = data[i];  // repeatable
			start_item(item);
			finish_item(item);
		end
	endtask
	...
endclass
```

Example 9: Non-random Sequence in UVM

which gets called from a SystemVerilog task, as seen in Example 10.

```
task tx(int unsigned seed, int unsigned bytes, int unsigned stride);
        uart_serial_tx_seq  tx_seq  = uart_serial_tx_seq::type_id::create();
        pvm_rand r = new(seed);

        for (int i=0; i<bytes; i+=stride) begin
                tx_seq.data[0] = r.next();  // Generate next repeatable-random value for r
                tx_seq.start(m_agent.m_seqr); // tx_seq will transmit one byte for each call
                // Spin the counter ahead for nonzero strides
                for (int j=1; j<stride; j++) begin
                        void'(r.next());
                end
        end
endtask
```

Example 10: Task in Virtual Sequence to Control Data Generation

The **pvm_rand** type provides a repeatably-generatable random value based on the **seed** constructor argument, as shown in Example 11. Thus, from a given **seed** passed to its constructor, the **next()** method will return a repeatable pseudo-random value.

```
class pvm_rand;
        local int unsigned  m_seed;

        function new(int unsigned seed);
                m_seed = seed;
        endfunction

        function int unsigned next();
                m_seed ^= (m_seed << 13);
                m_seed ^= (m_seed >> 17);
                m_seed ^= (m_seed << 5);
                return m_seed;
        endfunction

        endclass
```

Example 11: SystemVerilog Implementation of Transmit Action

Example 12 shows how the **tx** task is defined as the implementation in PSS.

```
        extend action uart_agent_c::tx_a {
                exec body SV = """
                        tx({{tx_info.ref}}, {{tx_info.sz}}, 1);
                """;
        }
```

Example 12: SystemVerilog Implementation of Transmit Action

The "{{ }}" notation allows values to be passed from the PSS model (i.e. the **tx_info** field in the **uart_agent_c** component) into the SystemVerilog task to allow the data stream to be generated in the target UVM implementation as a sequence of **uart_serial_seq_item** types.

Similarly, we can provide a UVM-compatible implementation of the **rx2mem_a** task for the UART component as in Example 13.

```
        extend action wb_uart_c::rx2mem_a {
                exec body SV = """
                        rx2mem({{dat_o.addr}}, {{dat_o.sz}});
                """;
        }
```

Example 13: SystemVerilog Implementation of UART Receive Action

which uses the **addr** and **sz** fields of the **dat_o** object to store the data in the UVM register model to be called from a SystemVerilog task is in Example 14.

```
virtual task rx2mem(int unsigned addr, int unsigned sz);
        byte unsigned data;
        ...
        for (int i=0; i<sz; i++) begin
                ...
                m_mem_if.write8(data, addr+i);
        end
endtask
```

Example 14: SystemVerilog Task That Implements the Rx2mem Action

We complete the PSS model as shown in Example 15 by declaring a new component and an action to check the data:

```
component pvm_data_util_c {
        action checkdata_mask_a {
                input data_ref_mem_b        dat_i;
                rand bit[7:0]               mask;
        }
}
```

Example 15: PSS Checkdata Action

and provide an implementation that maps to the SystemVerilog environment as in Example 16.

```
extend action pvm_data_util_c::checkdata_mask_a {
        exec body SV = """
                checkdata_mask({{dat_i.ref}}, {{dat_i.addr}}, {{dat_i.sz}}, {{mask}});
        """;
}
```

Example 16: PSS Checkdata Implementation

To complete the test, we need to modify the scenario from Example 4 above as shown in Example 17 to include the check:

```
component wb_uart_ip_scenarios_c {
        action test_rx_data {
                wb_uart_c::rx2mem_a                     uart_rx;
                uart_agent_c::tx_a                      uart_agent_tx;
                pvm_data_util_c::checkdata_mask_a       checkdata;

                activity {
                        bind uart_rx.rx_info uart_agent_tx.tx_info;
                        bind uart_rx.dat_o checkdata.dat_i;

                        parallel {
                                uart_rx;
                                uart_agent_tx;
                        }
                        checkdata;
                }
        }
}
```

Example 17: Compound Action to Fully Test UART Receive action

When the PSS model is rendered as a SystemVerilog test, the **test_rx_data** scenario could be represented in the **wb_uart_pss_rxtx_transfer_seq** virtual sequence as shown in Example 18.

