PSS and Protocol VIP: Like a Hand in a Glove

Tom Fitzpatrick, Siemens EDA, Groton, MA, USA (tom.fitzpatrick@siemens.com)
Bob Oden, Siemens EDA, Raleigh, NC, USA (bob.oden@siemens.com)
Ahmed Abd-Allah, Siemens EDA, Cairo, Egypt (ahmed.abd-allah@siemens.com)
Billy Graham, Siemens EDA, Bangalore, India (prodduturi-vasantha.billy-graham@siemens.com)

Abstract— The Portable Test and Stimulus Standard (PSS) encourages verification engineers to focus on describing test scenarios, without worrying about the underlying target environment on which the test will ultimately be run. By describing the scenarios in terms of a randomizable schedule of actions, or behaviors that will execute, the test can easily be retargeted to different implementations for different environments. Especially at the block- and subsystem-level, these target environments will usually be UVM environments, often incorporating standard-protocol VIP components.

The ability to reuse a PSS test scenario across multiple environments, including using different VIP to execute the same actions, is one of the main definitions of the "Portable" part of PSS. This paper will illustrate a PSS methodology to define an abstract model that can be retargeted at different VIP components to allow the same scenario throughout your development flow.

I. Introduction

The Portable Test and Stimulus Standard (PSS) from Accellera[1] defines an abstract modeling language to specify critical verification intent and allow the generation of target-specific implementations of a set of correct-by-construction *scenarios*, each of which includes the critical intent supplemented by additional behaviors to meet the requirements of the critical intent. The declarative nature of a PSS description allows a concise model to be the basis for a potentially large number of scenarios.

By definition, a PSS test defines a set of actions that represent the verification behaviors required to exercise desired functionality of the DUT. These actions themselves can be defined at multiple levels of abstraction, from basic bus read/write operations to higher-level actions, such as DMA transfers, message passing, or other behaviors.

A PSS test is composed of the following:

1. A set of *actions* that define the set of behaviors to be executed.
2. An *activity* that defines the critical actions and their relative scheduling constraints
3. Data flow requirements between actions
4. Additional data and scheduling constraints
5. The target-specific implementation(s) of each action

From these elements, a PSS tool can generate a target-specific implementation of the overall test scenario. To the extent that the schedule and constraints are flexible enough to permit multiple scenarios to satisfy the specified verification intent, a PSS tool can effectively create a constrained-random implementation of the desired scenario. This scenario-level randomization makes it easier to create complex test scenarios from easy-to-describe tests. This approach also provides the secondary benefit of allowing the block-level verification team to create a library of actions that can be reused at the subsystem and system level.

When targeting a UVM environment, the PSS test scenario can be realized as a UVM virtual sequence that implements the schedule of actions as defined in the activity in PSS, with each action being mapped to a target-specific implementation of the desired functionality. As multiple blocks are assembled into a subsystem, a PSS model may be created that schedules the actions defined for each sub-block into an activity that defines the subsystem-level test. If this test is also targeted at a UVM environment, the resulting realization will be a subsystem-level UVM virtual sequence that will still ultimately invoke the implementation of each action in the target environment, but it is much easier to coordinate these behaviors across blocks and interfaces from the PSS model than to write UVM sequences manually to try and create the desired scenarios.

II. Test Realization

To define exactly how a particular PSS action is implemented in the target system, PSS includes a *procedural interface* that allows UVM tasks to be imported into the PSS model for the given action. The task will be called from

the generated UVM virtual sequence and will ultimately result in a UVM transaction-level sequence being executed on the appropriate agent/sequencer to implement the functionality.

When the DUT includes an interface that utilizes a standard protocol, such as AXI, the appropriate VIP component will be included in the UVM test environment. Each VIP component will often include an application-programming interface (API) that will abstract the execution of the transaction-level sequence to simplify the use model. The PSS package that defines the imported functions is accompanied by a corresponding VIP-specific SystemVerilog package that supplies the implementation of the imported function for the target environment. When generating code for a SystemVerilog implementation, imported target functions are realized as tasks.

The separation in PSS between the abstract model and the realization layer is the key to portability in PSS. Since the abstract scenario model focuses on the behaviors to be executed by the DUT, it is largely agnostic to the particular protocol that may be used by the realization layer to implement the behaviors. Consider, for example, a simple DMA transfer action. The abstract model would specify fields such as the source and destination addresses and the size of the transfer, as well as the particular DMA channel to be used for the transfer. The implementation of the action would require the DUT to be programmed by writing the source and destination addresses and the transfer size into particular registers and then the channel-specific "start" bit would be written. The realization layer would call the design-specific method to execute the necessary register operations to perform the desired operation. The judicious use of packages, in both PSS and SystemVerilog, will allow the implementation of the register operations to be handled by the desired VIP by delegating the abstract read/write operations from PSS to the specific VIP API. If a different variation of the DUT used a different protocol, a different realization layer that provides the corresponding API calls for the new protocol could be used without requiring any changes in the original abstract model, resulting in the exact same scenario being executed in the new environment.

III. Illustrative Example

Introducing the Design Example

Consider the simple design shown in Figure 1. The memory element includes data storage that can be directly written to or read from, and a set of channel-specific registers to support DMA transfers of data chunks from one address to another, as well as transferring data to or from the peripheral port. The peripheral element contains a FIFO data buffer and a set of control registers to support transfering data to or from the memory port. Both RTL elements support the same simple read/write protocol for communication on their control ports, and the data bus between them follows a simple handshake protocol. The details of these interfaces are not important for the purposes of this paper.

The memory element includes four registers for each DMA channel:

- SRC: The base address of the data buffer to be transferred
- DST: The base address to which the data buffer is to be transferred
- SIZE: The size of the buffer to be transferred
- GO/DONE: The control register where the bit set determines if the transfer is memory-to-memory, memory-to-peripheral, or peripheral-to-memory. Setting the bit starts the desired transfer and the bit clears when the transfer is complete.

