8 ADVANCED TOPICS

This chapter addresses some of the advanced concepts in VMM such as scenario generator, scheduler, and broadcaster. Other previously addressed topics are further explained such as messaging servicing, and functional coverage. To demonstrate some of the advanced features of VMM, we use a serial-to-parallel converter design, as it provides a model with different characteristics than the FIFO model. This along with the FIFO model will be used throughout to explain the concepts. The Serial-to-parallel converter is first described.

8.1 SERIAL-TO-PARALLEL CONVERTER DUT INTERFACE

The DUT is a simple serial-to-parallel converter that converts input serial bit stream into parallel output stream of 8-bits. A new serial input data transfer starts when *ser_sop* (Serial Start Of Packet) signal is high for one clock, and the transfer ends when *ser_eop* (Serial End Of Packet) is high. The packet length is arbitrary; hence any number of clocks can elapse between a *ser_sop* and *ser_eop*. The transfer can also be aborted by asserting the signal *ser_abort*. The serial input interface is synchronous to the input clock *clk*. On the output side, the parallel data is sent out via an eight bits wide *par_data* bus. The data on this bus is considered valid when a qualifying signal *par_data_valid* is high. The clock for the parallel interface is same as that of the input side, i.e. *clk*.

Figure 8.1-1 and 8.1-2 demonstrate the S2P interface with the modports used throughout the design and the testbench.[1] We designed a VMM-based verification environment for this S2P design along the lines of what has been addressed in the previous chapters. The files used for this model include:

- *ch8/s2p_xactn.sv* transaction class that describes the basic transaction modeled as a packet with variable length.
- *ch8/s2p_cmd_xactor.sv* command-level transactor.
- *ch8/s2p_atomic_gen.sv* that contains an atomic generator using *`vmm_atomic_gen* macro.
- *ch8/s2p_env.sv* that encapsulates the various pieces of the environment.

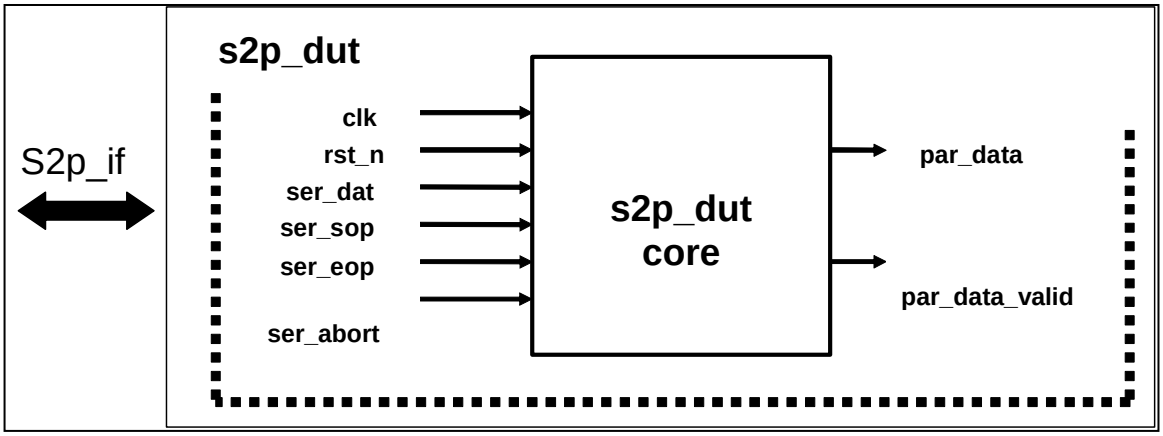

Figure 8.1-1 S2P Interface

[1] VMM Rule 4-9 Individual *modports* shall be declared for each type of proactive, reactive and passive transactors.

```
interface s2p_if (input logic clk, rst_n);
   logic ser_data;
   logic ser_sop, ser_eop, ser_abort;
   logic [7:0] par_data;
   logic       par_data_valid;

   clocking ser_cb @ (posedge clk);
     output ser_data, ser_sop, ser_eop, ser_abort;
   endclocking : ser_cb

   modport ser_drv_mp (clocking ser_cb);

 clocking par_cb @ (posedge clk);
   output   par_data, par_data_valid;
   endclocking : par_cb

   modport par_drv_mp (clocking par_cb);


// Monitor view of the interface.
 clocking mon_cb @ (posedge clk);
     input ser_data, ser_sop, ser_eop, ser_abort;
     input par_data, par_data_valid;
  endclocking : mon_cb

  modport s2p_mon_if_mp (clocking mon_cb);
endinterface : s2p_if
```

Figure 8.1-2 S2P Interface (file *ch8/s2p_if.sv*)

8.2 VMM SCENARIO GENERATOR

Chapter 3 introduced the various types of generators to generate random transactions. While the basic atomic generator is quite easy to use and provides basic randomization features, it is limited in capabilities when the verification requires a "sequence of related transactions". For example:

- A memory controller may need correlation in the sequence of read and write addresses. A video stream may need a color bar of seven colors, with each bar of width 25 pixels.
- A CPU may need instructions that meet particular sequence patterns.

No matter how many atomic transactions are generated, it is very unlikely that you would satisfy such requirements because the specific set of sequences are not likely to randomly occur. This is where a "scenario generator" is very useful. How can scenarios be defined? Before getting into the details of the mechanics, let us first analyze a simple example. Consider the verification of a vending machine design that accepts the following coins: dimes, quarters and dollars. Let us assume that a soft drink costs $2. What are the possible input sequences to get to a soft drink? Following are some of them:

- 5 Dimes, 2 Quarters and a Dollar
- 2 back-to-back Dollars
- 8 continuous Quarters
- 2 Quarters, a Dollar and 2 Quarters

The above list is non-exhaustive, but it represents the kind of scenarios needed for verification. An atomic transaction represents a single coin being inserted - a Dime, a Quarter or a Dollar. However, a scenario generator represents a sequence of coins being inserted, where the sequence satisfies a set of related constraints. Some of the requirements for a scenario generator are:

- It should be able to generate a sequence of atomic transactions, that we call *items*. These *items* represent one single scenario for the user (such as 2 back-to-back Dollars). The length of such a scenario should be constrainable.
- It should allow constraints on the individual elements in the *items* so that we can get to a final count of $2 given a random starting point.

How can such scenario be generated using SystemVerilog? While there are several ways to achieve it, we will show a piece of code using iterative constraints. Figure 8.2-1 shows the simple code to achieve this.

```
typedef enum {DIME, QUARTER, DOLLAR} coin_t;
class Vend_inp_c;
  //Instantiate an array of items to be generated.
   rand coin_t items [];
  // How many items? i.e. the length field
   rand bit [3:0] length;
// Describe one scenario
   constraint cst_2Q_1D_2Q_seq {
      this.length == 4;

      foreach (this.items [i] ) {
        // i==0, i==1, 2 Quarters
         (i < 2) -> this.items[i] == QUARTER;

        // i==2, 1 Dollar
         (i == 2) -> this.items[i] == DOLLAR;

        // i==3, i==4, 2 Quarters
         (i > 2 && i < 5) -> this.items[i] == QUARTER;
      } // foreach
   } // cst_2Q_1D_2Q
endclass : Vend_inp_c
```

Figure 8.2-1 Scenario Generator Using Iterative Constraints

The above code describes just one possible scenario. However, you typically need a set of such scenarios that VMM calls *scenario_set[$]*. The *scenario_set[$]* represents the set of available scenario descriptors that may be repeatedly randomized to create the random content of the output stream. Note that the above code does not show the complete generation of the scenario such as randomizing the items etc.

Conceptually a scenario generator builds on an atomic generator; and imposes constraints on the individual transactions with reference to the previous (or next) transactions. The generation process for the scenario generator is different than the process in the atomic generator. While the atomic generator creates a single object of the transaction, the scenario generator creates a queue of objects of the transaction. Aside from this difference, the two generators are very similar - they both send their output to a single channel. Figure 8.2-2 and 8.2-3 show this concept.

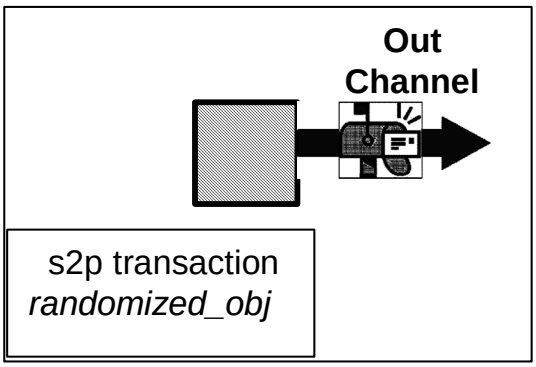

Figure 8.2-2 Atomic generator

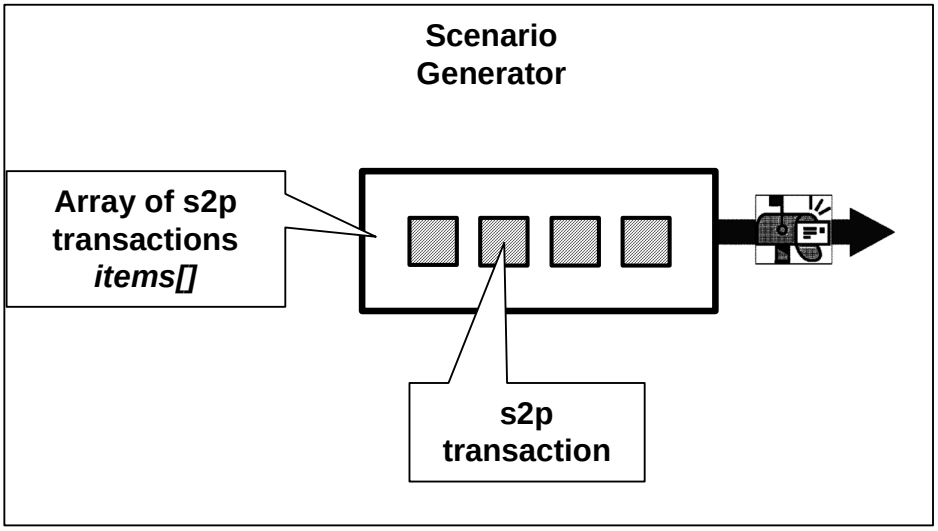

Figure 8.2-3 Scenario generator

Considering the S2P design, the following are some of the potential scenarios, as shown in Figure 8.2-4:

- Alternating a good packet and an error packet.
- A sequence of 5 good packets, followed by an error packet, and then a sequence of 5 good packets.

