Divide and conquer: Techniques for creating highly reusable stimulus generation process

Ning Guo

Paradigm Works Incorporation
Andover, MA U.S.A
ning.guo@paradigm-works.com

Abstract— **Stimulus generation is very important in verification processes. A well structured stimulus generation mechanism not only ensures better coverage to the DUT's functionalities but also significantly enhances reusability of the testbench. This paper focuses on the strategies to make stimulus representation and generation processes easy to reuse. Pseudo codes are provided for the proposed mechanism. VMM and OVM based System Verilog implementations are also presented for an example router testbench.**

I. Introduction

Stimulus generation is very important in verification processes. A well structured stimulus generation mechanism not only ensures better coverage to the DUT's functionalities but also significantly enhances reusability of the testbench.

A stimulus generation process can be broken up to three layers. From bottom up, the three layers are:

a. Layer one - the single stimulus stream generation layer. This layer is responsible for providing transaction streams to a specific interface protocol;

b. Layer two – the stimulus managing layer. This layer controls and coordinates stimulus generations to all interfaces of the entire testbench;

c. Layer three – the stimulus selection or customization layer. This layer determines which kind of stimulus a testcase will invoke.

Figure 1 illustrates the layered stimulus generation structure.

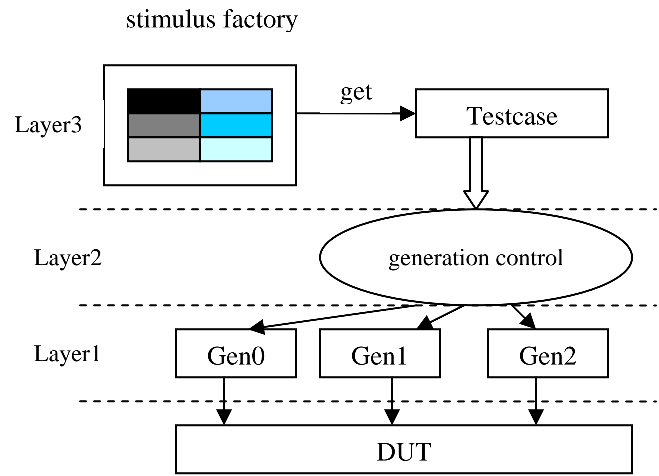

Figure 1 Layered Stimulus Generation

Besides the aforementioned stimulus generation layers, the stimulus itself can be represented as two layers: the lower layer is the atomic transaction, the upper layer is groups of the atomic transactions, i.e., sequences or scenarios.

In terms of reusing a stimulus generation process, the following variations need to be considered:

- The variations of the atomic transaction format
- The variations of how atomic transactions are grouped to form the sequence or the scenario
- The variations of the testbench topology and/or configuration

Sections II and III will look into each layer of the stimulus representation and generation, highlight the areas that are critical for preserving good reusability, and propose a mechanism that can be employed to achieve high reusability.

Sections IV applies the proposed stimulus generation mechanism to a router testbench using VMM and OVM implementations. In the end, this paper briefly lays out the future work.

II. Stimulus Representation

The stimulus generation process consists of two aspects, one is how to represent a stimulus, and the other is how to generate the stimulus. This section is focusing on the first aspect, i.e., the stimulus representation. Section III will discuss the details about the stimulus generation aspect.

2.1 Interface-aware stimulus representation

A device under test (DUT) and its testbench are connected through interfaces. To exercise the DUT, the testbench needs to create the interface-specific stimulus and send the stimulus to the corresponding interface. The stimulus eventually propagates to the physical pins of the DUT and makes the DUT perform certain operations.

Different interfaces require different kinds of stimulus. Therefore a stimulus can be identified by the interface type it is targeted for. However, there are multi-protocol layer interfaces. Different protocol layers also require different stimulus formats. Once the interface type and the protocol layer information are known, a stimulus representation can be uniquely determined.

Because of the aforementioned correlation between an interface and the stimulus to the interface, we can make stimulus representations to be interface-aware. The advantages of having an interface-aware stimulus include,

- We can use the interface information to allocate a unique stimulus;
- A stimulus sequence or a scenario, i.e., a stream of stimulus that has a specific order, can be represented by the sequence of the interface information. The sequence of the interface information is easier to randomize and manipulate.
- Easy for reuse. The stimulus developed for an interface type can be directly reused by any testbench which also has the same interface type to test.