Similary, the peripheral element supports writing data to and reading data from the internal FIFO, and it also includes two control registers:

- SIZE: The size of the buffer to be transferred to/from the memory
- GO/DONE: The control register to specify either memory-to-peripheral or peripheral-to-memory transfers. The set/clear behavior is the same as for the memory

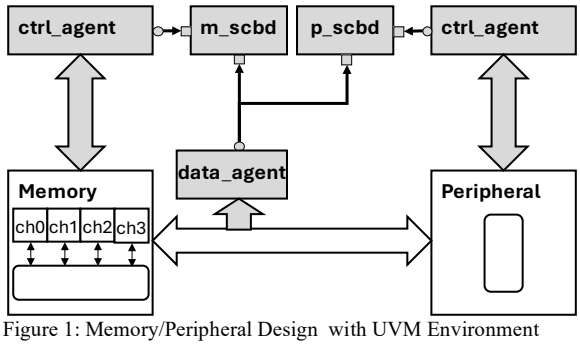

Figure 1: Memory/Peripheral Design with UVM Environment

The UVM Environment

In the UVM environment, each `ctrl_agent` supports transaction-level sequences to transfer data to/from the connected design element. The passive `data_agent` monitor tracks the data transferred between the two devices while the two scoreboard components serve as reference models to ensure that the data in the memory and peripheral devices are correct. A typical UVM test would consist of a UVM virtual sequence that coordinates the execution of agent-level transaction sequences to implement the desired scenario.

Suppose we want to test that we can read data out of the memory. To test that, we must first write data into the memory. We can either do a "memory fill" operation to write the data into the memory from the `cb_agent`, or we can load the data into the peripheral device and transfer it from the peripheral to the memory, or we could fill a memory buffer, do a DMA transfer to a different location in the memory, and dump it from there, or any combination of these. A basic set of such sequences is shown in Figure 2. Note that the m2p and p2m operations require both the memory and peripheral components to execute concurrently because there is no intermediate storage between them. The data produced by one component must be consumed concurrently by the other.

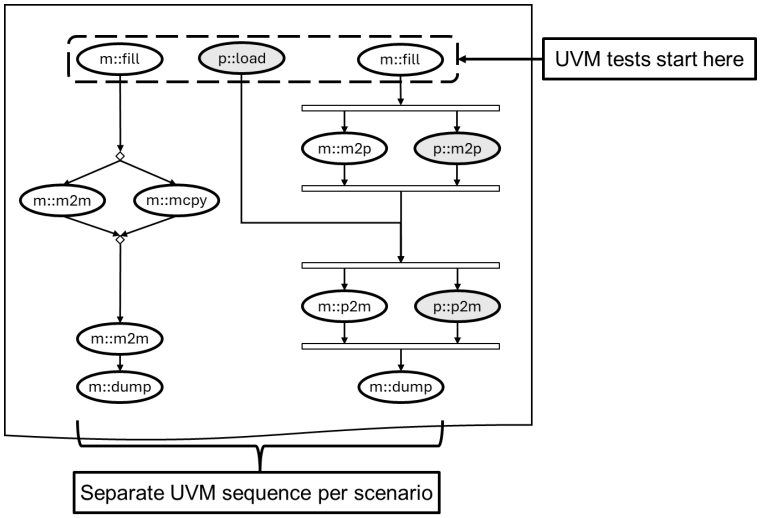

Figure 2: UVM Sequences to Model Scenarios

The UVM test writer would need to start at the top of this graph to consider the different ways to get data into the memory to dump the data and check that it is correct. This can start by filling the memory or by loading the peripheral. If we wish to keep the data local to the memory (such as in a block-level test that involves only the memory component), we could follow the initial fill operation with either a memory-to-memory DMA transfer (m2m) or we could do a copy operation (mcpy) where each memory word is read from the source address and written to the destination address. We could then do a separate m2m operation to get the data to the address from which we want to do the dump operation. Alternately, after the initial memory fill operation, we could choose to transfer the data over to the peripheral (m2p), but then we need to transfer the data back to the memory (p2m) before we can dump the data from the memory. The third scenario shown is to load the data originally into the peripheral, followed by the p2m operation prior to the dump.

In addition to the UVM test writer having to conceive of these scenarios, each one requires the test writer to keep track of the individual transactions to make sure that the ouput of one transaction is the input to the next. Even for this relatively simple example, there could be a lot of details to keep track of to ensure that each scenario is correct.

The other thing the astute reader will notice is that each scenario described above is essentially a "directed-random" test. The elements of each transfer, such as source, destination and size, could be randomized, but the overall scenario is the same so that, even if the same test were run with different random seeds, the set of scenarios would be the same albeit with different data characteristics of each transfer. In this case, there may also be some scenarios that use the m2m operation while others use the mcpy operation, but each of those operations has the same type of inputs and

outputs so it's not difficult to add an `if` statement in the scenario. It is much more difficult to randomize the entire scenario in UVM.

The test writer could start with an `if` statement to begin the scenario with a memory fill or a peripheral load. It rapidly becomes untenable to specify the follow-on transactions based on that choice, especially if there are more than just two choices at any particular node. Trying to write a single UVM virtual sequence with the required nesting of `if` or `case` statements is extraordinarily difficult.