```
class wb_uart_pss_rxtx_transfer_seq extends wb_uart_pss_vseq_base;
        `uvm_object_utils(wb_uart_pss_rxtx_transfer_seq)
        task body();
        ...
        fork
                wb_uart_dev_rx2mem(0, 27566, 5);
                uart_agent_dev_tx(1, 3027603715, 5, 1);
        join
                pvm_uvm_pkg::checkdata_mask(3027603715,
                                                27566, 5, 127);
                ...
        endtask
endclass
```

Example 18: Virtual Sequence Realization of test_rx_data Scenario

Note that the **rx2mem** and **tx** tasks execute in parallel, as specified in the original scenario, followed by the **checkdata_mask** task. Note also that the **checkdata_mask** task uses the same **ref** field (argument 1) as the **tx** task (argument 2), which generated the data in the first place, and the same **addr** value (argument 2) as the **rx2mem** task (argument 2), and that all three tasks use the same **sz** field value.

The data flow between the actions in the PSS model and the implementations in SystemVerilog are shown in Figure 4:

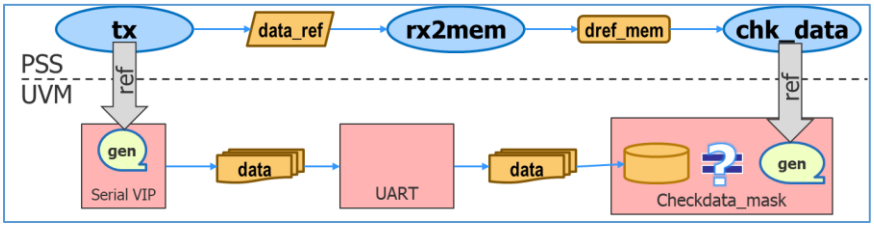

Figure 4: Data Flow in PSS and SystemVerilog

Since there is no formal reporting mechanism in PSS, any error reports must be provided by the underlying implementation. In this case, as seen in Example 19, we rely on a conditional **uvm_error** report from the **checkdata_mask** task:

```
task automatic checkdata_mask(
        bit[31:0]               seed,
        bit[31:0]               addr,
        bit[31:0]               sz,
        bit[7:0]                        mask);
        pvm_rand r = new(seed);

        for (int i=0; i<sz; i++) begin
                int unsigned v = r.next();// Generate same data stream from given seed
                byte unsigned d;

                pvm_ioread8(d, addr+i);
                if ((d & mask) != (v & mask)) begin
              `uvm_error("checkdata_mask", $sformatf("Address 'h%08h: expect %02h, \
                        receive %02h", (addr+i), (v&mask), (d&mask)));
                end
        end
endtask
```

Example 19: SystemVerilog Implementation of Checkdata Action

Similarly, the converse action, **test_tx_data**, is seen in Example 20.

```
component wb_uart_ip_scenarios_c {
        action test_tx_data {
                pvm_data_util_c::gendata_a   gendata;
                wb_uart_c::mem2tx_a          uart_tx;
                uart_agent_c::rx_a           uart_agent_rx;

                activity {
                        bind uart_tx.tx_info uart_agent_rx.rx_info;
                        bind uart_tx.dat_i gendata.dat_o;