2.2 More features to support stimulus reuse

Besides interface-awareness, there are other features that when built into the stimulus representation can improve the reusability of the stimulus.

- Handles to hold the blueprint stimulus, i.e., the stimulus that will be reproduced and the reproduced copies will be passed around, of each protocol layer
- A pointer to the stimulus factory, i.e., the class that contains all the stimulus descriptors, stimulus classes and stimulus sequences defined in the testbench
- Method to obtain the blueprint stimulus of a specified protocol layer
- Method to copy (or clone) self
- Method to convert current stimulus to the stimulus of a different protocol layer or a different interface

2.3 Base stimulus representation

In the following subsections, we will present the pseudo codes of the interface-aware and reusable stimulus representation. This stimulus representation will serve as the base class for all the interface specific stimulus representations.

2.3.1 Stimulus descriptor class

Before creating an interface-aware stimulus, we need to create a stimulus description class to store the information that can uniquely describe a stimulus format.

```
class stimulus_descriptor;
    string stim_kind;

    string  itf_name; // interface name
    string prot_layer_name; //protocol layer name
    string stim_name;

    function stimulus_descriptor copy();
    // Derive itf_name,prot_layer_name,stim_name from
    // stim_kind
    function void parse_stim_kind();
    ….
endclass
```

Pseudo Code 1 Stimulus Descriptor Class

In stimulus_descriptor class, the *'stim_kind'* determines the value of the rest class members. By default, '*stim_kind' = 'itf_name'_'prot_layer_name'_'stim_name'*.

Function *parse_stim_kind()* can look at the *'stim_kind'* to set the string value for *'itf_name'*, *'prot_layer_name'* and *'stim_name'*.

2.3.2 Base Stimulus Class

```
class base_stimulus extends <base_data>;
    stimulus_descriptor          stim_descr;
    stimulus_factory             parent;

    //  Hold the blueprint data structure for each protocol layer
    <base_data> data4prot_layer[string];
    <base_data> transaction;

    // Method Prototypes
     task post_randomize();
     virtual function <base_stimulus> copy();
     virtual function stimulus_descriptor create_descr();
     virtual function <base_data> create_transaction();
     virtual function int convert2prot_layer(string o_layer,
                                        base_stimulus ostim[$]);
     virtual function int convert2itf(string o_itf,
                                     base_stimulus ostim[$]);
     virtual function void blueprint4prot_layer(string layer,
                                             <base_data> blueprint);
     …..

endclass
```

Pseudo Code 2. Base Stimulus Class

In the base_stimulus class, the stimulus descriptor property defines the default stimulus kind. Extended stimulus class will assign the stimulus descriptor to support a specific interface and a specific protocol layer using a specific stimulus blueprint (defined by 'stim_name' in the stimulus descriptor).

The *data4prot_layer* associative array holds the blueprint data structure for each protocol layer of the interface.

The *data4prot_layer* array needs to be populated before the stimulus object is randomized because in the *post_randomize()* process, the blueprint stored in the *data4prot_layer[stim_descr.prot_layer_name]* is randomized and copied over to the *transaction* property.

The *transaction* object is used to hold the actual stimulus for the specific interface and the protocol layer.

The *post_randomize()* process creates the *transaction* object from the blueprint data for an interface protocol layer.

The *copy()* process creates a copy for the current stimulus object.

The *convert2prot_layer()* and *convert2itf()* methods are used to convert the current stimulus representation to a different format representation. The different representation can be for a different protocol layer of the same interface type, or for a different interface type. The conversion process outputs one or more new stimulus objects. The multiple outputs happen when each new stimulus is only a segment of the current stimulus.

The *blueprint4prot_layer()* is used to build the *data4prot_layer[]* array.

The *create_descr()* function adds a stimulus descriptor to the stimulus factory. Stimulus factory is discussed in section 2.4.

The *create_transaction()* function generates a copy of the blueprint data.

The base_stimulus class also has display methods (not shown in the pseudo codes) to print out the details of the stimulus.

The methods declared in the base stimulus class are virtual methods. The detailed implementation of each method is up to the extended stimulus classes.