Modeling the Tests in PSS

PSS makes this task much easier. A PSS model begins by defining the set of actions that must be supported, and the data flow requirements for them. For the memory operations, the data communication will be modeled using a PSS `buffer` data flow object, which is similar to a struct type in SystemVerilog. We then define the memory fill and dump actions as outputting or inputting a buffer object of that type, respectively, as shown in Example 1. Similarly, we can also define the m2m and mcpy actions that each input and output the corresponding buffer type.

```
struct cb_struct {
    rand bit [31:0] addr;
    rand bit [31:0] size; // number of bytes
}

buffer cb_mbuf : cb_struct {
    rand MemKindE kind; // RAM Type (DRAM or SRAM)
    rand bit[32]  alignment; // Data alignment in the associated memory location
    constraint alignment >= 64;
}

action mem_fill {
    output cb_mbuf obuf;
    constraint c1 {obuf.size in [16..64];}
}

action mem_dump {
    input cb_mbuf ibuf;
}

action mem_xfer {
    input cb_mbuf ibuf;
    output cb_mbuf obuf;
    rand int in [0..4] chan;

    constraint c1 {obuf.size == ibuf.size;}
}

action mem_m2m : mem_xfer {
}

action mem_mcpy : mem_xfer {
}
```

Example 1: Simple Memory Actions and Buffer Type

In the example, we can see that we start with the `struct` element `cb_struct`, which defines the `addr` and `size` fields, which will be common to other types that will be derived from `cb_struct`, such as `cb_mbuf`, which adds the `kind` and `alignment` fields. The `mem_m2m` and `mem_mcpy` actions are both derived from the `mem_xfer` base action type, and each inputs and outputs a `cb_mbuf` object, with the added constraint that the output and input buffers are constrained to have the same value for `size`. At the abstract model level, these two actions are identical, but we create two separate action types so that we can later add different realization layer implementations for each.

The PSS language was designed specifically to define stimulus scenarios, so the semantics include a number of features that make it easier to create multiple scenarios from a single model. Perhaps the most useful of these features is the concept of *action inferencing*. The semantic rules define that if an action inputs a buffer type, there **must** be corresponding action that outputs a compatible buffer type. Further, if the model includes both a producer and a consumer of the same type, then the output of the producer is automatically bound to the input of the consumer. If we consider the scenarios illustrated in Figure 2, the implications for this are staggering.

To create a test in PSS, we can "begin with the end in mind"[2] by defining our PSS model simply to traverse the `mem_dump` action, as in Example 2.

```
action pss_top {
    activity {
        repeat (3) {
           do mem_dump;
        }
    }
}
```

Example 2: Simple PSS Model to Create Multiple Scenarios

In this model, for each traversal of the `mem_dump` action, the PSS tool will need to infer an action that can provide the input `cb_mbuf` object. If we consider just the memory operations, if the `mem_fill` action is inferred, then we're done because that action doesn't require any additional input. However, if the `mem_m2m` or `mem_mcpy` actions are inferred, then we'll have to infer another action to supply its input buffer type, for which we are presented with the choice of inferring one of the same three action types. Eventually, each scenario must start with `mem_fill` since that is the only action that produces the correct buffer type without requiring an input. The user can limit the number of inferences in the chain and can choose to add additional fields and constraints to the model to ensure that a minimum number of operations is inferred.

At the subsystem level, where we test the combination of the memory and peripheral components, we can add a few more details to the model to, for example, avoid the simple fill/dump scenario, as shown in Example 3.

```
action pss_top {
    activity {
        repeat (4) {
             select {
                do mem_p2m with {obuf == mdump.ibuf;};
                do mem_m2m with {obuf == mdump.ibuf;};
            }
            do mem_dump;
        }
    }
}
```

Example 3: PSS Model to Add Complexity

Although not shown, a similar set of actions is defined for the peripheral, including the `m2p` and `p2m` actions, which share a data flow object type with the (not shown) `m2p` and `p2m` in the memory. If the solver chooses a `p2m` action to supply the buffer input to `mem_dump`, then the complementary `p2m` action must also be inferred to be traversed concurrently by the peripheral, and then that action will require the inferencing of either `load` or `m2p` to supply its

input as shown in Figure 2. The same model can obviously produce many more scenarios than just those shown in Figure 2.

Thus, with a very simple test specification, a PSS tool can automatically create a large number of scenarios, and manage the data connections and scheduling constraints between them. Notice that we have not yet considered how any of these actions will be implemented in the target environment. PSS was developed specifically with this distinction in mind, to allow this level of abstraction in the model itself. To specify the implementation of a scenario, we use the PSS Realization Layer

Realizing PSS Tests

The implementation of a PSS action is defined via one or more `exec` blocks in the PSS model. For our basic UVM example, we declare a package that extends the action declarations to define the design-specific API, as shown in Example 4.

```
package cb_pss_mem_uvm_pkg {
    extend component mem_c {
        extend action mem_fill {
            exec body {
                do_fill(comp.id, obuf.addr, obuf.size);
            }
        }
...
    }
}
```

Example 4: PSS Package to Provide Realization

By importing this package into the PSS model, the `exec body` block provides the implementation of the `mem_fill` action as a call to the `do_fill` task from the resulting UVM virtual sequence. An example implementation of this method would be a SystemVerilog task that can be called from the UVM virtual sequence, as shown in Example 5.

```
virtual task automatic do_fill(logic[ADDR_WIDTH-1:0] dest, int size);
    ...
endtask
```

Example 5: PSS Package to Provide Realization

The `dest` argument specifies the target address of where the data will be written, and `size` specifies how many words should be written.

Note that we could use an alternate package to provide a different realization of the `mem_fill` action, as shown in Example 6

```
package other_pss_mem_uvm_pkg {
    extend component mem_c {
        extend action mem_fill {
            exec body {
                fill_mem(comp.id, obuf.addr, obuf.size);
            }
        }
...
    }
}
```

Example 6: PSS Package to Provide Realization

By using the `other_pss_mem_uvm_pkg` package, we provide a different `exec body` definition for the action. This has no impact on the abstract model in Example 3, but allows a different method to be called to realize the action.