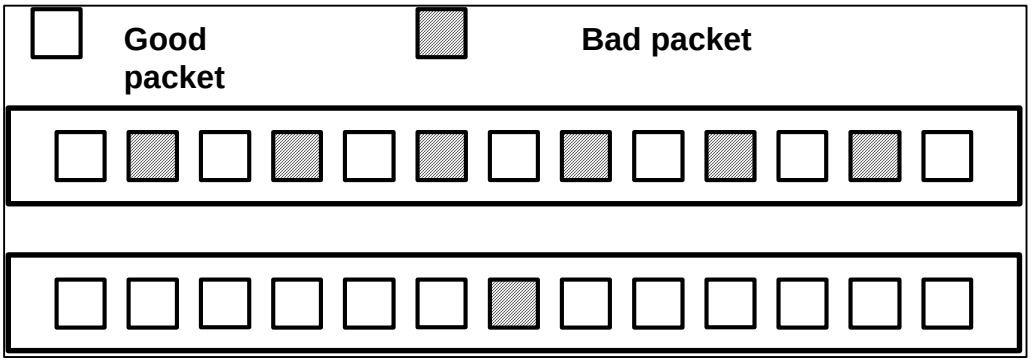

Figure 8.2-4 Sample scenarios for S2P

Such scenario requirements often stem from the higher level system perspective. In this section we demonstrate how to build a scenario generator for the above stated requirements. VMM provides the *`vmm_scenario_gen* pre-built macro to build a scenario generator. It is used with the basic transaction class as an argument. For example,

```
`vmm_scenario_gen(S2p_xactn)
```

This macro creates the required infrastructure for a scenario generator. From a user perspective, it does the following for the above example:

- Creates a transactor named *S2p_xactn_scenario_gen* - this is the top level scenario generator for our design.
- Creates a skeleton scenario class named *S2p_xactn_scenario*.
- This scenario base class contains a SystemVerilog dynamic array named *S2p_xactn_scenario::items[]* of type *S2p_xactn*.
- Declares a variable *S2p_xactn_scenario::scenario_kind* to identify the scenario.
- Provides a mechanism to associate an unique identifier to a scenario via *S2p_xactn_scenario::define_scenario()* method.

Some of the key elements of the VMM scenario generator (called *S2p_xactn_scenario_gen* in our example) are:

- A single scenario is represented by an array of atomic transactions. This is named as *items[]*. The length of this array is user constrainable.
- Individual elements of this items array can be constrained using the SystemVerilog constraint block.
- In a system there can be many different scenarios. Every single scenario is of a unique kind and hence needs to be identified with a unique ID that VMM calls *scenario_kind*. When this object is randomized, it selects the identifier of the scenario that is generated.

Following are the steps to build a scenario generator in VMM.

1. Use the macro *`vmm_scenario_gen* to create the basic infrastructure.
2. Declare a new class for the specific scenario by deriving it from *S2p_xactn_scenario* (This base class is created by the macro). We called the new class *S2p_scen_GP_EP*, which is shown in Figure 8.2-5.
3. Declare an *int* variable to identify this scenario.
4. Use the method *S2p_xactn_scenario::define_scenario()* to associate a unique identifier for the scenario.
5. Add a suitable constraint block to define the desired scenario. This is the key step in defining the scenarios. This constraint block should fix the length of the scenario. To constrain the individual transactions in the scenario, you need to constrain the *items[]* array. SystemVerilog supports iterative constraints that can operate on arrays, which is very useful here.

```
class S2p_scen_GP_EP extends S2p_xactn_scenario;
// an integer to identify the this scenario
  int sc_id_alternate_GP_EP;                          (3)

  function new();
    this.sc_id_alternate_GP_EP = define_scenario
      (.name("Alternating Good and Error Packets"),
       .max_len(10));                                 (4)
  endfunction : new

  constraint cst_GP_EP {                              (5)
    if (this.scenario_kind ==
        this.sc_id_Alternate_GP_EP) {
      this.length == 10; //Constrain the length

      // Constrain individual transactions
     // Note that items[] is declared in base class
      foreach (items [i] ) {
        (i % 2) -> items[i].err_pkt == 0; // Good Packet
       !(i % 2) -> items[i].err_pkt == 1; // Error Packet
      } // foreach
    } // if
   } // cst_GP_EP
endclass : S2p_scen_GP_EP
```

Creating a unique identity for the scenario

Constraint applicable only to this scenario

Figure 8.2-5 Creating of a Scenario Generator (file ch8/s2p_scen_gen.sv)

6. Instantiate the scenario generator in the environment and start its activities. As noted in Chapter 4, this is done in several steps, the instantiation belongs to the structural segment, the allocation of the scenario generator occurs in the *build*() step, and the transactor is started in task *start*(). This is shown in Figure 8.2-6.

```
class S2p_env extends vmm_env;

  S2p_scenario_gen s2p_sc_gen_0; // Variable declaration
  ..
  virtual function void build();
    s2p_sc_gen_0 = new("S2p Scen Gen", 0, s2p_chan_0);      (6a)
    // Instantiation of generator
   ..
  endfunction : build

  task start ();
    this.s2p_sc_gen_0.start_xactor(); // Start of generator   (6b)
    ..
  endtask : start
endclass : S2p_env
```

Fig 8.2-6 Building a Scenario Generator in *vmm_env* class (*ch8/ s2p_scen_env.sv*)

7. Instantiate in the program block the specific scenario, and add it to the *scenario_set[$]* of the scenario generator. The details on this *scenario_set[$]* are provided in Section 8.2.1; for now assume that this is a Queue of scenarios, and you need to push your own scenarios into this Queue.

```
program automatic s2p_scen_pgm;
  // Instantiate the scenario of interest
  S2p_scen_GP_EP s2p_scen_0;

 // ..
  initial begin : b1
    s2p_scen_0 = new();                                    (7a)
    s2p_scen_0.allocate_scenario();
    s2p_env_0.build();
    // Push it to the top level scenario_set queue                (7b)
    // More on this in next section
    s2p_env_0.s2p_sc_gen_0.scenario_set.push_back(s2p_scen_0);
    s2p_env_0.run();
  end : b1
endprogram : s2p_scen_pgm
```

Figure 8.2-7 Instantiating a Scenario Generator in a testcase (*ch8/ s2p_scen_pgm.sv*)

A sample run with this test yields alternating GOOD and ERROR packets as shown in the log file in Figure 8.2-8.

```
1950.00 ns [Normal:NOTE] | S2P GOOD   Pkt, data b6a859d4
2950.00 ns [Normal:NOTE] | S2P ERROR Pkt, data 66b4a122
3950.00 ns [Normal:NOTE] | S2P GOOD   Pkt, data 33c34598
4950.00 ns [Normal:NOTE] | S2P ERROR Pkt, data 102030aa
5950.00 ns [Normal:NOTE] | S2P GOOD   Pkt, data aa020398
6950.00 ns [Normal:NOTE] | S2P ERROR Pkt, data bb559087
```

Figure 8.2-8 Simulation result of S2P using scenario generator

8.2.1 Handling multiple scenarios

The verification of complex designs requires a set of scenarios that are randomized to simulate complex testcases. To appreciate the need for multiple scenarios, consider a networking system that handles internet traffic. While the major portion of the traffic is data packets, there are some control packets, and other types of packets typically sent to a host processor for analysis. To simulate this system, one requires generating: "A specific, focused (directed) scenario - such as downloading a video stream, followed by one (or few) random transaction and then again a directed scenario." To create this kind of mixed scenarios, the VMM scenario generator provides the following infrastructure:

- A SystemVerilog Queue of scenarios, *scenario_set[$]* that represents a set of scenarios, with each scenario containing a number of transactions.
- A mechanism to choose one of the scenarios as the next candidate. This is supported via an election class named <*class_name*>*_scenario_election*.

With this you can create a few scenarios of interesting combinations, and have them added to the *scenario_set[$]* queue. You can have the election class pick one of the scenarios as per a pre-defined election policy. This is represented in Figure 8.2.2-1.

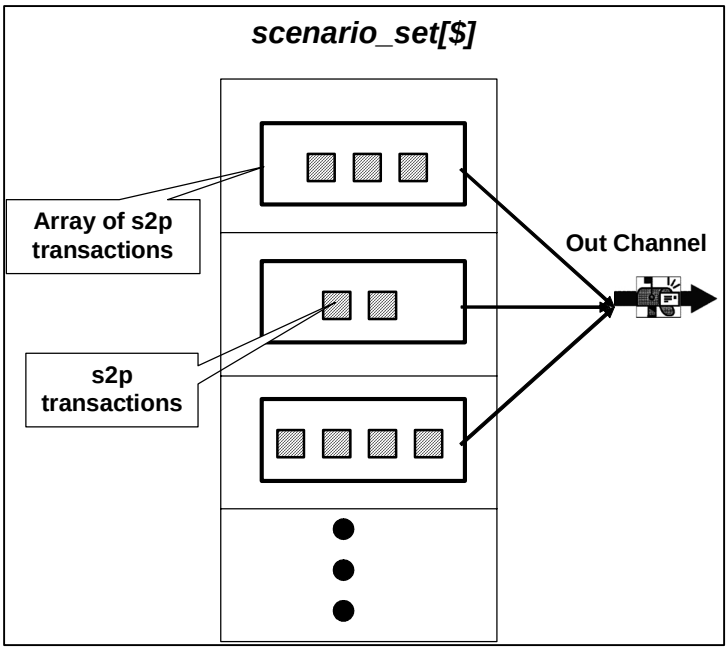

Figure 8.2.1-1 Scenario Generator

It is important to note that the macro introduced in the previous section, *`vmm_scenario_gen* creates these components automatically for the user. Figure 8.2.1.-2 shows a more comprehensive UML diagram of the entire infrastructure that gets generated by this macro.

For the verification of complex designs, it is common for verification engineers (or vendors of complex interfaces) to create libraries of comprehensive sets of scenarios that can be used during the verification process. These libraries can be packaged in SystemVerilog packages for direct use.

