VMM Scoreboarding User Guide

Version C-2009.06
June 2009
VMM SB Package Version 1.3 (Synopsys)

Comments?
E-mail your comments about this manual to:
vcs_support@synopsys.com.

SYNOPSYS®

Copyright Notice and Proprietary Information

Right to Copy Documentation

The license agreement with Synopsys permits licensee to make copies of the documentation for its internal use only. Each copy shall include all copyrights, trademarks, service marks, and proprietary rights notices, if any. Licensee must assign sequential numbers to all copies. These copies shall contain the following legend on the cover page:

> This document is duplicated with the permission of Synopsys, Inc., for the exclusive use of
>
> _____ and its employees. This is copy number_____."

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Registered Trademarks (®)

Synopsys, AMPS, Cadabra, CATS, CRITIC, CSim, Design Compiler, DesignPower, DesignWare, EPIC, Formality, HSIM, HSPICE, iN-Phase, in-Sync, Leda, MAST, ModelTools, NanoSim, OpenVera, PathMill, Photolynx, Physical Compiler, PrimeTime, SiVL, SNUG, SolvNet, System Compiler, TetraMAX, VCS, Vera, and YIELDirector are registered trademarks of Synopsys, Inc.

Trademarks (™)

AFGen, Apollo, Astro, Astro-Rail, Astro-Xtalk, Aurora, AvanWaves, Columbia, Columbia-CE, Cosmos, CosmosEnterprise, CosmosLE, CosmosScope, CosmosSE, DC Expert, DC Professional, DC Ultra, Design Analyzer, Design Vision, DesignerHDL, Direct Silicon Access, Discovery, Encore, Galaxy, HANEX, HDL Compiler, Hercules, Hierarchical Optimization Technology, HSIMplus, HSPICE-Link, iN-Tandem, i-Virtual Stepper, Jupiter, Jupiter-DP, JupiterXT, JupiterXT-ASIC, Liberty, Libra-Passport, Library Compiler, Magellan, Mars, Mars-Xtalk, Milkyway, ModelSource, Module Compiler, Planet, Planet-PL, Polaris, Power Compiler, Raphael, Raphael-NES, Saturn, Scirocco, Scirocco-i, Star-RCXT, Star-SimXT, Taurus, TSUPREM-4, VCS Express, VCSi, VHDL Compiler, VirSim, and VMC are trademarks of Synopsys, Inc.

Service Marks (SM)

MAP-in, SVP Café, and TAP-in are service marks of Synopsys, Inc.

SystemC is a trademark of the Open SystemC Initiative and is used under license.
ARM and AMBA are registered trademarks of ARM Limited.
Saber is a registered trademark of SabreMark Limited Partnership and is used under license.
All other product or company names may be trademarks of their respective owners.

Contents

A Scoreboarding Support Classes

1

Introduction

In many verification environments, the self-checking mechanism involves the use of a transaction-descriptor storage, retrieval and comparison infrastructure called a scoreboard.

The functionality of scoreboards can be generalized for different application domains. However, different scoreboards may be required for different application domains. It will be necessary to use the set of foundation classes the best correspond to the application to be verified.

All VMM scoreboarding application classes are prefixed with "vmm_sb". All foundations classes belonging to a particular application domain are further prefixed with a domain-specific prefix. For example, all data stream scoreboard foundation classes are prefixed with "vmm_sb_ds_".

At this time, a set of foundation classes is provided for the following types of applications:

Data Streams

Data stream applications involve the transmission, multiplexing, prioritization or transformation of data items. Data stream applications include—but are not limited to—busses, bridges, codecs, switches, routers and network processors.

2

Data Streams

This chapter provides guidelines and techniques for implementing a self-checking structure for data stream applications. Data stream applications involve the transmission, multiplexing, prioritization or transformation of data items. Data stream applications include—but are not limited to—busses, bridges, codecs, switches, routers and network processors.

Data stream scoreboards described in this chapter are based on the "vmm_sb_ds" class. Out of the box, this foundation class can be used to verify a single stream of ordered but unmodified data. This foundation class also supports—through user extensions—multiple concurrent in-order data streams, user-defined multiplexing of concurrent input data streams to concurrent output data streams, data losses and user-defined data transformations.