                        gendata; // Generate data in memory
                        parallel {
                                uart_tx; // Transmit data from memory
                                uart_agent_rx; // Receive and check data
                        }
                }
        }
}
```

Example 20: Compound Action to Fully Test UART Transmit action

REUSING TEST INTENT

The value of such an abstract reusable data-generation-and-checking model in PSS will become clear as we move from block-level verification to the subsystem level, where the UART block will be instantiated along with several other blocks and their attendant VIP components, such as is shown in Figure 5.

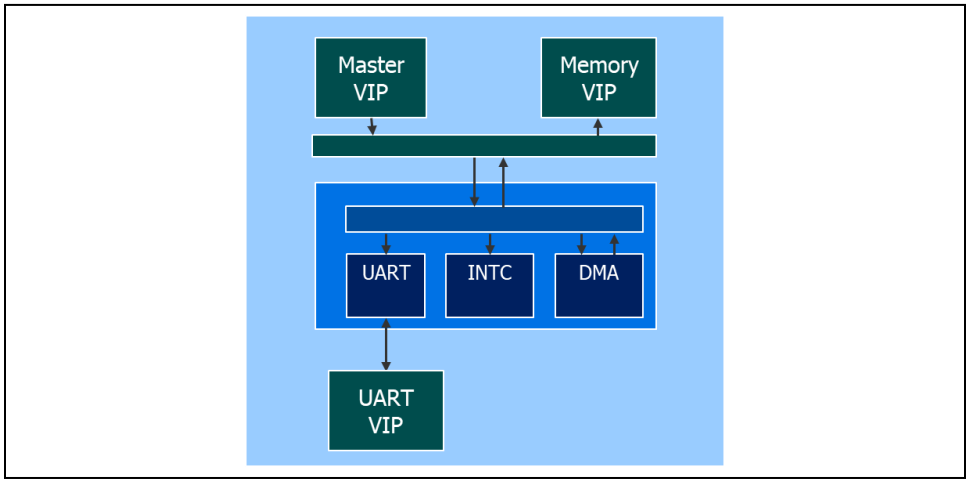

Figure 5: Subsystem Block Diagram

In this subsystem, the UART is combined with a DMA controller (**DMA**) and an Interrupt controller (**INTC**). The UART block is still connected on its external interface to the UART VIP, but the internal interface that was connected at the block level to the WB VIP is now connected to an actual bus that is itself connected to the other blocks in the design, as well as to new **Master VIP** and **Memory VIP** blocks. There are several things to consider about the new context in which we find the UART block.

The first is that we no longer have full access to all of the handshake/control signals to the UART, since they are now connected to other blocks in the design, such as the DMA. Also, since we have exhaustively verified the UART at the block level, we do not need to rerun the same block-level scenarios since it would not test anything new. However, we can certainly plan to reuse the RX and TX actions from the UART VIP, as well as the rx2mem and mem2tx actions for the UART VIP.

Similarly, we can expect that at this point in the project, the DMA and INTC blocks have similarly been fully-verified at the block level. This means that we have additional actions available from the DMA PSS component that we can use to build subsystem-level scenarios.

```
component wb_dma_c : pvm_dev_c {
        abstract action dma_dev_a : pvm_dev_a {
                // All transfers involve a channel
                rand bit[7:0] in [0..7]         channel;
                // Size of each transfer
                rand bit[4] in [1,2,4] trn_sz;
        }
        action mem2mem_a : dma_dev_a {
                input data_ref_mem_b        dat_i;
                output data_ref_mem_b       dat_o;
        }
        action dev2mem_a : dma_dev_a {
                output data_ref_mem_b       dat_o;
                input data_ref_s            info_i;
        }
        action mem2dev_a : dma_dev_a {
                input data_ref_mem_b        dat_i;
                output data_ref_s           info_o;
        }
}
```

Example 21: PSS DMA Component Actions

Note that in Example 21, since all three actions are ultimately derived from the **pvm_dev_a** base action type, they all contain a **devid** field. Graphically, these actions are shown in Figure 6.

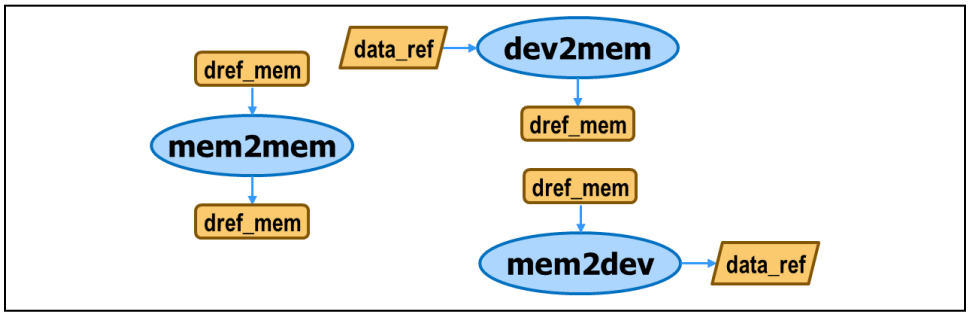

Figure 6: DMA Component Actions

In addition to the UART scenarios (**rx2mem** and **mem2tx**) from the block level, we can also specify additional scenarios for our subsystem that use the DMA actions to send data to or from the UART (**mem2dev** and **dev2mem**, respectively), and we can also use the DMA itself to create additional traffic on the bus using the **mem2mem** action, assuming the DMA block provides enough channels to operate multiple DMA transfers concurrently. We can extend the scenario space by adding additional details like using specific memory address regions or specific DMA channels for different operations via constraints applied to the actions. Note that the underlying UVM implementations for the UART and DMA blocks do not have to change from their block-level PSS specifications, so we can reuse those details as well.

In the abstract model, we continue to use the same basic data flow objects, relying on the **ref** field to allow data to be generated and checked by the underlying implementation. The UART VIP will continue to generate data in the **tx** action and check it in the **rx** action. The explicit **gendata** and **checkdata** actions can still be used at the subsystem scenario level since their implementations are memory-based. For simplicity in this paper, we will limit our scenarios to "one hop" transfers, which consist of one of the following

- Generate data buffer in memory, DMA transfer to another buffer, check data (**mem2mem**)
- Generate data buffer in memory, DMA transfer to UART, UART VIP receive/check data (**mem2dev**)
- Generate data buffer in memory, UART retrieve/transmit data, UART VIP receive/check data (**mem2tx**)
- UART VIP generate/transmit data, DMA transfer from UART to memory, check data (**dev2mem**)
- UART VIP generate/transmit data, UART receive and store data, check data (**rx2mem**)

We expand our choices in the PSS test scenario as shown in Example 22, using an expanded set of choices defined in the **scenario_e** enumeration, which gets randomized at the start of the **activity** statement, and the **m2m_sz** parameter which gets randomized, subject to constraints, before the operation is chosen.