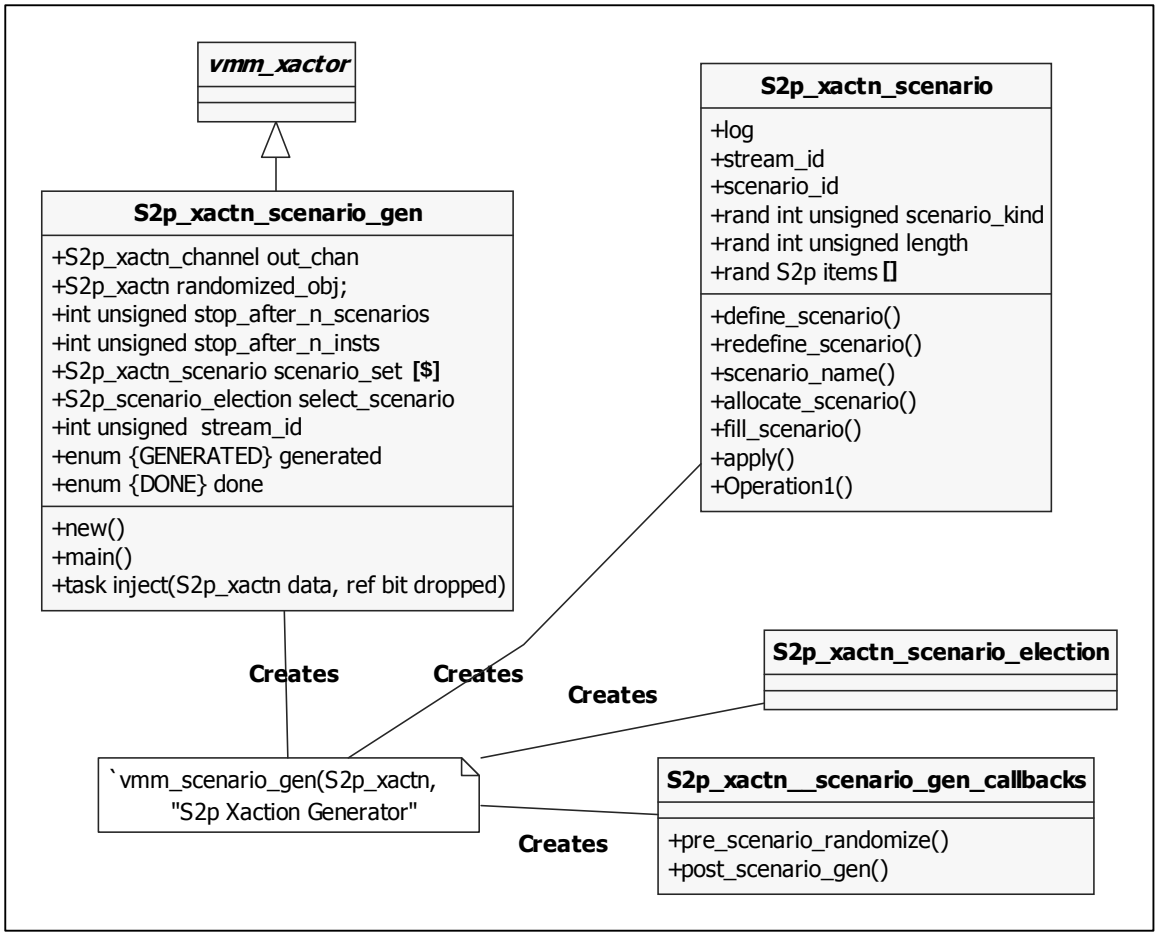

Figure 8.2.1-2 UML view of a Scenario Generator

8.2.2 Default scenario_set created by the `vmm_scenario_gen macro

So far we have shown the minimum steps required to use the *`vmm_scenario_gen* scenario generator macro to produce a realistic scenario. Given that this macro is generic and design independent, the default scenario it creates has to be generic as well. The default scenario that this macro creates is an atomic transaction, but embedded inside the *items[]* array, as demanded by a generic scenario. In other words, the default scenario that this macro creates is of length ONE, and is exactly the same as an atomic transaction. This is shown in Figure 8.2.2-1

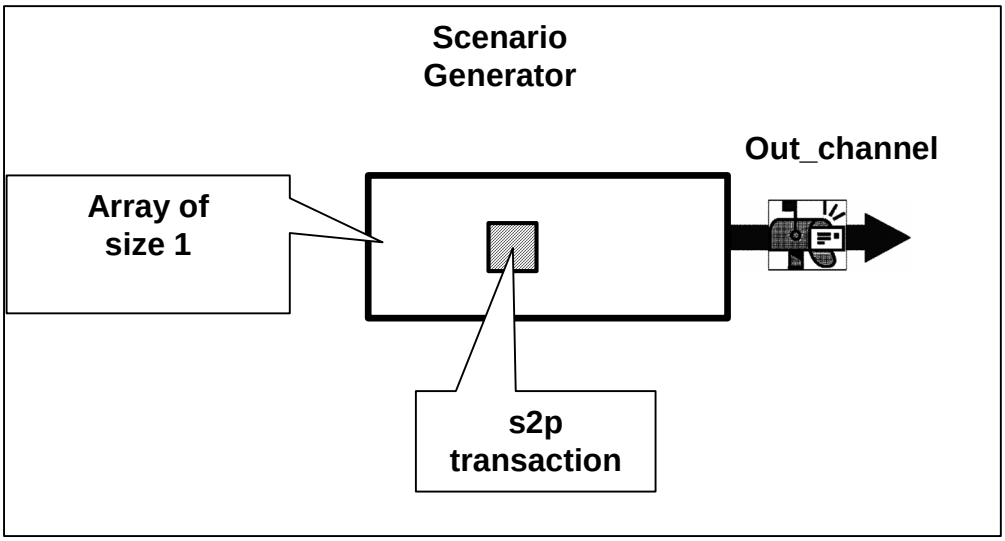

Figure 8.2.2-1 Default Scenario Generator

The *`vmm_scenario_gen* macro creates this basic scenario, and adds it to the set of scenarios. By default, the *scenario_set[0]* of a VMM scenario generator contains an atomic transaction. This will enable the immediate use of the scenario generator, even without defining any specific scenario. However, as you define more complex scenarios, you would progress away from the default scenario and progress to your definition of scenarios. To accomplish this, you simply delete the default scenario from the *scenario_set[$]* queue or override it. Figure 8.2.2-2 shows how to override the default atomic transaction in the S2P example.

```
program automatic s2p_scen_pgm;
S2p_scen_GP_EP s2p_scen_0;
 // ..
  initial begin : b1
    s2p_scen_0 = new();
    s2p_env_0.build();
 //s2p_env_0.s2p_sc_gen_0.scenario_set.push_back(s2p_scen_0);
// Instead of adding to the scenario_set, replace the
// 0th element of this queue.
   s2p_env_0.s2p_sc_gen_0.scenario_set[0] = s2p_scen_0;
    s2p_env_0.run();
  end : b1
endprogram : s2p_scen_pgm
```

Figure 8.2.2-2 Overriding the Default Atomic Transaction in VMM Scenario Generator *(file ch8/s2p_scen_gen.sv)*

Note: Just like the atomic generator, the scenario provides properties provide control over the scenarios. For example, the property *vmm_scenario_gen::stop_after_n_scenarios* will stop the generator after the specified number of scenarios have been generated and entirely consumed by the output channel. The rand *vmm_scenario_gen*::*repeated* defines the number of times the *items* in the scenario are applied. The repeated instances in the scenario count toward the total number of instances generated but only one scenario is considered generated, regardless of the number of times it is repeated.

8.3 CREATING DUMMY SINKS FOR CHANNEL OUTPUTS

A VMM channel has a producer and a consumer. Given the complexity of verification, it is quite possible to have a situation where the producer and consumer are being developed separately, and hence both may not be concurrently available for testing. For example, a generator might be ready prior to the availability of the corresponding command transactor. In such situations, it is desirable (and often necessary) to be able to test the producer alone, with the consumer replaced with an interim model. This can be achieved with a dummy or artificial sink created for a channel and hooked up in the environment until the real consumer becomes available. This would help flush out any design issues with the producer, without having to wait for the integration of all the components. This is also a useful technique to quickly test advanced VMM components, such as a scenario generator, broadcaster, etc. Since the design of these components is complex, they often need a few iterations to get them to behave as per user expectations. Emulating the consumer with a simple model reduces the issues to be resolved in the development of the producer. Having an artificial sink for channels expedites the development of such complex verification components. There are several techniques to achieve this goal, including:

- Producer uses a nonblocking channel
- Use the VMM channel's built-in *vmm_channel::sink()* method.
- Create a quick, dummy consumer, a.k.a an artificial sink transactor

These techniques are addressed in the following subsections.

8.3.1 Using a Nonblocking Channel

With this approach, the producer simply uses a nonblocking completion model, thus not relying on the channels to be sunk by a consumer. This technique allows other items to be added into the channel. This is often very useful in monitor transactors that continuously sample the DUT outputs. Consider the S2P design, where the output data can be available every 8 clocks, assuming an incoming packet length of 8 and a continuous, uninterrupted stream of input data. The parallel data interface monitor, which is a transactor, samples the DUT output data and sends it to higher layers via its output channel. The design has to be tested even if there are no higher layer components available yet. The monitor cannot use the *vmm_channel::put()* because it is a blocking method, and hence will block the monitor from further monitoring the DUT interface until the channel is freed up by a consumer. The use of the *sneak()* methods provides a nonblocking task to insert items into a channel, as demonstrated in Figure 8.3.1

```
class S2p_mon_xactor extends vmm_xactor;
  virtual s2p_if s2p_if_0;
  S2p_xactn allocated_xactn; // Using a factory pattern

  function new(string inst, integer stream_id,
               virtual s2p_if new_s2p_if,
               S2p_xactn_channel out_chan);
   ..
   this.allocated_xactn = new();
   ..
  endfunction : new
```

```
  virtual task mon_dut_if();
    forever begin : w1
       S2p_xactn cur_xactn;
       super.wait_if_stopped();
       wait (s2p_if_0.mon_mp.mon_cb.par_data_valid === 1'b1);
       $cast(cur_xactn, this.allocated_xactn.allocate());

        cur_xactn.pkt_pld = s2p_if_0.mon_mp.mon_cb.par_data;
        // .. Other code not shown
        // vmm_channel::put() is a Blocking method
        // this.out_chan.put(cur_xactn);
        // Instead use, vmm_channel::sneak()
        this.out_chan.sneak(cur_xactn);
    end : w1
  endtask : mon_dut_if
endclass : S2p_mon_xactor
```

Figure 8.3.1-1 Dummy sink using non blocking channel *(file ch8/s2p_mon_xactor.sv)*

VMM recommends the use of *vmm_channel::sneak()* method for such monitors.[2] The *sneak()* is similar to its counterpart *put()* in operation, except that it does NOT block if the channel is full, instead it "sneaks" the new transaction into the channel. A downside of using this approach is that potentially the channels can lead to increased memory usage, particularly in long running simulations.