The key to reuse is to separate the *design-specific* functionality from the *protocol-specific* functionality. In a general sense, the implementation of the `do_fill` method will use a loop of `write` method calls to load the memory

```
class qps_usr_mem_api extends uvm_sequence;
  ...
  pss_if_api_base ctrl_if_api;
  virtual task automatic do_fill(input int fid, bit[31:0] dest, size);
    bit[31:0] data;

    for(int i=0; i < size*4; i=i+4) begin
      bit stat = std::randomize(data);
      ctrl_if_api.write32(dest+i, data);
    end
  endtask
```

Example 7: UVM Implementation of `do_fill` method

The `qps_usr_mem_api` sequence defines the *protocol-independent* implementation of the *design-specific* `do_fill` method in terms of the `pss_if_api_base` sequence that defines the register-level API supported by PSS.

```
package pss_if_resources_pkg;
  import uvm_pkg::*;

  class pss_if_api_base;
    virtual task write32(input longint unsigned address, bit [31:0] data);
    endtask
    ...
    virtual task read32(input longint unsigned address,
                        output bit [31:0] data);
    endtask
    ...
  endclass
endpackage
```

Example 8: UVM Implementation of `pss_if_api_base` sequence

We next need to provide the *protocol-specific* implementation of these methods. We do this by declaring another sequence that is derived from the `pss_if_api_base` sequence that implements the register-level API in terms of the specific agent we are using to communicate with the memory component.

```
class uvmf_ctrl_bus_pss_if_api extends pss_if_api_base;
    ctrl_bus_api_sequence  seq;
    virtual task write32(input longint unsigned address, bit [31:0] data);
      seq.write32(address, data);
    endtask
    virtual task read32(input longint unsigned address,
                        output bit [31:0] data);
      seq.read32(address, data);
    endtask
```

```
    ...
endclass
```

Example 9: UVM Implementation of `pss_if_api_base` sequence

The `ctrl_bus_api_sequence` sequence is part of the ctrl_bus interface package. It uses the standard UVM mechanism to execute the desired transaction on the agent-level sequencer:

```
class ctrl_bus_api_sequence extends ctrl_bus_sequence_base ;
  `uvm_object_utils( ctrl_bus_api_sequence )

  task write32(input bit[31:0] a, d, bit[3:0]strb=4'b1111);
    req.memtrans = writew;
    req.addr = a;
    req.data = d;
    req.strb = strb;
    this.start(m_sequencer);
  endtask
  ...
endclass
```

Example 10: UVM Implementation of `pss_if_api_base` sequence

With this infrastructure in place, the user simply declares a top-level virtual sequence that implements the PSS actions as calls to the imported methods used in the `exec` blocks for the relevant actions. As shown in Example 11, this requires assigning the *protocol-specific* `ctrl_if_api` sequence in the top-level sequence to the corresponding API sequence in the *design-specific* `qps_usr_mem_api` sequence prior to running the scenario in the `body()` task.

```
class qps_usr_mem_periph_mapping_sequence extends qps_gen_sequence;
  `uvm_object_utils( qps_usr_mem_periph_mapping_sequence )

  uvmf_ctrl_bus_pss_if_api mem_ctrl_if_api;
  uvmf_ctrl_bus_pss_if_api periph_ctrl_if_api;

  qps_usr_mem_periph_api qps_usr_mem_periph_api;

  virtual task do_fill(bit[31:0] dest, size);
    qps_usr_mem_periph_api.do_cb_fill(dest,size);
   endtask

...
  task body();
    qps_usr_cbmem_periph_api.map_interface_sequences(
                                      .mem_ctrl_if_api(mem_ctrl_if_api),
                                      .periph_ctrl_if_api(periph_ctrl_if_api));

    super.body();
  endtask
endclass
```

Example 11: Mapping the Protocol-specific API Sequence to the Design-specific API methods

The recommended structure is to create a `qps_usr_<env>_api` sequence to represent the environment-specific interfaces required. As an example, the `qps_usr_mem_periph_api` sequence in Example 11 may represent a subsystem-level environment that is composed from separate block-level environments for each of the memory and peripheral components. Each of these block-level environments would have its own `ctrl_if_api` sequence to drive the relevant `ctrl_agent` instance.

```
class qps_usr_mem_periph_api extends uvm_sequence;
  ...
  pss_if_api_base mem_ctrl_if_api;
  pss_if_api_base periph_ctrl_if_api;

  qps_usr_mem_api    qps_usr_mem_api;
  qps_usr_periph_api qps_usr_periph_api;

  function map_interface_sequences( input pss_if_api_base mem_ctrl_if_api,
                                    input pss_if_api_base periph_ctrl_if_api);
    this.mem_ctrl_if_api = mem_ctrl_if_api;
    this.periph_ctrl_if_api = periph_ctrl_if_api;
    qps_usr_mem_api.map_interface_sequences(.ctrl_if_api(mem_ctrl_if_api));
    qps_usr_periph_api.map_interface_sequences(.ctrl_if_api(periph_ctrl_if_api));
  endfunction
  ...
endclass

class qps_usr_mem_api extends uvm_sequence;
  ...
  pss_if_api_base ctrl_if_api;
  pss_if_api_base data_if_api;

  function void map_interface_sequences( input pss_if_api_base ctrl_if_api,
                                         input pss_if_api_base data_if_api=null);
    this.ctrl_if_api = ctrl_if_api;
    this.data_if_api = data_if_api;
  endfunction
  ...
endclass
```

Example 11: Mapping the Protocol-specific API Sequence to the Design-specific API methods

The Questa Portable Stimulus tool generates the `qps_gen_sequence` that executes the schedule of the solved PSS `activity` and calls the appropriate methods as defined in the `qps_usr_mem_periph_mapping_seq` sequence according to that schedule. The resulting sequence hierarchy is shown in Figure 3.