```
component wb_periph_subsys_scenarios_c  {
        enum scenario_e {
                scenario_m2m,
                scenario_tx_pio,   scenario_rx_pio,
                scenario_tx_dma, scenario_rx_dma
        };
        rand scenario_e     scen;
        action bit[16]                  m2m_sz;
        constraint {
                if (scen == scenario_m2m) {
                        m2m_sz == mem2mem.dat_i.sz;
                } else {
                        m2m_sz == 0;
                }
        }
        activity {
                scen;                   // randomize the scenario
                m2m_sz;  // randomize the mem-to-mem transfer size
                select {
                        (scen == scenario_m2m): sequence {
                                gendata;
                                mem2mem;
                                checkdata;
                        }
                        (scen == scenario_tx_pio): sequence {
                                gendata;
                                parallel {
                                        uart_mem2tx;
                                        uart_agent_rx;
                                }
                        }
                        (scen == scenario_rx_pio): sequence {
                                parallel {
                                        uart_agent_tx;
                                        uart_rx2mem;
                                }
                                checkdata;
                        }
                        (scen == scenario_rx_dma): sequence {
                                parallel {
                                        uart_agent_tx;
                                        dev2mem;
                                }
                                checkdata;
                        }
                        (scen == scenario_tx_dma): sequence {
                                gendata;
                                parallel {
                                        uart_agent_rx;
                                        mem2dev;
                                }
                        }
                }
        }
}
```

Example 22: PSS Subsystem Scenarios

These choices are shown graphically in Figure 7.

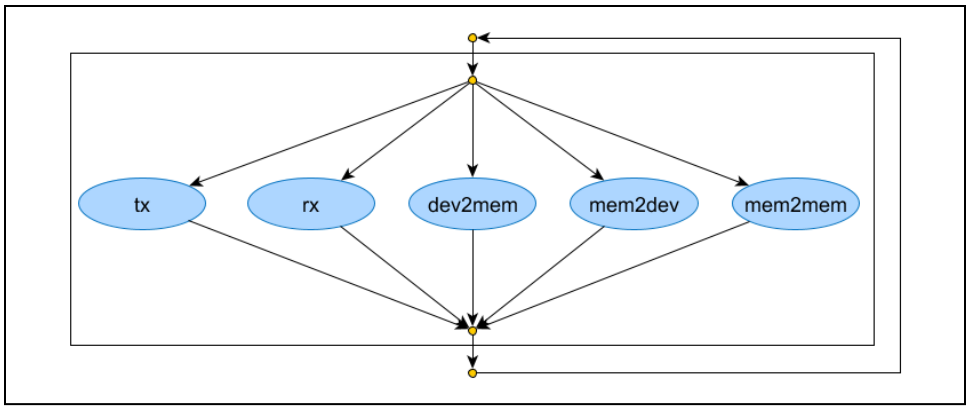

Figure 7: DMA Component Actions

SUBSYSTEM-LEVEL MODELING

At the subsystem level, there may be several details that are different from the block level environment(s). For example, the memory addresses may be different, the blocks may use different device IDs, and there may be different constraints on certain operations, such as requiring a specific set of DMA channels to communicate with the UART, or only allowing the UART to access certain address ranges. These can be modeled as additional layered constraints in the PSS model.