8.3.2 Using *vmm_channel::sink()* method

To prevent potential accumulation of transactions inside a VMM channel, VMM provides a built-in method, *vmm_channel::sink()* that flushes the content of the channel and sinks any further objects put into it. We used this technique in the design of the FIFO model for the monitor channel that was not used by any component. This code is shown in Figure 8.3.2-1.

```
task Fifo_env:: start();
    super.start();
    this.fifo_xactn_gen_0.start_xactor();
    this.fifo_cmd_xactor_0.start_xactor();
    this.mon_0.start_xactor();
    this.mon_0.fifo_mon_chan_0.sink(); // flush content of channel
endtask : start
```

Figure 8.3.2 Application of the *sink* Method for a Channel

A question that arises is, if a channel is currently being sunk, can I restore it to a normal flow such that it does not sink the future transactions? Yes! A channel that is currently sunk via *vmm_channel::sink()* can be restored to normal flow using the *vmm_channel::flow( )*method. This can be done in the program block at any time. For example, in the program block for a FIFO (Figure 8.3.2-2):

[2] VMM Rule 4-117 Reactive or passive transactors shall use the *vmm_channel::sneak()* method to put transaction descriptors in their output channels.

```
begin : test
   fork : f1 fifo_env_0.run(); join_none : f1
   #10000; // dummy delay
   // Return to normal flow
   fifo_env_0.mon_0.fifo_mon_chan_0.flow();
 end : test
```

Figure 8.3.2-2 Using the *vmm_channel::flow* Method

While this built-in *vmm_channel::sink()* is a fit for some situations to prevent channel build up, it is not very convenient to verify that a producer is working as expected. This is because, the *vmm_channel::sink()* method does not display the transactions that came through the channel, which is a fundamental requirement for verifying the producers. We explore a viable alternative of building an artificial sink transactor in the next section.

8.3.3 Using an artificial sink transactor

A real consumer samples the transactions from the channel with the *vmm_channel::peek()* method, waits for the completion of that transaction, and then uses the *vmm_channel::get()* method to remove the transaction from the channel. An artificial sink mimics the real consumer by retrieving the transaction, but without performing any further processing, thereby speeding up the development process. The only processing that an artificial sink does is to display the transaction that was observed. This artificial sink is modeled as a transactor; hence it can be used in exactly the same way as a real consumer.[3] The sink is instantiated and started in the environment. Figure 8.3.3-1 represents sample code for an artificial sink.

```
class S2p_asink extends vmm_xactor;
  S2p_xactn_channel in_chan;
  function new(string inst, int unsigned stream_id = -1,
               S2p_xactn_channel new_in_chan=null);
    super.new("Artificial Sink", inst, stream_id);
    if (new_in_chan!=null) this.in_chan = new_in_chan;
    else begin : null_chan_guard
     `vmm_note (log, "A NULL channel was passed to a sink");
      this.in_chan=new("s2p_channel", "channel");
     return;
    end : null_chan_guard
  endfunction : new
  extern task artificial_sink(S2p_xactn_channel chan);
  extern task main();
endclass : S2p_asink

task S2p_asink::artificial_sink(S2p_xactn_channel chan);
  S2p_xactn cur_tr;
  forever begin : w_0
    chan.get(cur_tr);
    `vmm_note(this.log,
       cur_tr.psdisplay(.prefix("ART_SINK: Removing a transaction: "));
  end : w_0
endtask : artificial_sink
```

[3] In simple cases, this can be modeled using a fork..join as well.

```
task S2p_asink::main();
  super.main();
  fork
     this.artificial_sink(in_chan);
  join_none
endtask : main
```

Figure 8.3.3-1 Sample code for an artificial sink *(file ch8/s2p_asink.sv)*

We defer the application of this artificial sink transactor to more realistic cases in the next sections.

8.4 VMM SCHEDULER

A scheduler, by definition schedules multiple tasks to a shared resource in a pre-defined order. In the context of verification, scheduling is often a desired mechanism to mimic real-life scenarios. For example:

- A networking device often gets multiple data streams competing to transmit data on a single bus.
- In a processor, the main memory is often sought for access by multiple agents at the same time.
- In a SoC, multiple bus masters seek to control a shared bus at the same time.
- Transaction Level Modeling (TLM) often uses such schedulers to emulate the environment of the system.

There are several ways to create such situations in a simulation environment. One of the simplest mechanism is to spawn off several threads (using fork-join for example), and hope that it will mimic real-life scenarios. Conceptually that works, but in reality, you need a much more controlled mechanism to determine the order of the individual scenarios. The VMM Scheduler is designed to provide users with such a sophisticated mechanism. It accepts multiple sources and schedules them into a single destination. The source and destination of the VMM scheduler are *vmm_channel*s, and the scheduler itself is implemented as a SystemVerilog class named *vmm_scheduler*. From a hardware perspective, a scheduler can be visualized as a Multiplexer, as shown in Figure 8.4-1. Figure 8.4-2 represents the UML for the *vmm_scheduler* class. .

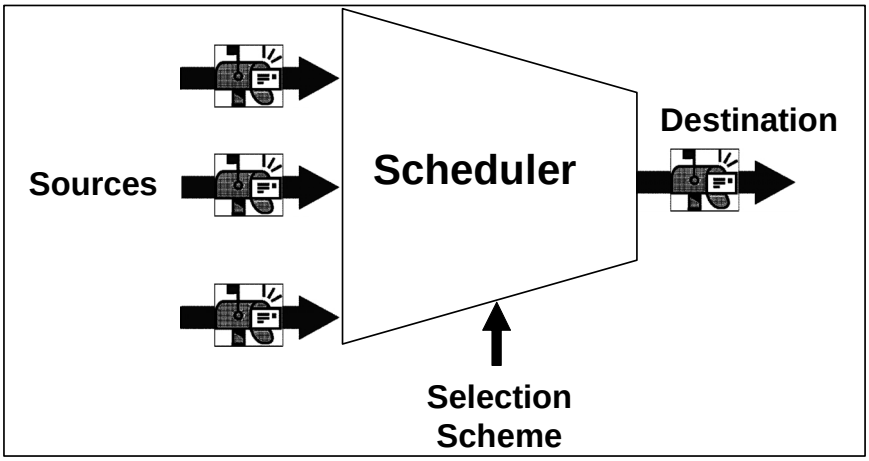

Figure 8.4-1 Scheduler

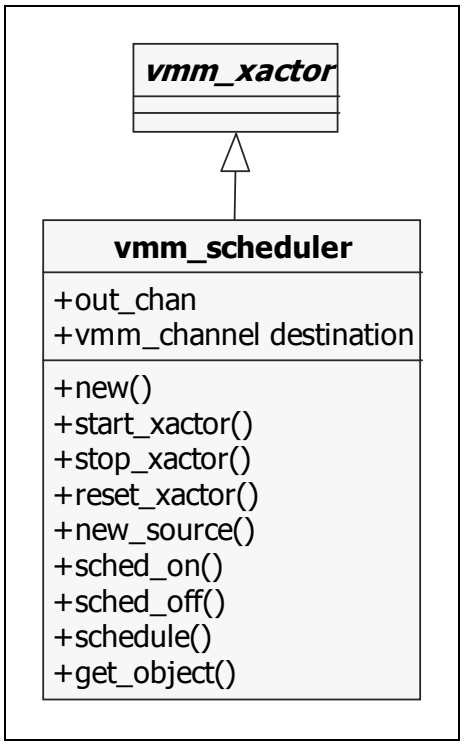

Figure 8.4-2 UML for *vmm_scheduler*

We will use our S2P design as a vehicle to explain a working VMM scheduler. Consider the case of three packet-stream generators; the first stream generator generates packets of length 8 bits, the second generates packets of 8 bit length but with an error (i.e. abort condition), and a third one generates 64 bit packets. The testplan requires that we inter-mix these interesting packets in a round-robin manner. We will use a scheduler and the consumer to this inter-mixed data stream is the *s2p_cmd_xactor* model. Figure 8.4-3 represents the scheduler flow diagram

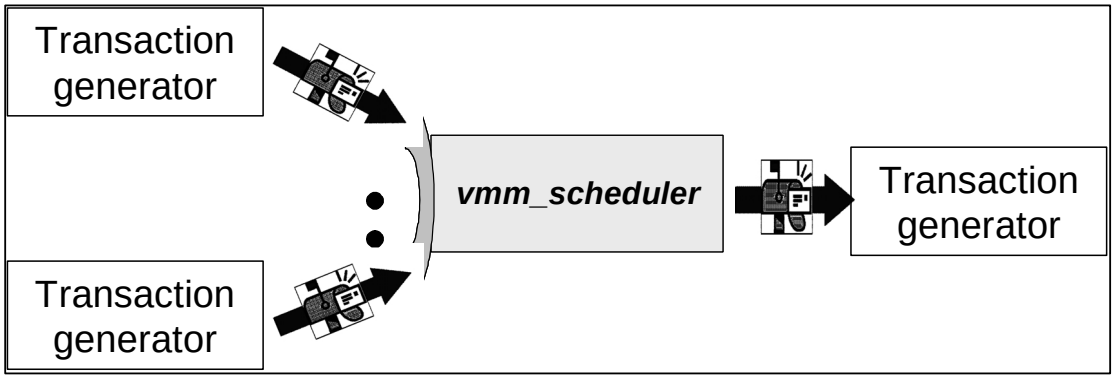

Figure 8.4-3 Scheduler Flow diagram

In the above diagram, the consumer of the *vmm_scheduler* is a command transactor, however for the design of the scheduler alone that is often not required. We will make use of an artificial sink transactor described in Section 8.3.3. Such choices during the development of a verification environment often speed up the process. The basic steps in building this scheduler for the S2P model are as follows:

In the *S2p_env* class, under structural segment (Refer to Chapter 4):

1. Declare a variable of the *vmm_scheduler* class.
```
vmm_scheduler s2p_scheduler;
```
2. Instantiate an array of packet generators. We can use a SystemVerilog dynamic array of *S2p_atmoic_gen* class for better reuse and maintainability.
```
S2p_xactn_atomic_gen gen[];
```
3. Declare a dynamic array of input channels *src_channel []* and a destination channel *dest_channel* of type *S2p_xactn_channel*.
```
S2p_xactn_channel src_chan[];
S2p_xactn_channel dest_chan;
```
4. Declare a local property *S2p_env::num_scheduler_sources* of type *int unsigned*. This property can be updated by the testcase, can be randomly generated etc.
```
int unsigned num_scheduler_sources;
```
5. Declare an artificial sink transactor variable.
```
S2p_asink s2p_asink_0;
```