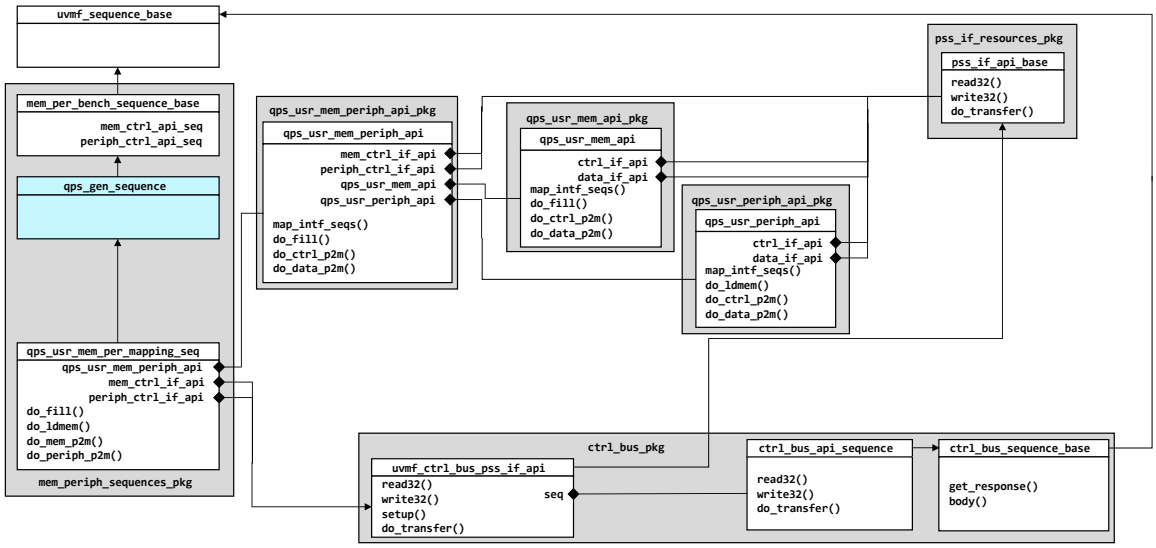

Figure 3: UVM Sequence Hierarchy

The generated `qps_gen_sequence` extends from a base virtual sequence that includes pointers to any sequencers or other test bench infrastructure required to implement the scenario. Figure 3 shows additional detail to show how the above scheme can be deployed to support multiple instances of the `ctrl_agent` as shown in Figure 1.

IV. USING STANDARD PROTOCOL VIP

In any SoC (System-on-Chip) verification flow, integrating standard protocol VIPs such as AXI, APB, or PCIe is crucial for ensuring compliance and reliability in data communication. In this section, we explore how to incorporate AXI VIP into a UVM environment using PSS and how to update the realization layer to map the abstract PSS actions to the AXI VIP API.

Updating UVM environment to add AXI VIP

In the verification environment, the AXI Responder is an RTL design that is connected directly to the AXI Manager VIP instantiated in the top_hdl module. The UVM environment leverages the AXI Manager VIP to generate transactions (reads and writes) to interact with the AXI Responder RTL.

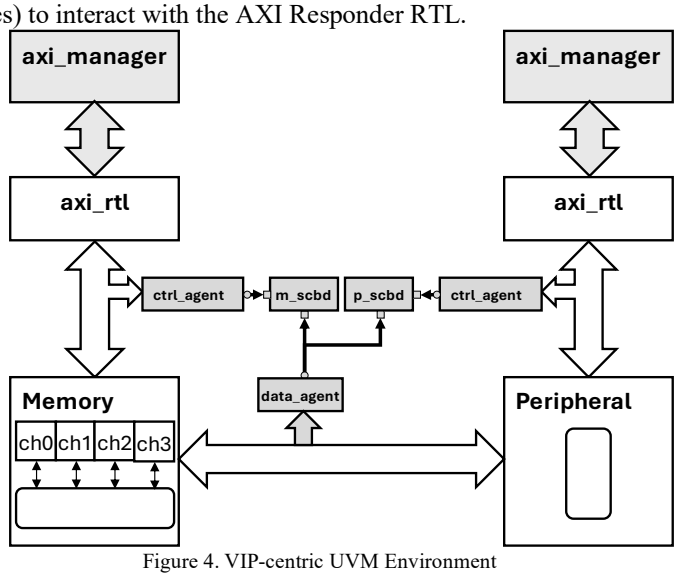

Figure 4. VIP-centric UVM Environment

The AXI Manager VIP is configured to generate the necessary AXI transactions, including configuration parameters such as address width, data width, and burst capabilities. Notice that the `ctrl_agent` components are now passive UVM components, serving as monitors of the traffic between the `memory` or `peripheral` blocks and

the connected `axi_rtl` block, allowing the same scoreboards to be used for error checking. The testbench focuses on configuring and driving transactions through the Manager VIP, ensuring a streamlined connection to the DUT.

```
mem_if_i #(
    .ADDR_WIDTH (ADDR_WIDTH),
    .DATA_WIDTH (WDATA_WIDTH)
    ) m_if[2] (
    .clk (CLK),
    .rst (ARESETn)
);

mem_periph_if_i #(
    .DATA_WIDTH(WDATA_WIDTH),
    .DMA_CHANNELS(DMA_CHANNELS)
    ) mp_if(CLK);

genvar i;
generate for(i=0;i<2;i=i+1) begin: gscope
    vip_axi4_hdl_manager #(
        .ADDR_WIDTH        (ADDR_WIDTH             ),
        .ID_R_WIDTH     (ID_R_WIDTH          ),
        .ID_W_WIDTH    (ID_W_WIDTH         ),
        .RDATA_WIDTH   (RDATA_WIDTH        ),
        ...
        ) manager_bfm (
            .ACLK      (CLK        ),
            .ARESETn   (ARESETn    ),
            .AWID      (AWID       [i]),
            .AWADDR    (AWADDR     [i]),
            .AWLEN     (AWLEN      [i]),
        ...
    );

    axi_slave_responder #(
        .C_S_AXI_ADDR_WIDTH        (ADDR_WIDTH             ),
        .C_S_AXI_ID_WIDTH    (ID_W_WIDTH         ),
        .C_S_AXI_DATA_WIDTH  (WDATA_WIDTH       ),
        ) axi_slave (
        .S_AXI_ACLK      (CLK        ),
        .S_AXI_ARESETN   (ARESETn    ),
        .S_AXI_AWID      (AWID       [i]),
        .S_AXI_AWADDR    (AWADDR     [i]),
        .S_AXI_AWLEN     (AWLEN      [i]),
        .m_if (m_if[i])
    );
    end
```

```
endgenerate

cbmem #(
    .ADDR_WIDTH (ADDR_WIDTH),
    .DATA_WIDTH (WDATA_WIDTH)
    ) memory (
        .m_if (m_if[0]),
        .mp_if(mp_if)
);

periph #(
    .ADDR_WIDTH (ADDR_WIDTH),
    .DATA_WIDTH (WDATA_WIDTH)
    ) periph (
        .m_if (m_if[1]),
        .mp_if(mp_if)
);
```