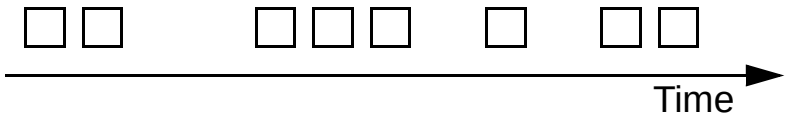
Even though data stream scoreboards are applicable to a wide range of applications, this chapter uses the term "packet" to describe the data items or transactions that are being checked. This does not

imply that these guidelines and foundation class are only applicable to packet-based systems. They are applicable to any data stream-oriented system, whether they process packets, frames, segments, cycles, digitized samples, etc...

What is a stream?

A stream is a sequence of packets, as illustrated by Figure 2-1. The stream may have an implied order. Packets in a stream may arrive at different intervals.

Figure 2-1 Data Stream



Streams are usually composed of packets of the same or compatible type.

In VMM, a stream is usually composed of transaction descriptors, based on the `vmm_data` class. Therefore, a `vmm_channel` can be used to transport a data stream between two transactors.

Logical stream

A stream of packets flowing between two transactors through a `vmm_channel` instance or to the DUT through a physical interface is a structural stream. Through multiplexing, it may be composed of packets from different input streams or destined to different output streams. If there is a way to identify the original input stream a packet

came from or the destination output stream a packet is going to, a structural stream can be said to be composed of multiple logical sub-streams.

Packets

Packets must be based on the `vmm_data` base class to be usable with the "vmm_sb_ds" foundation class. It is especially important that the `vmm_data::compare()` and `vmm_data::psdisplay()` method be appropriately implemented for the packet class. Although required for VMM compliance, the other `vmm_data` methods need not be implemented.

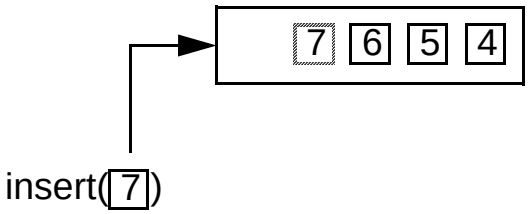
Single stream

This section shows how to use the "vmm_sb_ds" foundation class to implement self-checking structures for a single data stream. The single-stream application is the simplest one to use as it requires no extensions to the foundation class. Single stream applications need not worry about stream identifiers.

The techniques shown here can be combined with the techniques dealing with multiple streams, transformations and user-defined behavior illustrated in the following sections in this chapter to implement a self-checking structure that matches the functionality of your design.

Streams are ordered sequences of expected packets. The order of the sequence is determined by the order in which the packets where added to the scoreboard, as illustrated by Figure 2-2. A packet is added to a scoreboard by using the “vmm_sb_ds::insert()” method as shown in Figure 2-2.

Figure 2-2 Adding an Expected packet in a Data Stream Scoreboard



In-order stream

An in-order stream scoreboard is a simple FIFO, as illustrated in Figure 2-3. It is defined as in-order by the absolute ordering expected in the observed response. In an in-order data stream, the observed packets must be in the exact same order they were sent to the DUT: the next observed output packet must match the expected packet located at the front of the scoreboard. Whether or not an observed response matches or does not match the packet at the front of the scoreboard, as illustrated in Figure 2-4, the expected

packet is removed from the scoreboard, as illustrated in Figure 2-5. The subsequent observed packet will thus be matched with the subsequent packet in the scoreboard.

Figure 2-3 Expected Response in In-Order Data Stream

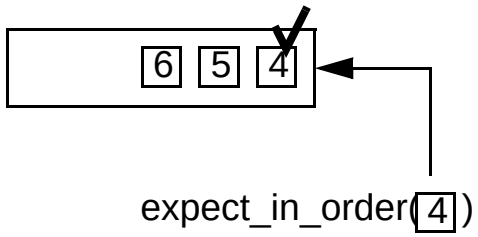


Figure 2-4 Unexpected Response in In-Order Data Stream

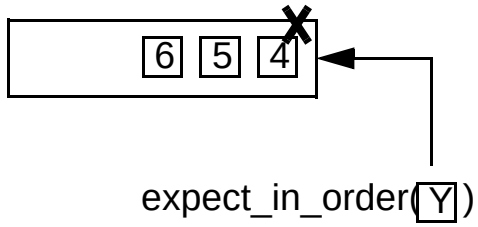


Figure 2-5 Data Stream Scoreboard After Response

To check that the packets form an in-order data stream, the "vmm_sb_ds::expect_in_order()" method is used as shown in Example 2-3. The matching of the observed packet and the expected packet is performed by the "vmm_sb_ds::compare()", which by default calls the packet::compare() method.