2.4 Stimulus factory

We have briefly mentioned that a stimulus factory is a class that contains all the stimulus descriptors, stimulus classes and stimulus sequences defined in the testbench. Having such a central place to keep all the stimulus information provides great flexibility for stimulus generation processes. The pseudo codes of a stimulus factory are shown below.

```
class stimulus_factory;
  stimulus_descriptor   defined_descr;
  string                defined_kind[$];
  //scenario descriptor stores stimulus kinds for each scenario
  scenario_descriptor   scenario_descr[string];

  function base_stimulus create_stimulus(string kind);
     // To be implemented in transactor/testbench scope.
     // It creates a copy from the blueprint stimulus
     // associated with the input kind.
  endfunction:create_stimulus

  function string create_scenario_descr(string sc_name);
    //Generate random stimulus descriptor queue and save the
    // queue to the factory
  endfunction
endclass

class scenario_descriptor;
    e_stim_kind  stim_kind_q[$];
endclass
```

Pseudo Code 3. Stimulus Factory Class

2.5 Atomic stimulus and stimulus scenarios/sequences

The following pseudo codes show an example that randomly generates ten stimulus objects to an interface X at protocol layer A. This generation process is considered as 'atomic' because each stimulus object is independently generated and has no correlation with the preceded or the subsequent stimulus object.

```
 // Create the stimulus class
class my_stimulus extends base_stimulus;
     my_data  stim_data;   // interface specific transaction
     …
endclass

// In generator's main() process
task my_generator::main();
    for (int i=0; i<10; i++)  begin
         rand_stim_data = my_stimulus_inst.create_transaction()
         send2downstream(rand_stim_data);
    end
endtask
```

Pseudo Code 4. Atomic Stimulus Generation

In the above example, the *send2downstream()* process sends the stimulus to the downstream component for driving the DUT.

As we have mentioned, making the stimulus representation interface-aware can simplify the process of creating a scenario or a sequence. A scenario can be represented by an array of the stimulus descriptors, i.e., the scenario_descriptor. The pseudo codes below shows an example of generating a scenario. The scenario consists of ten stimulus objects to interface X at protocol layer A. All the even number stimulus objects in the scenario have *stim_name* =SHORT, all the odd number stimulus objects in the scenario have *stim_name-* =LONG.

Step 1 – Define interface stimulus

```
class my_stimulus extends base_stimulus;
      my_data    stim_data;
      virtual function <base_data> create_transaction();
         my_data tr;
         // Define SHORT/LONG stimulus
         case(stim_descr.stim_name)
           "SHORT": stim_data.randomize() with {len<10;};
           "LONG": stim_data.randomize() with {len>100;};
         endcase
         $cast(tr, stim_data.copy());
         return tr;
      endfunction
endclass
```

Pseudo Code 5. Scenario Generation (Step 1)

Step 2 – Define Scenario, i.e., ten my_stimulus objects with the following pattern: items {0,2,4,6,8} are SHORT, items {1,3,5,7,9} are LONG.

```
stimulus_factory::create_scenario_descr(string sc_name);
  string v_stim_kind;
  scenario_descr[sc_name] = new;

  if (sc_name == "PKT_SC_SHORT_LONG") begin
   for (int i==0; i<10; i++)
     if (i%2==0) v_stim_kind = PKT_NA_SHORT;
     else    v_stim_kind = PKT_NA_LONG;
    scenario_descr[sc_name].stim_kind_q.push_back(v_stim_kind);
   end
endfunction
```

Pseudo Code 6. Scenario Generation (Step 2)

Step 3 – generate the specified scenario

```
task stimulus_generator::main();
   my_stimulus stim;

   // rand_stim_data has type: my_data
   for (int i=0; scenario.stim_kind_q.size(); i++)  begin
      stim= factory.create_stimulus(scenario.stim_kind_q[i]);
      rand_stim_data =stim.create_transaction();
      send2downstream(rand_stim);
   end
endtask
```

Pseudo Code 7. Scenario Generation (Step 3)

III. STIMULUS GENERATION

Section II discussed the details of an interface-aware stimulus representation mechanism. This stimulus representation mechanism makes blueprinting to a stimulus or a scenario easy to achieve, and hence provides better reusability support. This section will discuss the second aspect of a stimulus generation process, that is, the generation.