Example 12. Connecting VIP to the DUT

The two instantiated AXI VIP instances provide a pre-defined API, including high-level methods like `write32` and `read32`. These methods abstract away the low-level protocol details, allowing the verification engineer to perform basic read and write operations over the AXI interface. To deploy the AXI VIP, we simply update the sequence hierarchy as shown in Figure 5.

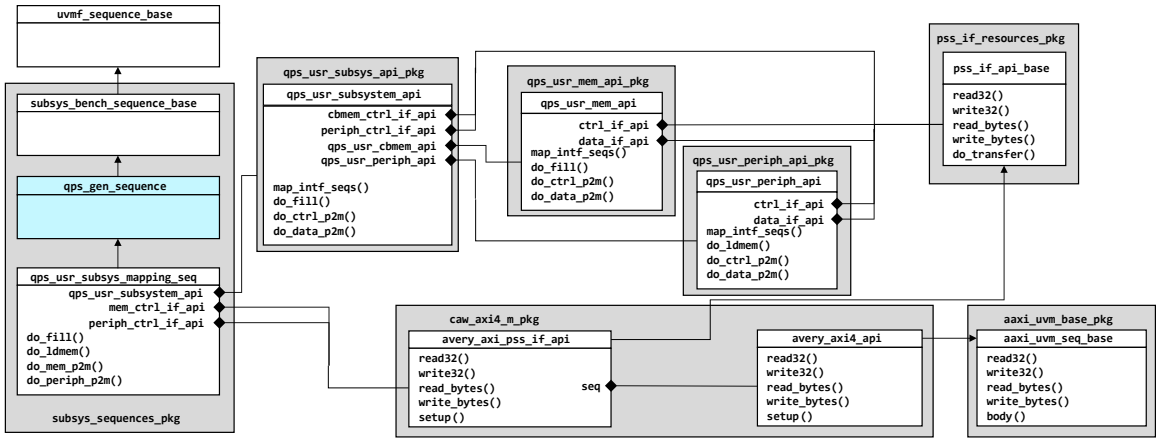

Figure 5: UVM AXI/AXI Sequence Hierarchy

The mechanism is similar to that shown in Figure 3 with the exception that the `mem_ctrl_if_api` and `periph_ctrl_if_api` sequence instances in `qps_usr_subsys_mapping_seq` are now of type `avery_axi_pss_if_api`, which is a PSS-specific extension to the existing `avery_axi4_api` sequence which is part of the Siemens Avery family of VIP components.

Since each action in the PSS model calls the same imported method, which calls the same method of the `mapping_seq` sequence, there is no need for the PSS model to change. However, since the AXI protocol supports

burst read and write operations, we now have the option of implementing operations like `do_fill` as a single AXI burst write operation instead of a loop of individual word writes as we did previously.

Changing Peripheral AXI to APB

As verification environments evolve, it is common to transition between bus protocols depending on system requirements. In SoC designs, the AXI protocol, known for its high throughput and burst capability, is often used for high-speed memory transactions. However, when interacting with low-latency peripherals, the Advanced Peripheral Bus (APB) protocol is more appropriate due to its simpler design and lower power consumption. This section discusses how the PSS model can be adapted to accommodate the change from an AXI-based peripheral to an APB-based peripheral, focusing on the necessary updates to both the UVM environment and the PSS realization layer.

In this updated environment, the Design Under Test (DUT) now contains a peripheral that interfaces over the APB protocol instead of AXI.

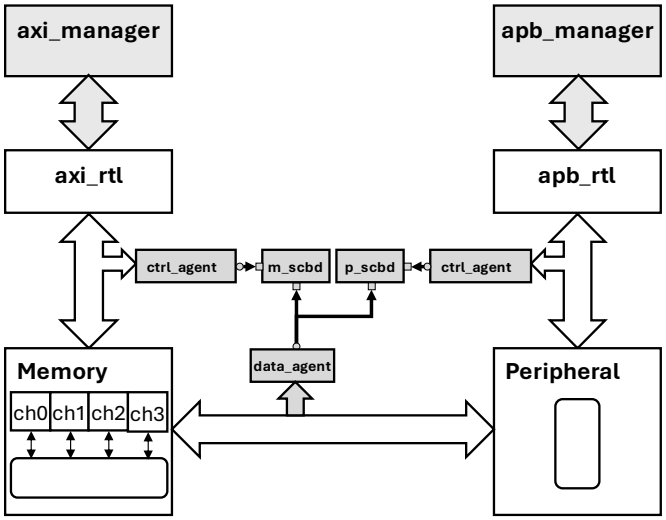

Figure 6. Mixed AXI/APB UVM Environment

APB transactions are simple and occur in a single clock cycle, making the protocol ideal for low-latency peripheral control. The APB protocol reduces overhead by eliminating the complex handshaking and burst mechanisms present in AXI, simplifying both hardware and software. With the transition from AXI VIP to APB VIP, the PSS model remains unchanged at the abstract level. However, specific adjustments are needed in the realization layer to map abstract actions such as `periph_ldmem` and `periph_p2m` to the APB-specific transactions, which are simpler and more direct compared to AXI.