Example 2-1 Single-stream in-order scoreboard

```
class my_env extends vmm_env;
   ...
   eth_tx    tx;
```

```
   eth_rx    rx;
   vmm_sb_ds sb;
   ...
   virtual function void build();
      super.build();
      ...
      this.sb = new("Ethernet Frame Stream");
      ...
   endfunction: build
   ...
   virtual task start();
      super.start();
      ...
      fork
         forever begin
            eth_frame fr;
            this.tx.notify.wait_for(eth_tx::TXED);
            fr = this.tx.notify.status(eth_tx::TXED);
            this.sb.insert(fr);
         end

         forever begin
            eth_frame fr;
            this.rx.notify.wait_for(eth_tx::RXED);
            fr = this.rx.notify.status(eth_tx::RXED);
            this.sb.expect_in_order(fr);
         end
      join_none
      ...
   endtask: start
   ...
endclass: my_env
```

Losses

A strict in-order stream scoreboarding strategy may be difficult to implement in the presence of packet losses. If it is possible to predict the exact packets that will be lost, they (or their corresponding expected output packets) can be removed from the scoreboard by using the “vmm_sb_ds::remove()” method.

However, predicting the exact packets that will be lost may be almost impossible without duplicating the detailed RTL architecture of the design or snooping into the design. An alternative is to accept that "some" packets will be lost without trying to predict exactly which ones. If lost packets are acceptable, the “vmm_sb_ds::expect_with losses()” method is used. As illustrated in Figure 2-6, it looks for an expected packet matching the supplied observed packet and, when found, assumes that all expected packets in front of the matching packet were lost. The final state of the scoreboard is shown in Figure 2-7.

Figure 2-6 Response in Lossy Data Stream

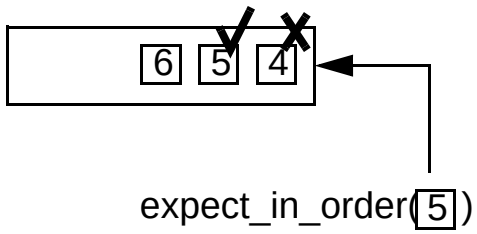


Figure 2-7 Data Stream Scoreboard After Response

The “vmm_sb_ds::expect_with losses()” method returns the matching packet, as well as the packets that were assumed to have been lost. As shown in Example 2-2, it is important to ensure that the number of lost packets is acceptable.

The matching of the observed packet and the expected packet is performed by the “vmm_sb_ds::quick_compare()”, which by default always returns 1 “vmm_sb_ds::match()”, which by default calls

"vmm_sb_ds::quick_compare()" which by default always returns 1. It is thus important to overload this method when using the "vmm_sb_ds::expect_with losses()" method.

Example 2-2 Single-stream scoreboard with losses

```
class my_env extends vmm_env;
   ...
   eth_tx    tx;
   eth_rx    rx;
   vmm_sb_ds sb;
   int       n_lost;
   ...
   virtual function void build();
      super.build();
      ...
      this.sb = new("Ethernet Frame Stream");
      ...
   endfunction: build
   ...
   virtual task start();
      super.start();
      ...
      fork
         forever begin
            eth_frame fr;
            this.tx.notify.wait_for(eth_tx::TXED);
            fr = this.tx.notify.status(eth_tx::TXED);
            this.sb.insert(fr);
         end

         forever begin
            eth_frame fr;
            vmm_data  mtch, lost[];
            this.rx.notify.wait_for(eth_tx::RXED);
            fr = this.rx.notify.status(eth_tx::RXED);
            this.sb.expect_with_losses(fr, mtch, lost);
            this.n_lost += lost.size();
           if (this.n_lost > 10) `vmm_fatal(log, "Too many
losses");
         end
      join_none
      ...
   endtask: start
   ...
```

```
endclass: my_env
```

Out-of-order stream

If the order in which packets will be observed is unpredictable, the "vmm_sb_ds::expect_out_of_order()" method is used. As illustrated in Figure 2-8, it looks for an expected packet matching the supplied observed packet and, when found, removes it—and only it—from the scoreboard. The final state of the scoreboard is shown in Figure 2-9.

Figure 2-8 Response in Out of Order Data Stream

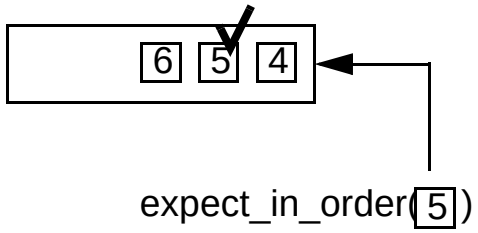


Figure 2-9 Data Stream Scoreboard After Response

Example 2-3 shows an example of using an out-of-order single-stream scoreboarding approach. The matching of the observed packet and the expected packet is performed by the "vmm_sb_ds::compare()", which by default calls the packet::compare() method.