We could create more elaborate scenarios, of course, such as a loopback test where the data is generated by the UART VIP, gets received by the UART and stored in memory either by the UART or via DMA, then the same data is retrieved from memory either by the UART or via DMA, sent to the UART VIP and checked against the original data. We could also put a random number of memory-to-memory transfers of the data between the receive and the transmit actions. Each additional scenario choice adds to the possible solution space of the resulting implementation, but all will still carry the burden of having to check that the data that winds up at the terminus of the given operation is correct. In this particular case, since the data always winds up either in memory or in the UART VIP, we can reuse the same generation and checking implementations from the UART block-level environment seen earlier.

Of course, there is more to "correctness" at the subsystem level than just ensuring that the correct data was able to flow through the system. Often subsystem tests are concerned with measuring throughput and/or latency as well as bus utilization. Because these aspects are platform-specific, they will similarly rely on the underlying verification environment to perform the measurements and report the results. To be sure, the scenarios can be modeled to maximize bus traffic, for example, by performing N DMA transactions in parallel with the above scenarios. However, it would still fall on the underlying implementation environment to measure and report on how these complex scenarios actually impact the actual performance of the system.

MOVING TO THE SOC LEVEL

Having verified that the subsystem can correctly transfer data between the UART and the memory, we take advantage of much of the PSS infrastructure when we move up to the SoC level.

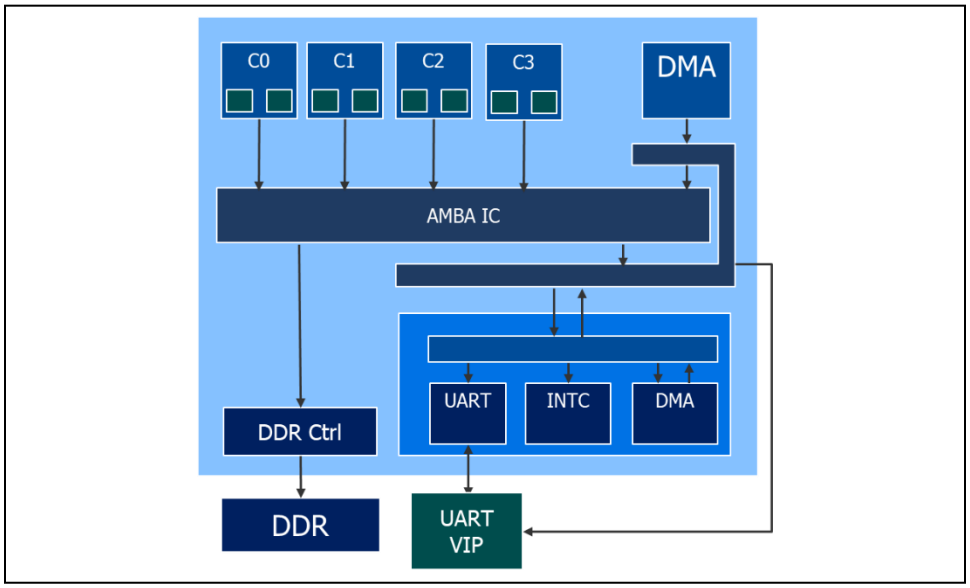

Figure 8: SoC Block Diagram

At this level (Figure 8), the Memory VIP from the subsystem model (Figure 5) is replaced by an actual DDR memory, and the Master VIP is replaced by a (set of) CPU model(s), which will actually execute C code. In a simulation or emulation environment, these may be models, while in an actual FPGA or post-silicon SoC, these may be actual processors and memories.

Also, in this particular example, we have another DMA controller in the system that is available to transfer data between devices. This is where the **devid** field comes into play, since each DMA action will need to specify which DMA instance is to be used to perform the action, as shown in Example 23.