Stimulus is generated by generators. A testbench can have multiple generators with one generator dedicating for one interface of the DUT. A testbench can also have one generator that generates stimulus for all of the DUT's interfaces. We will be focusing on the one generator per interface structure because dedicating one generator to one interface makes the stimulus generation easier to manage and reuse.

3.1 Base stimulus generator

The common features of stimulus generators include,

1) Default stimulus descriptor. It specifies the interface type and the protocol layer that the generator is targeting for
2) Blueprint stimulus. It determines the format of the stimulus that the generator will produce.
3) Blueprint scenario. It is used by the generator when a stimulus scenario needs to be generated

Following is the pseudo codes for the base stimulus generator.

```
class stimulus_gen extends <base_xactor>;
   base_xactor_cfg    cfg;    // configuration descriptor
   base_stimulus       randomized_stim;
   string                   default_stim_kind;
   string                   default_scenario;
   ….
 task generate_stimulus(sting kind);
 task generate_scenario(stimulus_scenario sc);
 function bit is_done();
 task main();
      while (!is_done()) begin
         if (gen_kind==0) generate_stimulus(default_stim_kind);
          if (gen_kind==1) generate_scenario(default_scenario);
        end
 endtask
endclass
```

Pseudo Code 8. Base Stimulus Generator Class

3.1.1 Stimulus Blueprinting

In order to preserve better reusability, we should avoid using 'randomize() with {..}' during stimulus generation. This is because randomize() with {..} can be hard to control from the top level testbench or testcases and hence it prevents the generation process from being reusable. We should always use a blueprint stimulus, which can be easily defined and modified at the top level testbench or testcases.

We have described in section II that all blueprint stimulus are instantiated in the stimulus factory. The base stimulus generator has a handle to the stimulus factory, therefore it has access to all the stimulus blueprints.

3.1.2 Stimulus Interleaving

For channelized interfaces, i.e., the interface has multiple virtual channels, the stimulus heading for a different virtual channel of the interface can be interleaved. To generate interleaved stimulus, one way is to implement a scheduling component that takes non-interleaved stimulus, mixes those stimulus and then applies certain scheduling mechanisms before forwarding the stimulus to the downstream component. For example, a packet generator creates several complete packets, and each packet has a different source ID. The packets are segmented to streams of fragments. Each fragment also carries its source ID. The scheduler schedules the fragments coming from different packets using certain scheduling mechanisms, such as, round robin algorithm, and forwards the interleaved fragments to the driver component.

The key feature of a scheduler is the multiple inputs and single output structure. Therefore, the stimulus with different source ID comes to the scheduler through different queues without blocking each other. The stimulus goes out of the scheduler is in a defined order.

The scheduler can be made generic and reusable. Below shows the pseudo code of a base scheduler class.

```
class base_scheduler extends <base_xactor>;
  // input channels or queues
  <channel>    in_chan[$];
  <channel>    out_chan;

  // Implement scheduling mechanism
  function scheduling(<base_data> in_data[$],
                      output <base_data> out_data[$]);

endclass
```

Pseudo Code 9. Base Scheduler Class

3.2 Layered Stimulus Generation

The following diagram illustrates the layered structure of the stimulus generation process in a multiple interface testbench.

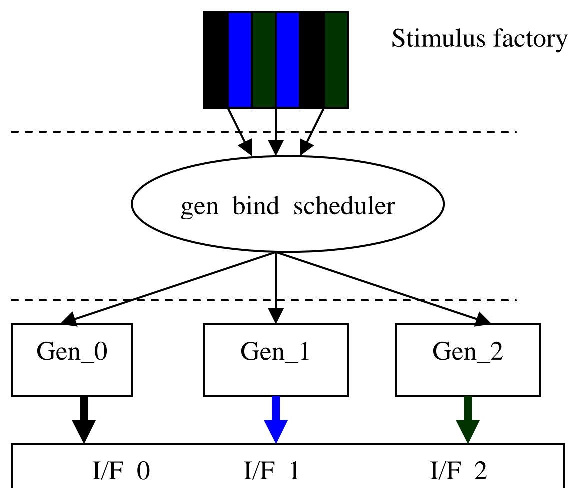

Figure 2. Layered stimulus generation

3.2.1 Top layer – Stimulus Factory

Stimulus factory has been discussed in section II. It contains the blueprint stimulus of an interface transactor or a testbench.