Since we've already set up our sequence hierarchy to be reusable, we just need to modify our `qps_usr_mapping_seq` sequence to account for the new type of the `periph_ctrl_if_api` sequence type.

```
class qps_usr_subsystem_mapping_sequence extends qps_gen_sequence;
  `uvm_object_utils( qps_usr_subsystem_mapping_sequence )
  avery_axi_pss_if_api mem_ctrl_if_api;
  avery_apb_pss_if_api periph_ctrl_if_api;
  qps_usr_subsystem_api qps_usr_subsystem_api;

  task body();
    qps_usr_subsystem_api.map_interface_sequences(
                                .mem_ctrl_if_api(mem_ctrl_if_api),
                                .periph_ctrl_if_api(periph_ctrl_if_api));
    ...
    super.body();
  endtask
```

```
endclass
```

Example 13: Mapping the Protocol-specific API Sequences to Different Design-specific API methods

The sequence class hierarchy for this new structure is shown in Figure 7

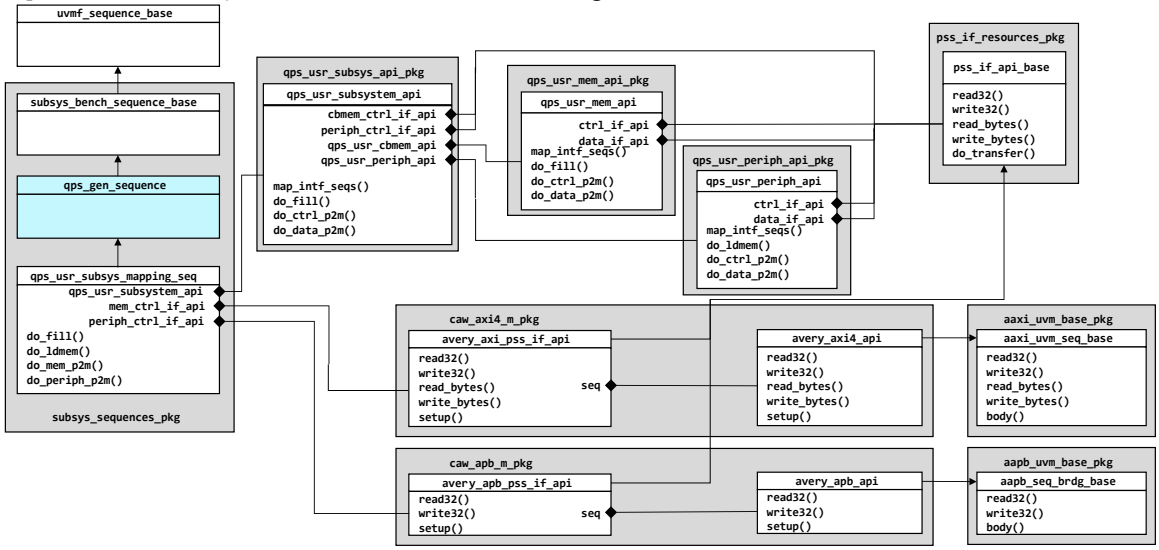

Figure 7: AXI/APB VIP Sequence Hierarchy

The changes required in the UVM environment are subtle, but each unique UVM environment requires a different `qps_usr_mapping_seq` sequence. This will result in a lot of copying and pasting across sequences in order to replicate the same scenario for multiple environments. With PSS, as we have shown, the stimulus model stays constant and we rely on the tool, such as Questa Portable Stimulus, to generate the environment-specific implementation of the scenario.

V. Adding Executors

In PSS, an executor acts as a link between a test's abstract model and its implementation on software or hardware platforms. Executors manage the platform-specific details, ensuring that the test operates correctly on the targeted platform by interpreting PSS actions and resource constraints. They simplify the mapping of test scenarios into the appropriate platform-specific instructions, ensuring compatibility with the intended hardware resources.

Defining executor traits

To bind test scenarios to specific hardware, PSS defines executor *traits*, which describe the functional and behavioral characteristics of the executor. These traits are essential to ensure that the right executor is matched to the test scenario, enabling accurate interaction with the hardware.

In PSS, executors are equipped with two primary traits: device and protocol. These traits help to describe how the executor interacts with various system elements, including peripherals, memory, and communication buses.

The device field represents the type of hardware resource that an executor is designed to interact with. This field can specify various devices, including system memory, I/O peripherals, or specialized hardware components. For example, an executor handling memory accesses could be responsible for managing transactions between the CPU and the system's mem or periph.

```
component my_axi_executor_group_c : executor_group_c<vip_trait_s> {
    my_axi_executor_c axi_exec[2];

    exec init_down {
        axi_exec[0].trait.id = 0;
        axi_exec[0].trait.device = mem;
        add_executor(axi_exec[0]);
```

```
            axi_exec[1].trait.id = 1;
            axi_exec[1].trait.device = periph;
            add_executor(axi_exec[1]);
        }
    }
```

The protocol field defines the communication interface used by the executor. As discused earlier, AXI (Advanced eXtensible Interface) is commonly used for high-speed memory accesses, while APB (Advanced Peripheral Bus) is used for peripheral communications that require lower bandwidth.

```
component vip_group_c : executor _group_c<vip_trait_s> {
    pcie_executor_c pcie_exec;
    axi_executor_c axi_exec;
    ahb_executor_c abh_exec;

    exec Init_down {
        pcie_exec.trait.vip = PCIE;
        add_executor(pcie_exec);
        axi_exec.trait.vip = AXI;
        add_executor(axi_exec);
        ahb_exec.trait.vip = AHB;
        add_executor(abh_exec);
    }
}
```

Matching executor traits

PSS uses the traits defined for executors to match the appropriate executor to a test scenario. Matching executor traits is all about ensuring that the right executor is chosen for a specific test scenario based on the needs of the test. Executors are like specialized workers who handle different tasks (like reading from memory or interacting with peripherals), and each one has specific characteristics, or "traits," that define what they can do. These traits include things like the type of device the executor works with (e.g., memory or peripheral) and the communication protocol it uses (e.g., AXI or APB).