```
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}});
        """;
}
```

Example 23: DMA Action Implementation in C

At this point, we must understand that, although our PSS model is still abstract and may consist of the same set of actions available at the subsystem level, nevertheless, the target implementation must rely on the processors to drive traffic, which requires the use of C code to implement the actions as in Example 24.

```
extend action uart_agent_c::tx_a {
        exec body C = """
                uart_agent_dev_tx({{devid}}, {{tx_info.ref}}, {{tx_info.sz}}, 1);
        """;
}
extend action wb_uart_c::rx2mem_a {
        exec body C = """
                wb_uart_dev_rx2mem({{devid}}, (void *){{dat_o.addr}}, {{dat_o.sz}});
        """;
}
extend action pvm_data_util_c::checkdata_mask_a {
        exec body C = """
                pvm_checkdata_mask({{dat_i.ref}}, {{dat_i.addr}}, {{dat_i.sz}}, {{mask}});
        """;
        }
```

Example 24: C code Implementations to Support the test_rx_data Scenario

It becomes a file management problem to pass the new extensions, such as in Example 24, to the PSS processing tool along with the same PSS files for the abstract model to define the actual test intent. When implementing the actions, there are a few things to keep in mind.

The first is that, although we are still using the UART VIP as the external device, we no longer have the UVM test environment to control, nor do we have access to the virtual sequence. Instead, we rely on the PSS tool to generate the same set of scenarios, but as C code shown in Example 25 that runs on the processors (with appropriate threading and inter-process communication) according to the schedule specified by the PSS model.

```
void wb_periph_subsys_test(void) {
  ...
            {
            infactpss_thread_t branch_1 = infactpss_thread_create(&thread_func_5);
            infactpss_thread_t branch_2 = infactpss_thread_create(&thread_func_6);
            infactpss_thread_join(branch_1);
            infactpss_thread_join(branch_2);
            }

            pvm_checkdata_mask(2362696682, 1612443855, 14, 255);
  ...
}

void thread_func_5(void) {
            uart_agent_dev_tx(0, 2362696682, 14, 1);
}
void thread_func_6(void) {
            wb_uart_dev_rx2mem(2, (void *)1612443855, 14);
}
```

Example 25: C code Realization of test_rx_data Scenario

Compare the C code in Example 25 to the UVM virtual sequence in Example 17. The **infactpss_thread*** calls in Example 25 are part of a threading package developed for our target environment and passed to the PSS tool. The **infactpss_thread_create** methods cause **uart_agent_dev_tx** (via **thread_func_5**) and **wb_uart_dev_rx2mem** (via **thread_func_6**) to be called in parallel, just as in the virtual sequence, followed by **pvm_checkdata_mask**. As you would expect, the **uarg_agent_dev_tx** and **pvm_checkdata_mask** methods use the same **ref** value, and the **wb_uart_dev_rx2mem** and **pvm_checkdata_mask** methods refer to the same address, while all three use the same **sz** value. The **uart_agent_dev_tx** method, and all methods that control the behavior of the UART VIP, rely on a "trickbox" implementation where writes to specific memory addresses cause the environment to invoke the specific tasks in the VIP SystemVerilog implementation to affect the same behavior as if calling them in UVM from a virtual sequence. The trickbox itself could easily be the subject of another paper.

It is easy enough to provide C code implementations of the **gendatat** and **checkdata** methods as in Example 26.

```
void pvm_checkdata_mask(uint32_t ref, uintptr_t addr,
                        uint32_t sz, uint8_t mask) {
            pvm_rand_t r;
            void *addr_p = (void *)addr;
            int i;

            pvm_rand_init(&r, ref);

            for (i=0; i<sz; i++) {
                        uint8_t exp_v = pvm_rand_next(&r);
                        uint8_t v = uex_ioread8(addr_p+i);
            }
}
```

Example 26: C code Implementation for Checkdata Method

As mentioned above, the key to the whole methodology laid out here is for the **pvm_rand** data object to be able to provide a repeatably-pseudorandom value from a seed (which will be left as an exercise to the reader).

CONCLUSION

The new Portable Test and Stimulus Standard from Accellera defines a declarative language for specifying *verification intent* at an abstract level. A critical aspect of "portable" is the ability to provide implementations of this

verification intent on multiple platforms and environments. Since results checking is so often dependent on the underlying implementation, and often entails platform-specific metrics that go beyond the scope of the abstract PSS intent model, we must approach the problem with foresight.

In an SoC environment, all data is managed by the C code running on the processor(s), even the data that eventually gets generated by external VIP, through a software trickbox or perhaps an emulator-specific mechanism. Because the processor(s) can only manage data via memory accesses, a **gendata**/**checkdata** scheme must be used to guarantee portability throughout the flow. In fact, this is the "end" we must have in mind when beginning at the block level to create a PSS model that will indeed be portable. Augmenting this result-centric approach to checking with detailed checkers that are environment-specific ensures that verification driven by portable-stimulus tests gets the consistency of environment-independent result checking and can benefit from detailed checks that are environment specific.

---

[1]. Accellera Systems Initiative. *Portable Test and Stimulus Standard Version 1.0*. June 2018
[2]. Covey, Stephen R. *The 7 Habits of Highly Effective People: Restoring the Character Ethic*. [Rev. ed.]. New York: Free Press, 2004.