3.2.2 Middle layer – Generator Binding and Scheduling

The second layer of the stimulus generation structure is shown as a *gen_bind_scheduler* component. This component is responsible for assigning blueprint stimulus or scenario to the appropriate generator, that is, the 'binding' functionality. The 'scheduling' functionality of the *gen_bind_scheduler* is used when there is an ordering requirement across different generators. Under such circumstance, the *gen_bind_scheduler* will create a status indicator to keep track of the generation status of each generator. Each generator queries the status indicator prior to start its generation process. The generators also update the status indicator when their generating processes change state. For example, Gen_0 needs to send a stimulus stream to initialize the DUT before Gen_1 and Gen_2 can send data stimulus. The *gen_bind_scheduler* class can define a three bit vector: generator_status. Gen_0 looks at generator_status[0], if it sees '0', it will start its generation process. When Gen_0 completes generating the initialization stimulus stream, it sets generator_status[0] to '1' and stops. Gen_1 and Gen_2 are both waiting for generator_status[0] = 1. They start their generation processes as soon as they see generator_status[0] comes true.

3.2.2.1 Base gen_bind_scheduler

Below is the pseudo code of the base gen_bind_scheduler class.

```
class gen_bind_scheduler;
   // handles to the generators
    stimulus_gen generators[string];
    string       generator_status[string];
    function void bind_blueprints();
      // Implementation is DUT specific. For example,
      // generator[0].default_stim_kind = X;
      // generator[1].default_stim_kind = Y;
    endfunction
endclass
```

Pseudo Code 10. Base gen_bind_scheduler Class

3.2.3 Bottom layer – Generators

The generator layer consists of a number of per interface generators. The base generator class has been discussed in section 3.1.

To allow all generators to access the *gen_bind_scheduler*, the handle of the *gen_bind_scheduler* needs to be passed to each generator.

IV. IMPLEMENTATION EXAMPLE

In this section, we will show how to apply the proposed stimulus generation mechanism to a simple router DUT using VMM and OVM methodology.

Figure 5 shows the block diagram of the simple router DUT.

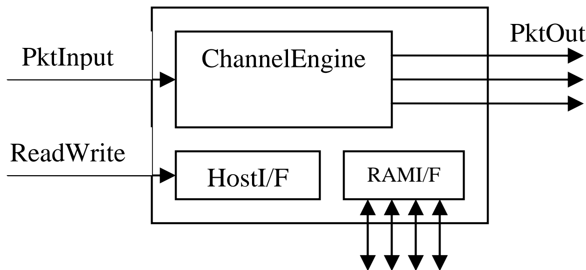

Figure 3. **PWR router block diagram**

Before employing this proposed stimulus generation mechanism, we already have working VMM and OVM testbenches for the DUT. All we need to do are,

1) Implement all the base classes needed by the mechanism;
2) Create wrapper stimulus classes around the existing transaction classes
3) Replace existing generator components with the base stimulus generator
4) Create top level stimulus factory and gen_bind_scheduler classes to manage the stimulus generation

We will show the VMM implementation of the above four processes in section 4.1, and the OVM implementation in section 4.2.

4.1 VMM Implementation Example

1) Base stimulus classes

- stimulus_descriptor class is implemented as a standalone class.
- base_stimulus class extends from **vmm_data**
- stimulus_gen class extends from **vmm_xactor**
- gen_bind_scheduler class is implemented as standalone class
- stimulus_factory class is implemented as a standalone class

2) Stimulus Wrapper Classes

The router device needs two transaction classes, one for each interface: *pi_transaction* and *host_transaction*. They both extend from **vmm_data**.