In **S2p_env class, under the test flow segment** (Refer to Chapter 4):

6. In the *S2p_env::build()* task, allocate the *s2p_scheduler* object.
```
s2p_scheduler = new(
 "S2P Scheduler", // name
  "0",            // inst
  dest_chan);     // Destination
```
7. Allocate the array of source channel and the generator in build phase based on the local property *S2p_env::num_scheduler_sources*. SystemVerilog provides the convenient *foreach* construct.
```
src_chan = new[this.num_scheduler_sources];
gen = new[this.num_scheduler_sources];
foreach (src_chan [id]) begin : fe_l_0
    s2p_chan[id] = new ("Src Channel", $psprintf("%d",id));
end : fe_l_0

foreach (gen [id]) begin : fe_l_1
    gen[id] = new("S2P_Gen", id, src_chan[id]);
end : fe_l_1
```
8. Allocate the dest_channel.
```
dest_channel = new ("Sched Dest Channel", "0");
```
9. Allocate the artificial sink transactor. Pass the *dest_channel* as argument to this artificial sink.
```
s2p_asink_0 = new("ART_SINK", dest_channel);
```
10. Start the scheduler and the artificial sink and the atomic generators in task *start()*.
```
this.s2p_scheduler.start_xactor();
  foreach (gen [id]) begin : fe_l_2
    this.gen[id].start_xactor();
  end : fe_l_2
this.s2p_sink_0.start_xactor();
```

In the testcase (program):

11. Choose the number of inputs to the scheduler. Remember that in Step 4 (above) we have declared a variable to hold this value that needs to be updated before the *S2p_env::build()* phase. So in an initial block in the program block we execute the flow until the *S2p_env::gen_cfg()* configuration phase, update this variable, and then continue the control flow up to the *build()* step. We used a parameter to select this value.

```
parameter NUM_SOURCES = 3;
s2p_env_0. num_scheduler_sources = NUM_SOURCES;
s2p_env_0.build();
```

12. Add the needed sources to the scheduler. VMM provides a method called *vmm_scheduler::new_source()* for this process. This is done after the *build()* phase. A call to *S2p_env::run()* takes care of the rest of the test flow.

```
foreach (s2p_env_0.src_chan [id]) begin : fe_l_1
    s2p_env_0.s2p_scheduler.new_source(src_chan[id]);
end : fe_l_1

s2p_env_0.run();
```

Figures 8.4-5 shows the environment for the S2P Scheduler. The actual steps are numbered in the same order as the description above.

```
class S2p_env extends vmm_env;
  // Structural segment
  // #1 declare a scheduler variable
  vmm_scheduler s2p_scheduler;

  // #2 Dynamic array of generators
  S2p_xactn_atomic_gen gen[];

  // #3 Dynamic Array of Channels
  S2p_xactn_channel src_chan[];
  S2p_xactn_channel dest_chan;

  // #4 how many sources?
  int unsigned num_scheduler_sources;

  // #5 Artificial sink
  S2p_asink s2p_asink_0;

  extern function void build();
  extern task start();
endclass : S2p_env
```

```
// Test flow segment
function void S2p_env::build();
  super.build();
  // #6 allocate scheduler
  s2p_scheduler = new("S2P Scheduler", "0", dest_chan);

  // #7 allocate array of src channel and generator
  src_chan = new[this.num_scheduler_sources];
  gen = new[this.num_scheduler_sources];

  foreach (src_chan [id]) begin : fe_l_0
    s2p_chan[id] = new ("Src Channel", $psprintf("%d",id));
  end : fe_l_0
  foreach (gen [id]) begin : fe_l_1
    gen[id] = new("S2P_Gen", id, src_chan[id]);
  end : fe_l_1

  // #8 allocate dest_channel
  dest_channel = new ("Sched Dest Channel", "0");

  // #9 allocate artificial sink
  s2p_asink_0 = new ("ART_SINK", dest_channel);
endfunction : build

task S2p_env::start();
  fork
    super.start();
  join_none
  // #10 start all transactors
  this.s2p_scheduler.start_xactor();
  foreach (gen [id]) begin : fe_l_2
    this.gen[id].start_xactor();
  end : fe_l_2
  this.s2p_sink_0.start_xactor();
endtask : start
```

Figure 8.4-5 S2P Scheduler environment
(file ch8/s2p_sched_env.sv)

Figure 8.4-6 shows a testcase for the S2P Scheduler.

```
program automatic s2p_sched_pgm;
  parameter NUM_SOURCES = 3;
  S2p_env s2p_env_0;

  initial begin : test
    s2p_env_0 = new();
    s2p_env_0.gen_cfg();

   // #11 Choose number of inputs to be scheduled
    s2p_env_0.num_scheduler_sources = NUM_SOURCES;
    s2p_env_0.build();


   // #12 Add the number of sources as inputs to the scheduler
    foreach (s2p_env_0.src_chan [id]) begin : fe_l_1
      s2p_env_0.s2p_scheduler.new_source(src_chan[id]);
    end : fe_l_1

    s2p_env_0.run();
  end : test
endprogram : s2p_sched_pgm
```

Figure 8.4-6 S2P Scheduler testcase *(file ch8/s2p_sched_pgm.sv)*

A sample run of this scheduler code shows the actual scheduling of the transactions in the log file as shown in Figure 8.4-7. The transaction ID is marked bold to emphasize that the three streams are mixed in a round robin manner.

```
[Normal:NOTE]|ASINK: #0.0.0 s2p ERR  Pkt, len 191, data b6a859d4
[Normal:NOTE]|ASINK: #0.1.0 s2p ERR  Pkt, len 244, data 52531565
[Normal:NOTE]|ASINK: #0.2.0 s2p GOOD Pkt, len 143, data be409a8d
[Normal:NOTE]|ASINK: #0.0.1 s2p GOOD Pkt, len 212, data 1e52fc5a
[Normal:NOTE]|ASINK: #0.1.1 s2p GOOD Pkt, len 86,  data 20e9a7ea
[Normal:NOTE]|ASINK: #0.2.1 s2p GOOD Pkt, len 223, data e5a14c07
```

Figure 8.4-7 Log file showing the scheduling of different transaction streams

NOTES:

- The addition of new sources to the scheduler is dynamic. A new source can be added after some simulation cycles, if need be.
- The scheduling mechanism by default is round-robin. If you need purely random scheduling, you can turn off the constraint block inside the *vmm_scheduler* class.

8.5 VMM BROADCAST

Channels are point-to-point data transfer mechanisms. But there are situations where the channel needs to be broadcasted to multiple transactors. For example, consider an Ethernet switching device with eight ports. Ethernet packets can be of different variants, as there are several fields that determine the nature of the packet. A generic verification environment for such a design will have an individual generator for each port. Each generator will generate packets and send them to individual ports. However there are situations where the verification plan requires that all ports send exactly the same kind of packets, such as: L2 type, 64-byte packets with no CRC error. For this kind of testing it is often desirable to have a single generator and broadcast the traffic to all

ports. With such a broadcaster, one could easily control the packet generation from one central location and have it broadcasted to all ports. With individual generator for each port, it will be hard to make sure that every port sends the same packet.

From a hardware perspective, a broadcaster can be visualized as a De-Multiplexer as shown in Figure 8.5-1.

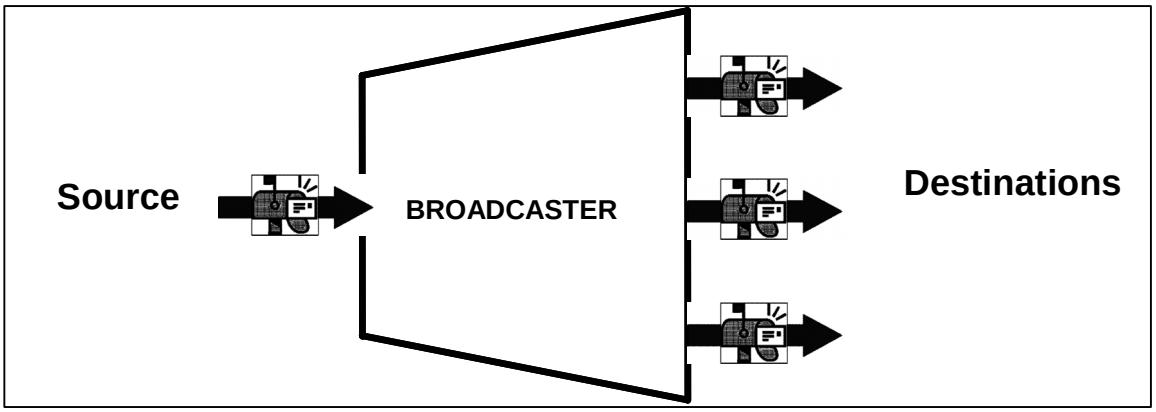

Figure 8.5-1 A hardware perspective of *vmm_broadcast*

To solve this potential requirement for a broadcast, VMM provides the *vmm_broadcast* class, supported by many methods and properties to tailor its operation to the user's needs. Figure 8.5-2 represents a UML view of this class.

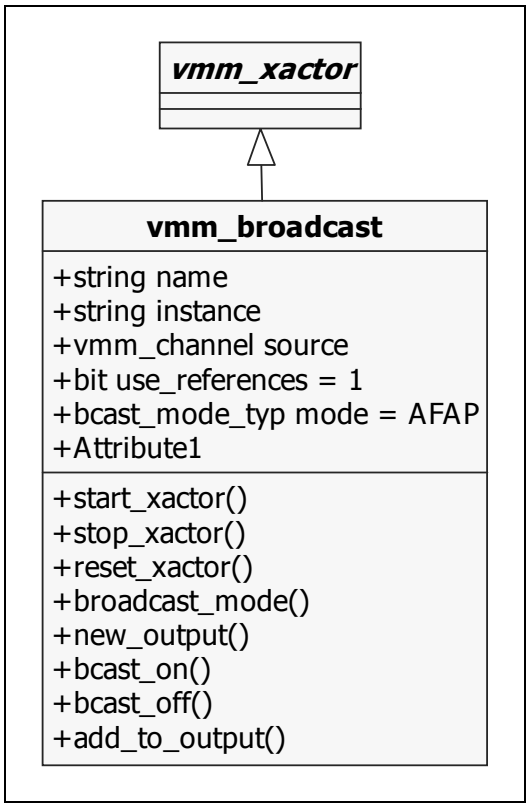

Figure 8.5-2 UML View of vmm_broadcast

We will use an Ethernet device example to demonstrate a working VMM Broadcaster code. We will use the example requirement of having a single generator feeding up to eight different Ethernet ports, as shown in Figure 8.5-3.