When you run a test, PSS automatically selects the best executor by comparing the traits required by the test with the traits of the available executors. For example, if a test scenario needs to access memory using the AXI protocol, the system will look for an executor that matches those exact traits i.e., device="memory" and protocol="AXI". This is done using the executor claim, which is like a request that specifies exactly what kind of executor the test needs.

Using executor-specific APIs

In PSS, executors are equipped with their own APIs (Application Programming Interfaces), which allow them to perform specific tasks related to hardware, like reading from memory or communicating with peripherals. These APIs act as higher-level tools that executors use to interact with the system's hardware resources, simplifying complex operations.

In many verification setups, a transactor, or testbench agent, is connected to an I/O interface of the system. This transactor exposes transactional APIs or higher-level stimulus sequences to the PSS tool. These APIs allow the test

scenarios to communicate with the system in a more abstract way, letting the PSS model focus on higher-level intent while the executor handles the lower-level details.

For example, a memory executor might have an API that includes read() and write() functions to handle memory operations. A peripheral executor might expose APIs for sending and receiving data to an I/O device or over communication buses like UART or SPI.

By using these executor-specific APIs, PSS helps simplify complex hardware interactions, making the testing process more efficient and flexible across different platforms. Follow-on work to this paper will involve extending the methodology to utilize executors to further improve the reusability of models that may require protocol- or device-specific knowledge as a key part of the test scenario.

VI. Automating UVM Environment Creation

UVM Framework

The Universal Verification Methodology Framework (UVMF)[3] is an advanced and comprehensive toolset that extends the capabilities of UVM, the Universal Verification Methodology. UVMF provides a robust and structured approach to verification, offering a wide range of pre-built components, utilities, and testbenches that accelerate and simplify the verification process.

With UVMF's flexible architecture, verification engineers can effortlessly customize and integrate the components into their specific projects, fostering reusability and scalability. By leveraging UVMF, verification teams can significantly reduce development time, enhance collaboration, and ensure the delivery of high-quality, error-free semiconductor designs to meet the ever-increasing demands of the electronics industry. Although not required to implement the methodology described in this paper, the examples were generated using UVMF.

Generating UVM Environment

Using a code generator to create the UVM environment and test bench is an efficient way to create code that is correct by construction. The abstract characterization input to UVMF's code generator provides the information required to generate environments and test benches that are PSS ready. This information includes the number of agents within an environment and the type of each agent. The active agents in a simulation are also identified. This allows the code generator to identify which agents could be used in the definition of user API's.

Integrating PSS

The code generator in UVMF automatically integrates PSS into the test bench by generating the required infrastructure. This infrastructure includes class methods the user will complete. These classes are automatically included in the environment package and test package.

The first step of integration is to generate the PSS sequence and the package which contains the user defined API's using Questa's Portable Stimulus Compiler, `qpsc`. When executing `qpsc`, the user identifies the top-level virtual sequence which will be the base class for the generated `qps_gen_sequence`. This provides `qps_gen_sequence` with access to test bench level resources needed by PSS to initiate interface level operations on the DUT. The resources include sequencer handles and other resources needed by the VIP sequence to provide read and write operations on the DUT interface. The package generated by `qpsc` contains an api class with methdos defined by the user in the PSS code. The user completes the method bodies in the second step of integration. The packages and sequences created by the PSS tool are automatically integrated into the test bench by the tool flow as outlined above.

The above mapping allows selection of the PSS test using a simple factory override.

```
class qps_test extends test_top;
    `uvm_component_utils( qps_test );

    function new(string name = "qps_test", uvm_component parent = null );
      super.new(name,parent);
    endfunction

    virtual function void build_phase(uvm_phase phase);
      // The factory override below replaces the default top-level sequence
```

```
      // with the QPS sequence
      subsystem_bench_sequence_base::type_id::set_type_override(
                  qps_usr_subsystem_mapping_sequence::get_type()
                  );
      super.build_phase(phase);
    endfunction

endclass
```

CONCLUSION

PSS encourages verification engineers to focus on defining test scenarios at an abstract level, without initially worrying about the underlying target environment in which the scenario will ultimately be realized. This scenario-based approach is complementary to the UVM, which provides a protocol-specific transaction-level abstraction layer to isolate the *what* of a particular operation from the *how* of the pin-level communication to the DUT. The PSS abstract model raises this level of abstraction to *actions* that define behaviors to be exercised, each of which may consist of multiple transaction-level operations. Reuse can be maximized with PSS by combining the powerful features of the language with the judicious of packages in both PSS and SystemVerilog to allow the abstract verification scenarios to be targeted to a variety of standard protocol-specific VIP components, such as the Avery VIP library from Siemens EDA.

In this paper, we have shown how to organize a PSS model and accompanying UVM sequences to use different realizations of actions depending on the desired protocol to be used in the target environment. Whether the target is a non-standard user-defined block-level protocol, or a more complex UVM environment including multiple different VIP components, the same scenarios can be realized throughout.

When it comes to creating the UVM target environment, the use of a UVM code generator like UVMF can greatly accelerate test bench creation. Its integration with Questa Portable Stimulus Compiler, `qpsc`, accelerates integration of a PSS model into a UVM test bench. Using Avery VIP from Siemens EDA can also accelerate the required mapping from generic interface writes and reads to specific interface level operations for DUT stimulus.

REFERENCES

[1] Accellera, "Portable Test and Stimulus Standard Version 3.0", https://www.accellera.org/images/downloads/standards/pss/Portable_Test_Stimulus_Standard_v3.0.pdf

[2] Covey, Stephen R. *The 7 Habits of Highly Effective People: Powerful Lessons in Personal Change*. New York: Free Press, 1989.

[3] https://verificationacademy.com/topics/uvm-universal-verification-methodology/uvmf/uvm-framework/