A *pi_itf_stimulus* class and a *host_itf_stimulus* class are defined to wrap around the transaction classes. They both extend from base_stimulus class as follows.

```
class pi_itf_stimulus extends base_stimulus;
      pi_transaction stim_data;
      extern function new(string inst,
                  stimulus_factory parent);
      extern function void create_transaction();
      extern function vmm_data copy(vmm_data to=null);
endclass
```

The key function of the wrapper stimulus class is the *create_transaction()*. It creates a copy of the blueprint transaction. The following two code snippets show the *create_transaction()* implementation of the *pi_itf_stimulus* and the *host_itf_stimulus.*

```
// Packet Interface Stimulus Blueprinting Transaction
function void pi_itf_stimulus::create_transaction();
  bit    result;
  case (stim_descr.stim_name)
    "SHORT":
    stim_data.SHORT_pkt_constr.constraint_mode(`ON);
    "LONG":
    stim_data.LONG_pkt_constr.constraint_mode(`ON);
   endcase // case(stim_descr.stim_name)
   result = stim_data.randomize();
   if (!result)
     `vmm_error(log,"create_descr: Failed to randomize
                          the stim!");
    $cast(transaction, stim_data.copy());
endfunction:create_transaction

// Host Interface Stimulus Blueprinting Transaction
 function void host_itf_stimulus::create_transaction();
   bit[31:0] v_addr;
   bit [7:0] v_data;
   e_host_rw v_rw;
   bit      result;

   v_data = 0;
   case (stim_descr.stim_name)
     "INIT_CTRL": begin
        v_addr = 32'h56740000;
        v_data = 8'h07;
        v_rw   = e_HOST_WR;
     end
     …
    endcase // case(stim_descr.stim_name)
     result = stim_data.randomize() with {addr==v_addr;
                                     rw==v_rw;
                                     data== v_data;};
  if (!result)
     `vmm_error(log,$psprintf("create_descr: Failed to
          randomize stim_%s!", stim_descr.stim_name));
   $cast(transaction, stim_data.copy());
endfunction:create_transaction
```

3) Replace Stimulus Generator

The original testbench has a packet interface transactor and a host interface transactor. Each transactor has an interface specific generator, which extends from the base **vmm_xactor**.

We replaced the interface specific generator with our base stimulus_gen class as follows,

```
class pi_xactor extends vmm_xactor;
   // Original: pi_gen    gen;
```

```
    stimulus_gen          gen;
    pi_driver              drv;
  …
  endclass
```

The VMM version of the base stimulus_gen class is shown below.

```
class stimulus_gen extends vmm_xactor;
   base_xactor_cfg    cfg;   // configuration descriptor
   base_stimulus       randomized_stim;
   string                  default_stim_kind;
   string                  default_scenario;
   vmm_channel        out_chan;
   // Other members (omitted)

 task generate_stimulus(sting kind);
    // calling pre_gen callback (omitted)
    randomized_stim=factory.create_stimulus(stim_kind);
    // calling post_gen callback (omitted)
    out_chan.put(randomized_stim.transaction);
endtask

 task generate_scenario(stimulus_scenario sc);
     string  v_stim_kind;
     scenario_descriptor v_sc_descr;
      int     v_sc_len;
      // calling pre_scenario_gen callback (omitted)
     v_sc_len = factory.create_scenario_descr(sc_name);
    for(int i=0; i<v_sc_len; i++) begin
    v_stim_kind=factory.scenario_descr[sc_name].stim_kind_q[i];
      generate_stimulus(v_stim_kind);
    end
    // calling post_scenario_gen callback (omitted)
    scenario_cnt++;
endtask:generate_scenario
     ….
endclass
```