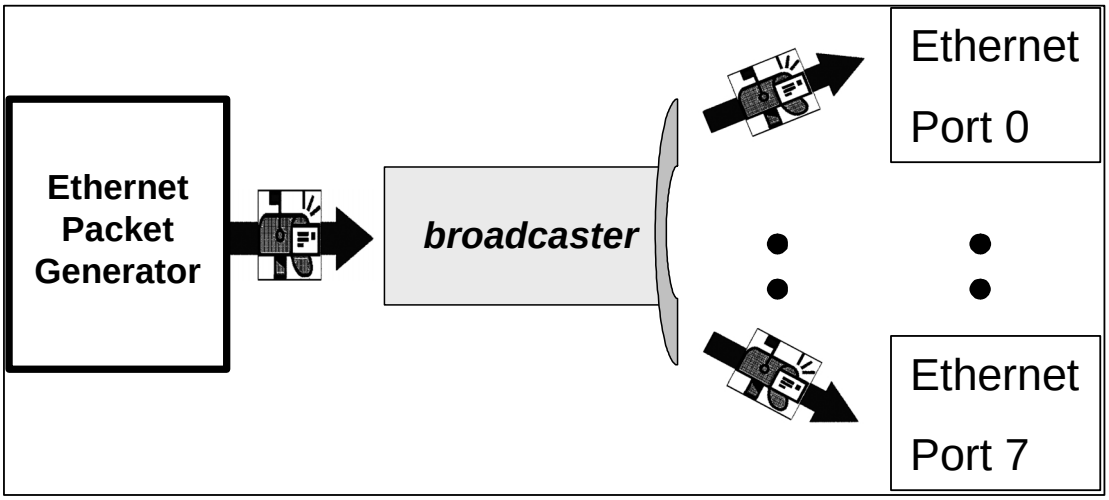

Figure 8.5-3 Ethernet Broadcaster

As with the scheduler example, we will use artificial sinks in place of the actual Ethernet ports to demonstrate the broadcast operation. Following are the detailed steps to design a VMM broadcaster for this example.

In Eth_env class, under the structural segment (Refer to Chapter 4):

1. Declare a variable of the *vmm_broadcast* class.

   ```
   vmm_broadcast eth_broadcaster;
   ```

2. Instantiate an Ethernet packet generator. We use an atomic generator.

   ```
   Eth_xactn_atmoic_gen gen_0;
   ```

3. Declare a source channel eth_src_channel and a dynamic array of output channels *eth_dest_channel_array []* of type *Eth_xactn_channel*.

   ```
   Eth_xactn_channel eth_src_chan;
   Eth_xactn_channel eth_dest_channel_array [];
   ```

4. Declare a local property *Eth_env::num_bcast_dest* of type *int unsigned*. This property can be updated by the testcase, can be randomly generated, etc.

   ```
   int unsigned num_bcast_dest;
   ```

5. Declare an array of artificial sink transactor.

   ```
   Eth_asink eth_asink_array[];
   ```

In Eth_env class, under the test flow segment (Refer to Chapter 4):

6. In the *Eth_env::build()* task, allocate the *eth_broadcaster* object.

```
eth_broadcaster = new(
    "Ethernet Broadcaster", // name
     "0",             // instance
     eth_src_chan);     // Source
```

7. Allocate the generator and array of destination channel in build phase based on the local property *Eth_env::num_bcast_dest*. SystemVerilog provides the convenient *foreach* construct.

```
eth_dest_channel_array = new[this.num_bcast_dest];
foreach (eth_dest_channel_array [id]) begin
  eth_dest_channel_array[id] = new ("Eth_Dest_Chan",
                                    $psprintf("%0d", id));
end
```

8. Allocate the *eth_src_channel.*

```
eth_src_chan = new ("Ethernet Source Channel", "0");
```

9. Allocate the array of artificial sink transactors. Pass the *dest_channel_arrary[id]* as argument to this artificial sink.

```
eth_asink_array = new[this.num_bcast_dest];

foreach (eth_asink_array [id]) begin
  eth_asink_array[id]=new("ART_SINK", dest_channel_array[id]);
end
```

10. Start all the transactor components such as: generator, broadcaster and array of artificial sink in the *Eth_env::start()* task.

```
this.eth_broadcaster.start_xactor();

  this.gen_0.start_xactor();
  foreach (eth_asink_array [id]) begin : fe_l_2
    this.eth_asink_array[id].start_xactor();
  end : fe_l_2
```

In the testcase (program):

11. Choose the number of outputs of the broadcaster. Remember that in Step 4 (above) we have declared a variable to hold this number, its value needs to be updated before the *Eth_env::build()* phase. So in an initial block inside the program, we run the flow until *Eth_env::gen_cfg()* phase, update this variable and then call the *build()*. We used a parameter to select this number.

```
eth_env_0.gen_cfg();
eth_env_0.eht_broadcaster.num_bcast_dest = NUM_DEST;
```

12. Add the destinations to the broadcaster. VMM provides a method *vmm_broadcast::new_output()* for this process. This is done after the *build()* phase. A call to *Eth_env::run()* takes care of the rest of the test flow.

```
program test_pgm;

// ..
initial begin : test
  eth_env_0.build();
  foreach (eth_env_0.eth_dest_channel_array [id])
   begin : fe_l_1
     output_id[id] = eth_env_0.eth_broadcaster.new_output(
                        eth_dest_channel_array[id]);
    end : fe_l_1

  eth_env_0.run();
end : test
endprogram : test_pgm
```

Figures 8.5-4 shows the environment for the Ethernet Broadcaster. The actual steps are numbered in the same order as the description above.

```
class Eth_env extends vmm_env;
  // Structural segment
  // #1 declare a broadcast variable
  vmm_broadcast eth_broadcaster;

  // #2 Ethernet Atomic generator
  Eth_xactn_atomic_gen gen_0;

  // #3 Dynamic Array of Channels
  Eth_xactn_channel eth_src_chan;
  Eth_xactn_channel eth_dest_channel_array [];

  // #4 how many destinations?
  int unsigned num_bcast_dest;

  // #5 Array of Artificial sink
  Eth_asink eth_asink_array [];
   //  remaining steps in the environment
  extern function void build();
  extern task start();
endclass : Eth_env

// Test flow segment
function void Eth_env::build();
  super.build();
  // #6 allocate broadcaster
  eth_broadcaster = new("Ethernet broadcaster", "0",
                        eth_src_chan);

  // #7 allocate generator and array of destination channel
  eth_dest_channel_array = new[this.num_bcast_dest];
  foreach (eth_dest_channel_array [id]) begin : fe_l_0
    eth_dest_channel_array[id] = new ("Eth_Dest_Chan",
                                      $psprintf("%0d", id));
  end : fe_l_0

  // #8 allocate eth_src_channel
  eth_src_chan = new ("Ethernet Source Channel", "0");

  // #9 allocate array of artificial sink
  eth_asink_array = new[this.num_bcast_dest];
  foreach (eth_asink_array [id]) begin : fe_l_1
    eth_asink_array[id] = new ("ART_SINK",
                               eth_dest_channel_array[id]);
  end : fe_l_1
endfunction : build
```

```
task S2p_env::start();
  fork
    super.start();
  join_none
  // #10 start all transactors
  this.eth_broadcaster.start_xactor();
  this.gen_0.start_xactor();
  foreach (eth_asink_array [id]) begin : fe_l_2
    this.eth_asink_array[id].start_xactor();
  end : fe_l_2
endtask : start
```

Figure 8.5-4 Ethernet Broadcaster Environment (*file ch8/eth_broadcast_env.sv)*

Figure 8.5-5 shows a testcase for the Ethernet Broadcaster.

```
program automatic eth_bcast_pgm;
  parameter NUM_DEST = 8;
  Eth_env eth_env_0;
  int output_id[NUM_DEST];

  initial begin : test
    eth_env_0 = new();
   // #11 run the flow till gen_cfg and
   // Choose number of destinations to be broadcasted

    eth_env_0.gen_cfg();
    eth_env_0.eht_broadcaster.num_bcast_dest = NUM_DEST;

   // #12 Run the flow till build() and
   // add the number of destinations as outputs
   // to the broadcaster
    eth_env_0.build();

    foreach (eth_env_0.eth_dest_channel_array [id])
    begin : fe_l_1
     output_id[id] = eth_env_0.eth_broadcaster.new_output(
                       eth_dest_channel_array[id]);
    end : fe_l_1

    eth_env_0.run();
  end : test
endprogram : eth_bcast_pgm
```

Figure 8.5-5 Ethernet Broadcaster testcase *(file ch8/eth_broadcast_pgm.sv)*

A sample run of this broadcaster code shows that the same packet gets broadcasted to all the consumers as shown in Figure 8.5-6. The transaction ID is marked bold to highlight that the same transaction is received by all consumers.

```
              Etnet Port   Transaction Type   Length      Data
              <--------->  <---->     <---->   <------>  <---------->
[Normal:NOTE]|ASINK ID 0: #0.0.0 Eth ERR Pkt, len 210, data b6a859d4
[Normal:NOTE]|ASINK ID 1: #0.0.0 Eth ERR Pkt, len 210, data b6a859d4
[Normal:NOTE]|ASINK ID 2: #0.0.0 Eth ERR Pkt, len 210, data b6a859d4
[Normal:NOTE]|ASINK ID 0: #0.0.1 Eth OK  Pkt, len 64, data ab098722
[Normal:NOTE]|ASINK ID 1: #0.0.1 Eth OK  Pkt, len 64, data ab098722
[Normal:NOTE]|ASINK ID 2: #0.0.1 Eth OK  Pkt, len 64, data ab098722
[Normal:NOTE]|ASINK ID 0: #0.0.2 Eth OK  Pkt, len 20, data 809011ab
[Normal:NOTE]|ASINK ID 1: #0.0.2 Eth OK  Pkt, len 20, data 809011ab
[Normal:NOTE]|ASINK ID 2: #0.0.2 Eth OK  Pkt, len 20, data 809011ab
```

Figure 8.5-6 Log file showing the broadcast operation of same transaction

NOTES:

- The broadcaster can be configured to operate in 2 modes – ALAP (As Late As Possible) and AFAP (As Fast As Possible).[4]
- The broadcaster by default sends only handles of the source transaction to the output channels. This is a very efficient way as a single copy is maintained and all consumers use references to this transaction. If you need individual copies, set the *use_references* argument to the *vmm_broadcast::new()* to 0. Such a control is also possible on an individual output basis by passing the *use_references* bit to the *vmm_broadcast::new_output()* method.
- The scheduling mechanism by default is round-robin. If you need a purely random scheduling, you can turn off the constraint block inside the *vmm_scheduler* class. The *vmm_scheduler_election* is a class that implements the random election rules for the next scheduling cycle. *vmm_scheduler::randomized_sched* is an instance of *vmm_scheduler_election class*. This *vmm_scheduler_election* class has a constraint named "default_round_robin" that can be set to *constraint_mode(0).* Thus, in the *initial* block of the program:

```
// Setting the scheduling to random
my_env_0.my_scheduler.randomized_sched.default_round_robin.
constraint_mode(0);
```

8.6 VMM LOG

A typical verification environment produces significant amount of log messages to inform the users about the progress and activities during a simulation run. Often the messages are added by newcomers and a consistent style in printing the information may not be followed. A consistent style of message logging helps in extracting the needed information from a log file. The requirements for a comprehensive log message handler can be surprisingly much larger than what it initially conceived. More often than not, every design team thinks about such huge list of requirements, but given the amount of time it would take to create such an infrastructure and the effort to maintain it, this activity is put in the back burner and never gets done. A significant advantage of a framework such as VMM is that it already has a built in library to handle most of these requirements (and much more). The base class *vmm_log* is the VMM's message service that offers a very flexible and convenient way to handle the messages in a simulation environment. Some of the key features of *vmm_log* are:

[4] The VMM pp 399

- Easy to use, pre-built macros to hide the complexity and to present users with a syntax resembling Verilog's *$display*.
- Consistency in message format across the entire project.
- Customizable message format capability.
- Promotion and demotion capabilities of the message severities, - to turn errors to warnings for example.
- Tracking of the number of errors in the system. This allows users to react to a pre-specified error count in order to avoid running a simulation beyond a certain number of errors (Such a run with 1000 errors is often meaningless; a threshold of 10 is often a good choice to stop the simulation).
- Determines the PASS/FAIL condition of a simulation run based on error counts.

8.7 CUSTOMIZING VMM MESSAGE OUTPUT FORMAT

Almost every design verification team has its own style preferences in extracting information out of the simulator including log data, coverage analysis flow, etc. A generic methodology, such as VMM, needs to cater to such a wide audience to be acceptable. One of the fundamental tools for a verification engineer is the simulation log file. In the authors' experience, a good 40% of a verification engineer's time is spent in analyzing the log file(s). This means that the format of log file that displays the messages needs to be highly customizable.
Typically users extract key information from log files via PERL, UNIX-Shell script etc. Thus, while adopting or migrating to VMM, teams may want to reuse their existing scripts and have the log files produced in the format that their scripts expect. For example, consider a sample VMM code: `vmm_error(this.log, "Sample Error");
The default format of vmm_log is spread across two lines in the output as shown Figure 8.7-2.

```
!ERROR![FAILURE] on Pgm_Logger() at                    1950:
    Sample Error!
```

Figure 8.7-2 Sample vmm_log with Default Format

While this 2-line format may be seen as useful by some, other design teams prefer to have every output message in one single line. That eases post-processing of log files via commands such as UNIX *grep* etc. VMM allows the entire message to be customizable. To be able to customize the message format, you need to overload the virtual function *format_msg* in the *vmm_log_format* class. The prototype for the *format_msg* function is shown in Figure 8.7-3.

```
virtual function string format_msg( string name,
                                    string inst,
                                    string msg_type,
                                    string severity,
                                    ref string lines[$]);
```

Figure 8.7-3 Prototype for the *format_msg* Function

There are five arguments to this function; each one is briefly explained from a user stand point below.

name & inst: Identifies the message service agent that issued this message, and is useful to quickly locate from which file/component this message originated from.

msg_type: Message Type can be FAILURE, NOTE, DEBUG, TIMING, XHANDLING, TRANSACTION, COMMAND, REPORT, PROTOCOL, CYCLE.

severity: Indicates the seriousness of the message, such as FATAL, ERROR, WARNING etc.

lines[$] : A SystemVerilog queue of strings containing the actual message.

Given that all individual pieces of the message are available as different arguments, it is simple procedural code to customize the message. To have the output formatted in single line such as "`1950.00 ns [*ERROR*:FAILURE] | Sample Error`",
you can use the *vmm_log_format::format_msg* function as shown in Figure 8.7-4.

```
class s2p_log_fmt extends vmm_log_format;
 virtual function string format_msg( string name,
                                     string instance,
                                     string msg_type,
                                     string severity,
                                     ref string lines[$]);

      foreach (lines [ l ] )
        $sformat(format_msg, "%0t %s [%0s:%0s] | %s",
             $time, name, severity, msg_type, lines[l]);

    endfunction : format_msg
endclass : s2p_log_fmt
```

Figure 8.7-4. S2P log format *(file ch8/s2p_log_fmt.sv)*

A single derived class can alter the message format of all *vmm_log* instances in the environment as the VMM Log is designed to cater such a versatile requirement; this is very convenient for design teams to customize all log messages from a single place. The new derived class *s2p_log_fmt* needs to be used in the environment. A sample usage is shown in Figure 8.7-5.

```
`include "s2p_log_fmt.sv"
class S2p_env extends vmm_env;
  S2p_log_fmt log_fmt_cntl;

  function new(virtual s2p_if.ser_drv_mp new_vif
               );
    $timeformat(-9,2, " ns");
    this.log_fmt_cntl = new();
    this.log = new("S2P Env Logger", "0");
    this.log.set_format(this.log_fmt_cntl);
  endfunction : new
endclass : S2p_env
```

Figure 8.7-5 S2P log format usage in the environment *(file ch8/s2p_env.sv)*

The important lines of code in Figure 8.7-5 are shown in bold font. Essentially, the base class *vmm_log* provides a function *set_format()* that takes this *vmm_log_format* class object as an argument and makes the change effective.

Guideline: Every project team should agree on a common log messaging format to use throughout the project, and implement the same.[5]

8.8 FUNCTIONAL COVERAGE

In this section we present how to arrive at coverage points for a design, using FIFO and S2P as design examples. We will then demonstrate how to integrate a coverage model to an existing VMM verification environment. Later on we show how to use a good constrained-random environment to target coverage holes using different seeds.

8.8.1 Extracting coverage points

One of the first steps in adopting functional coverage is answering a key question of "What to cover"? Such a question is often not relevant to its code coverage counterpart, as code coverage covers all of what is coded in the design. Functional coverage requires explicit definitions of what needs to be covered. This is often referred to as the "Coverage Plan", and represents an evolving technology. When this technology is adopted well, it has proven to be very effective in many leading edge designs. From an overall perspective, formalizing the answer to "What to cover" is a good thing as users expect a good ROI (Return On Investment) in adopting a new technology. A detailed Coverage Plan is critical to a fruitful functional coverage adoption.

While the exact details on a comprehensive coverage plan are beyond the scope of this book, we will show some of the coverage plan items as relevant to the example designs being used: the FIFO and S2P. Before we list the coverage points, it is useful to understand the characteristics of coverage plan. Specifically:

- A coverage plan should list important features of the design.
- A coverage plan should be of sufficient details so that it can be easily used to capture the coverage points using SystemVerilog construct such as *covergroup* and *cover property*. For example, stating that the FIFO should work in all configurations is too abstract a statement. Instead breaking that statement into the interesting threshold values for the configuration registers is more practical.
- An item in a coverage plan should convey valuable information other than simply many uncorrelated data points.

The coverage plan documentation is thereafter captured in SystemVerilog. The choice of constructs is situation dependent - data coverage is better captured using the *covergroup*, while temporal coverage such as bus interfaces, latency etc. are easier to capture using the *cover property*.

Interesting coverage points for the FIFO model include:

- Fifo Full
- Fifo Empty
- Fifo Full and a Push
- Fifo Full and a simultaneous occurrence of Push and Pop
- Fifo Empty and a Pop
- Fifo Empty and a simultaneous occurrence of Pop and Push

[5] The VMM page 134. To ensure a consistent look and feel to the messages issued from different sources, a common message service should be used.

- Fifo threshold registers: boundary values of almost_empty_threshold and almost_full_threshold.
- Fifo threshold value in between minimum and maximum.
- Fifo almost_full / Fifo almost_empty followed by reset

Interesting coverage points for the S2P design include:
Basic coverage points:

- Various packet lengths: 8, 16, 24, 32,
- Odd packet lengths - not multiple of 8 (to test the padding logic in design).
- Packet abort during start of packet (*sop)*
- Packet abort during end of packet (*eop)*
- Packet aborts occurring in between *sop* & *eop*.

Cross coverage across basic points:

- Cross of packet abort and packet length.
- Sequence coverage - abort followed by good packet.
- Good packet followed by abort packets and then good packets.

8.8.2 Integrating coverage models into the environment

Functional coverage integration into a VMM environment is often done through callbacks as it provides an easy entry point for the coverage model.[6] In the FIFO model we added two callback methods inside the command transactor that enabled us to cover the basic coverage model. The coverage model is build as a separate class as shown in Figure 8.8.2-1. Note that this directly stems from the coverage plan. The code snippet in the figure shows the implementation of the coverage item: "Occurrence of Full, Push and Pop".

```
class Fifo_fcov_plug_in extends Fifo_fcov_cb;

  Fifo_cov_model fifo_cov_model_0;
  virtual fifo_if.fifo_mon_if_mp vif;

  covergroup full_cg;
    full_cpt : coverpoint fifo_cov_model_0.full;
    push_cpt : coverpoint fifo_cov_model_0.push;
    pop_cpt  : coverpoint fifo_cov_model_0.pop;

    pp_during_full_cr : cross full_cpt,push_cpt,pop_cpt;

  endgroup : full_cg
endclass : Fifo_fcov_plug_in
```

Figure 8.8.2-1 Façade for FIFO Coverage Callback
(file ch8/fifo_cov_model.sv)

The callback implementation is shown in file *ch8/fifo_fcov_cb_defs.sv*.