4) Top level stimulus control

At the testbench level, we built a DUT specific *pwr_stimulus_factory* and a *pwr_gen_bind_scheduler* to manage the stimulus generation. The *pwr_stimulus_factory* extends the base stimulus_factory. It instantiates a *pi_itf_stimulus* instance and a *host_itf_stimulus* instance. The following example code shows the process of blueprinting the packet and the host stimuli and the scenarios.

```
// PWR_STIMULUS_FACTORY::create_stimulus
function base_stimulus create_stimulus(string stim_kind);
    bit result;
    pi_itf_stimulus v_pkt_stim;
    host_itf_stimulus v_host_stim;

    // log message (omitted)
   if (stim_kind.substr(0,2)=="PKT") begin
        `vmm_note(log,"Interface is PKT");
        pkt_stim.stim_descr.set_descriptor(stim_kind);
        result = pkt_stim.randomize();
        if (!result) `vmm_error(…);
        $cast(v_pkt_stim, pkt_stim.copy());
        return v_pkt_stim;
   end
   else if (stim_kind.substr(0,3)=="HOST") begin
        host_stim.stim_descr.set_descriptor(stim_kind);
        result =  host_stim.randomize();
        if (!result) `vmm_error(…);
        $cast(v_host_stim, host_stim.copy());
        return v_host_stim;
    end
endfunction:create_stimulus
```

```
// PWR_STIMULUS_FACTORY::create_scenario_descr
function int create_scenario_descr(string sc_name);
    string v_stim_kind;
    int    sc_len;
    scenario_descr[sc_name]  = new;
   case (sc_name)
      "HOST_SC_INIT":begin
    v_stim_kind = "HOST_NA_INIT_CTRL";
     scenario_descr[sc_name].stim_kind_q.push_back(v_stim_kind);
               v_stim_kind = "HOST_NA_INIT_FIFO_CLEAR";
     scenario_descr[sc_name].stim_kind_q.push_back(v_stim_kind);
              v_stim_kind = "HOST_NA_INIT_CTRL_CHECK";
     scenario_descr[sc_name].stim_kind_q.push_back(v_stim_kind);
 endcase // case(sc_name)
 sc_len = scenario_descr[sc_name].stim_kind_q.size();
return sc_len;
endfunction:create_scenario_descr
```

The *pwr_gen_bind_scheduler* class extends the base gen_bind_scheduler class. In the example below, the *pwr_gen_bind_scheduler* passes the handle of the blueprint stimulus to the corresponding generator in the *bind_blueprints()* function. It instructs the packet interface generator to wait for the host interface generator to complete the initialization process in the *main()* task.

```
// PWR_GEN_BIND_SCHEDULER::bind_blueprints
function void bind_blueprints();
    generators["PWR_PI_gen"].default_stim_kind=
                                                        "PKT_NA_RAND";
        generators["PWR_HI_gen"].default_scenario=
                                                        "HOST_SC_INIT";
endfunction:bind_blueprints
```

```
// PWR_GEN_BIND_SCHEDULER::wait_for_ready
function void main();
    // fork off super.main
    …
    // Packet generator needs to wait for Host init to complete
    if  (generators["PWR_HI_gen"].default_scenario
         =="HOST_SC_INIT")
       generators["PWR_HI_gen"].start_xactor();

    // Loop through the rest generators
    …
   while (generators.next(gen_id)!=null) begin
      if (gen_id == "PWR_PI_gen")
         generators["PWR_HI_gen"].notify.wait_for(
                                  generators["PWR_HI_gen"].DONE);
      generators[gen_id].start_xactor();
   end
endtask:main
```

4.2 OVM implementation Example

1) Base stimulus classes

- stimulus_descriptor class is implemented as a standalone class.
- base_stimulus extends **ovm_sequence_base**
- stimulus_gen class extends **ovm_sequencer**

- gen_bind_scheduler class extends **ovm_virtual_sequencer**
- stimulus_factory class extends **ovm_component**

2) *Stimulus Wrapper Classes*

The OVM packet transaction and host transaction are defined as extensions from **ovm_sequence_base.**

Below shows the example stimulus wrapper class, *pi_itf_stimulus,* in OVM. Note that instead of the ***copy()*** function in VMM, we use ***clone()*** function in OVM.

```
class pi_itf_stimulus extends base_stimulus;
   pi_transaction stim_data;
   extern function new(string inst,
                              stimulus_factory parent);
   extern function void create_transaction();
   extern function ovm_object clone();
endclass
```

The OVM implementation of *create_transaction()* is almost the same as the VMM implementation, except for the messaging and the *copy()-clone()* changes. Below shows the *create_transaction()* code for the *pi_itf_stimulus*.

```
// Packet Interface Stimulus Blueprinting Transaction
function void pi_itf_stimulus::create_transaction();
      bit     result;
      pi_transaction v_data;

      case (stim_descr.stim_name)
        "SHORT":
        stim_data.SHORT_pkt_constr.constraint_mode(`ON);
        "LONG":
        stim_data.LONG_pkt_constr.constraint_mode(`ON);
       endcase // case(stim_descr.stim_name)
       result = stim_data.randomize();
        if (!result) `dut_error(…);
        $cast(v_trstim_data.clone());
        transaction = v_tr
endfunction:create_transaction
```

3) *Replace Stimulus Generator*

In our OVM implementation, the base stimulus_gen class replaced the interface specific sequencer in the *pi_master_agent* and *host_master_agent* classes.