[6] Integration of coverage model can also be done using *vmm_notify* and/or peeking to an existing transaction channel via *vmm_channel::tee()* method. Depending on the problem at hand one approach is easier than the other.

8.9 SEEDING THE RANDOMIZATION

One of the key motivations of a constrained random testing is to be able to create a constrained random testcase that produces different scenarios when simulated with different seeds. By adding randomness and simulating the same testcase with several different seeds, the same test should be able to hit multiple coverage points, effectively replacing multiple directed testbenches. For example, in the S2P simulation model with the coverage points as identified in the previous section, the entire packet lengths can be covered by a single test when the simulation is run with various seeds. Similarly, in the FIFO design, the complex coverage goal requirement for the generation of PUSH and POP when the FIFO is full is achieved by simple random tests executed with several seeds.

The randomization seed can be changed using SystemVerilog constructs. Some tools might also provide other ways to change the randomization seed. To modify the seed in the Synopsys VCS simulator, use the run time switch: `+ntb_random_seed=<number>` to transfer the initial seed to the simulator.

8.10 CHANGING ERROR SEVERITY DYNAMICALLY

One of the common requirements in any practical verification setup is the ability to change the severity level of certain groups of messages on the fly during a simulation run. For example, while a certain portion of the design is undergoing major change in the RTL code, the verification environment might start flagging failures for the correct reasons that the RTL is misbehaving. But given that this specific portion of the RTL code is undergoing changes, you may want to temporarily lower the severity of these errors to WARNING. Another good example is the concept of negative tests - tests that are inducing invalid/error scenarios to make sure that the system can safely handle such erroneous operations. One of the problems with such negative tests in a simulation run is that the verification environment flags these errors (assertion failures[7], data integrity, missing packets etc.). One option for the users is to carefully analyze each and every error occurrence and guarantee that these are indeed false failures. The problem gets compounded when a negative test scenario is followed by a positive test scenario in a single simulation run; in such cases the errors are false errors only during the negative testing period and any error occurring during positive testing period is a potential design bug. A recommended approach would be to be able to alter the severity of these errors (in general any messages) on the fly within the test itself. In that case, the test developer knows exactly when to turn an error to a warning and vice versa.

VMM provides this flexibility via its messaging service, *vmm_log* with a promotion and demotion of messages. The function *modify()* allows the modification of the severity of a specific (or all) messages issued through it. The prototype of the function *vmm_log::modify()* is shown in Figure 8.10-1.

[7] Provided assertions use a fail action block and the action block uses a *vmm_log* to emit the error messages. For a sample, users are referred to the SVA Checker Library shipped along with VCS, $VCS_HOME/packages/sva/sva_std_task.h

```
virtual function int modify
  (string name = "",
   string inst = "",
   bit recursive = 0,
   int typs = ALL_TYPS,
   int severity = ALL_SEVS,
   string text = "/./",
   int new_typ = UNCHANGED,
   int new_severity = UNCHANGED,
   int handling = UNCHANGED);
```

Figure 8.10-1 Prototype *vmm_log:: modify*

The usage of this function is quite simple. For example, Figure 8.10-2 demonstrates the demotion of all errors.

```
class S2p_env extends vmm_env;
  // ..
  int log_change_id;

  function demote_bfm_errors;
    this.log_change_id = this.s2p_cmd_xactor_0.log.modify
     (.name("/./"),
      .inst("/./"),
      .new_severity(vmm_log::WARNING_SEV));

  endfunction : demote_bfm_errors
endclass : S2p_env
```

Figure 8.10-2 Usage of *vmm_log::modify* for demotion (*file ch8/pro_demo.sv)*

There are two important steps in this process:

- Identify the log instance that requires the promotion/demotion
- Change the severity level

The second step is straight forward - specify what the new severity is (FATAL_SEV, ERROR_SEV, WARNING_SEV, NORMAL_SEV, TRACE_SEV, DEBUG_SEV, VERBOSE_SEV, DEFAULT_SEV, ALL_SEVS.). The first step is where VMM provides a great deal of flexibility to the users. You can "identify" the message to be altered via a number of ways:

- By specific log agent (shown in Figure 8.10-2 above).
- By specifying the name and/or instance of the log agent
- By a specific text message being printed.

Such selection is string based, and VMM allows an *awk* programming style regular expression in choosing them. For example, to alter the severity of all messages that have a key word "data integrity", one can use the code shown in figure 8.10-3.

```
class S2p_env extends vmm_env;
  // ..
  int log_change_id;

  function demote_data_integrity_errors;
    this.log_change_id = this.log.modify
     (.name("/./"),
      .inst("/./"),
      .text("/data integrity/"),
      .severity(vmm_log::ERROR_SEV)
      .new_severity(vmm_log::WARNING_SEV));

  endfunction : demote_data_integrity_errors
endclass : S2p_env
```

Figure 8.10-3 Selection by message text. ***(file ch8/pro_demo.sv)***

8.11 FILE STRUCTURE

Table 8.11 demonstrates the file Structure and the purpose of each file.

Table 8.11 File Structure and Functions

/ch8/ch8_fifo_fcov directory: FIFO Coverage through Callback

File	Function	Used by
fifo_pkg.sv	Defines types and initialized variables	ALL
fifo_if.sv	Defines the FIFO interface	RTL and by program, testbench, transaction and transactors
fifo_csr_if.sv	Defines the FIFO configuration interface	RTL, property models, and by environment, and possibly transactors
fifo_xactn.sv	Defines the transaction class with the constraints Also used for the channel generation with: *`vmm_channel* (Fifo_xactn)	*`vmm_channel* macro for generation of channel, *`vmm_atomic_gen* macro for generation of atomic generator, monitor transactor for creation of transaction from observed values on bus interface.
fifo_rtl.sv	Represents the FIFO RTL DUT	Top level
fifo_props.sv	Defines the properties for assertions	Top level for bind
fifo_log_fmt.sv	Defines formatting information for display	FIFO environment
fifo_pgm.sv	Creates the modeling for simulation and initiates the *run* in the environment	Top level
fifo_env.sv	Creates the build and start for simulation	program
fifo_mon_xactor.sv	Creates a copy of the observed transaction onto a transaction channel	Scoreboard, top level
top_tb.sv	Represents the top level and instantiates the RTL, the bind, the monitor, etc	none
fifo_gen_xactor.sv	Uses the macro *`vmm_atomic_gen* for generation of atomic generator, defines the constraints for the number of transactions	Environment for creation of the build model,
inject_err.sv	Error injection classes	Command transactor
fifo_fcov_model.sv	Functional Coverage model for FIFO	Environment
fifo_fcov_cb_defs.sv	Callback definitions for the FIFO model	Command transactor

/ch8/ch8_s2p_scen_gen directory: Scenario Generator

File	Function	Used by
s2p_if.sv	Defines the Serial-to-Parallel interface	RTL and by program, testbench, transaction and transactors
s2p_xactn.sv	Defines the transaction class with the constraints Also used for the channel generation with: *`vmm_channel* (S2p_xactn)	*`vmm_channel* macro for generation of channel, *`vmm_atomic_gen* macro for generation of atomic generator, monitor transactor for creation of transaction from observed values on bus interface
s2p_rtl.sv	Represents the S2P RTL DUT	Top level
s2p_log_fmt.sv	Defines formatting information for display.	S2P environment
s2p_pgm.sv	Creates the modeling for simulation and initiates the *run* in the environment	Top level
s2p_env.sv	Creates the build and start for simulation	program
s2p_mon_xactor.sv	Creates a copy of the observed transaction onto a transaction channel.	Scoreboard, top level
top_tb.sv	Represents the top level and instantiates the RTL, the bind, the monitor, etc	none
s2p_scen_gen.sv	Uses the macro *`vmm_scenario_gen* for generation of scenario generator.	Environment for creation of the build model,
s2p_scenario_defs.sv	Contains scenario definition class with constraints	Environment

/ch8/ch8_s2p_scheduler directory: Channel Scheduler

File	Function	Used by
s2p_if.sv	Defines the Serial-to-Parallel interface	RTL and by program, testbench, transaction and transactors
s2p_xactn.sv	Defines the transaction class with the constraints Also used for the channel generation with: *`vmm_channel* (S2p_xactn)	*`vmm_channel* macro for generation of channel, *`vmm_atomic_gen* macro for generation of atomic generator, monitor transactor for creation of transaction from observed values on bus interface.
s2p_log_fmt.sv	Defines formatting information for display.	S2P environment
s2p_gen_xactor.sv	Uses the macro *`vmm_atomic_gen* for generation of atomic generator, defines the constraints for the number of transactions	Environment for creation of the build model,
s2p_asink.sv	Creates artificial sink for verifying the scheduler model	Environment for verifying the scheduler output
s2p_scheduler_env.sv	Creates the scheduler and the envrionment	program
s2p_pgm.sv	Creates the modeling for simulation and initiates the *run* in the environment	Top level
top_tb.sv	Represents the top level and instantiates the RTL, the bind, the monitor, etc	none

/ch8/ch8_eth_bcast directory : Channel Broadcater

File	Function	Used by
eth_xactn.sv	Defines the transaction class with the constraints Also used for the channel generation with: *`vmm_channel* (Eth_xactn)	*`vmm_channel* macro for generation of channel, *`vmm_atomic_gen* macro for generation of atomic generator.
eth_log_fmt.sv	Defines formatting information for display	ETH environment
eth_asink.sv	Creates artificial sink for verifying the Broadcaster model	Environment for verifying the broadcaster output
eth_gen_xactor.sv	Uses the macro *`vmm_atomic_gen* for generation of atomic generator, defines the constraints for the number of transactions	Environment for creation of the build model
eth_bcast_env.sv	Creates the scheduler and the envrionment	program
eth_bcast_pgm.sv	Creates the modeling for simulation and initiates the *run* in the environment	Top level
eth_top.sv	Represents the top level and instantiates environment	none

Chapter 8 Questions and LAB

Q1. Why are scenario generator needed? Why can't atomic generator be used instead?

Q2. What technique can be used to define a constraint for a sequence that has 3 PUSH, 2 IDLE, 4 POP instructions?

Q3. How do I repeat a scenario few times?

Q4. Why do I get a default atomic transaction when I use *`vmm_scenario_gen* macro?

Q5. I created a scheduler using *vmm-scheduler* – what is the quickest way to verify its behavior without going through very many DUT simulation cycles?

Q6. In VMM scheduler, how do I get a pure random scheduling (not the default round robin scheme)?

Lab08

See instructions in subdirectory lab/lab08/todo/readme.txt.