In the subsection 3) of section 4.1, we have shown the VMM implementation of the base stimulus_gen class. The main differences of the OVM implementation of the base stimulus_gen class include:

- It extends **ovm_sequencer**
- It implements a *build()* function
- It implements a *run()* instead of the *main()*
- It connects to the driver through TLM export

Below shows the OVM implementation of the *generate_stimulus()* process. The OVM specific portion of the code is highlighted.

```
class stimulus_gen extends  ovm_sequencer;
   base_xactor_cfg    cfg;    // configuration descriptor
   base_stimulus        randomized_stim;
   string                     default_stim_kind;
   string                     default_scenario;
    // Other members (omitted)

 task generate_stimulus(sting kind);
    // calling pre_gen callback (omitted)
    randomized_stim=cfg.factory.create_stimulus(stim_kind);
    // calling post_gen callback (omitted)
    grab(randomized_stim);
    wait_for_grant(randomized_stim);
    send_request(randomized_stim,
                          randomized_stim.transaction);
    ungrab(randomized_stim);
    …
  endtask
endclass
```

4) *Top level stimulus control*

The top level *pwr_stimulus_factory* and *pwr_gen_bind_scheduler* can be easily converted from their corresponding VMM implementation. Below are the main differences to be taken into account during the conversion.

- OVM uses *clone()* vs. VMM uses *copy()*
- OVM uses *convert2string()* vs. VMM uses psdisplay()
- OVM uses **ovm_event** for synchronization vs. VMM uses **notify**

V. FUTURE WORK

In the previous sections, we have discussed the basic idea of the interface-aware stimulus and the layered stimulus generation structure. We also showed the example implementations of this mechanism using VMM and OVM.

In fact, this proposed mechanism is also capable of handling more advanced stimulus generation requirements, such as, dynamic generation, flexible generation termination, etc.

Being able to control all the stimuli and the stimulus generators at the top level testbench makes the stimulus generation process independent of all the lower level variations, such as the data format variation, the interface protocol variation, and so on. In terms of DUT's topological variations, we can use proper configuration mechanism with the proposed stimulus generation process to further enhance the reusability of the proposed mechanism.

Another useful application of the proposed stimulus generation mechanism is that the VIP vendors can build a basic stimulus factory a) to verify their VIPs, 2) to deliver to their users as a jumpstart. The stimulus factory can contain interesting blueprint stimulus for different protocol layers. Users can instantiate the VIP's stimulus factory in their testbench's top level stimulus factory, create a DUT specific gen_bind_scheduler and start generating stimulus for their DUTs.

We are currently working on putting together more examples for the advanced features of the stimulus generation mechanism. We will present our findings in the future.

VI. CONCLUSION

This paper proposes an interface-aware stimulus representation strategy and a layered stimulus generation structure. The main goal of the proposed mechanism is to bring all the control of the stimulus generation process to the top level of a testbench. If we could achieve this goal, we will significantly improve the reusability of the stimulus generation process. The interface-aware stimulus representation makes it easier to identify and select interesting stimulus of an interface. The generic stimulus generator works as a vehicle to pass the stimulus to the driver of an interface. The testbench specific stimulus factory is the single place where all the stimuli defined for a DUT are accessible. The testbench specific gen_bind_scheduler hooks up a stimulus (or scenario) to the appropriate generator, and also coordinates the operations of the generators.

The proposed stimulus mechanism is at a higher level abstraction and hence it is independent of the VMM or OVM methodology.

There is more work need to be done in order to make this proposed mechanism easy to use and robust. We will continue exploring and experimenting this mechanism and will be happy to share our findings with the verification community.

ACKNOWLEDGEMENT

I would like to thank Steve D'Onofrio and Rich Musacchio at Paradigm Works for reviewing the paper draft and providing very valuable feedbacks.

REFERENCES

[1] Verification Methodology Manual for System Verilog

[2] OVM Class Reference, Version 2.0, September 2008

[3] Open Verification Methodology User Guide, Version 2.0, September 2008