Example 2-3 Single-stream out-of-order scoreboard

```
class my_env extends vmm_env;
   ...
   eth_tx    tx;
   eth_rx    rx;
```

```
   vmm_sb_ds sb;
   ...
   virtual function void build();
      super.build();
      ...
      this.sb = new("Ethernet Frame Stream");
      ...
   endfunction: build
   ...
   virtual task start();
      super.start();
      ...
      fork
         forever begin
            eth_frame fr;
            this.tx.notify.wait_for(eth_tx::TXED);
            fr = this.tx.notify.status(eth_tx::TXED);
            this.insert(fr);
         end

         forever begin
            eth_frame fr;
            eth_frame lost;
            this.rx.notify.wait_for(eth_tx::RXED);
            fr = this.rx.notify.status(eth_tx::RXED);
            this.expect_out_of_order(fr);
         end
      join_none
      ...
   endtask: start
   ...
endclass: my_env
```

Multiple streams

Designs often have to deal with multiple streams. Multiple input streams can be combined into a single expected stream. An input stream can be divided into multiple expected streams. Multiple input streams can be combined and divided into one or more expected streams.

An input stream is assumed to flow into the design under test and thus into the scoreboard. An input stream is a stream of stimulus packets. An expected stream is assumed to flow out of the design under test. An expected stream is a stream of expected or observed packets.

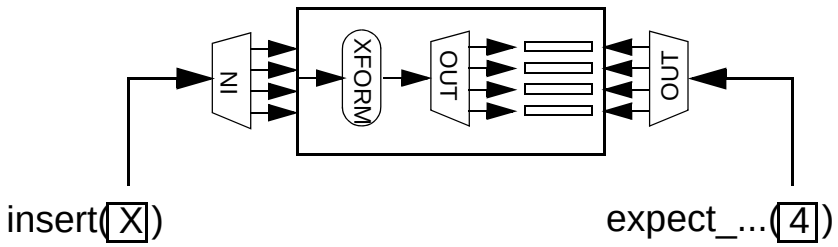
The VMM data stream scoreboard foundation class handles multiple input and expected streams. Each input stream is identified by a user-defined unique non-negative integer identifier. Each expected stream is similarly identified by a user-defined non-negative integer identifier. The input stream identifiers are kept separate from the expected stream identifiers and need not be mutually unique.

Streams can be explicitly identified or they can be identified by the "vmm_sb_ds::stream_id()" method. By default, if a stream identifier is not explicitly specified, the "vmm_sb_ds::stream_id()" is used to determine the input or expected stream identifier for a specific packet. By default, the "vmm_sb_ds::stream_id()" method always returns the value of "0". Thus, without any modifications to the scoreboard foundation class, only one input stream and one expected stream can be used. This method can be user-extended to identify the input or expected stream a packet belongs to. The user code implements the mapping from the content of the packet to the corresponding stream identifier.

As illustrated in Figure 2-10, an INPUT packet is part of an input stream. It is added to the scoreboard using the "vmm_sb_ds::insert()" method. An EXPECT packet is part of an expected stream. It is added to the scoreboard as an expected output (after transforming an input packet into the corresponding packet via the "vmm_sb_ds::transform()" method) or as an observed

response supplied to one of the response checking methods “vmm_sb_ds::expect_in_order()”, “vmm_sb_ds::expect_with losses()” or “vmm_sb_ds::expect_out_of_order()”.

Figure 2-10 Multiple Stream Scoreboard



Example 2-4 shows an example of mapping packets into 256 input and 256 expected streams based on the packet source or destination address, according to its kind. The number of streams is somewhat arbitrary (but should be kept as small as possible) and the stream identifiers need not be consecutive or adjacent values.

Example 2-4 Mapping packets to streams

```
class multi_stream_eth_sb extends vmm_sb_ds;

   virtual function int stream_id(vmm_data pkt,
     kind_e   kind);
       eth_frame fr;

       $cast(fr, pkt);
       if (kind == INPUT) begin
          return fr.src[7:0];
       end
       else begin
          return fr.dst[7:0];
       end
   endfunction: stream_id

endclass: multi_stream_eth_sb
```

If the input and expected streams for a packet can be determined form the packet content, as shown in Example 2-4, it is preferable to use an extension of the “vmm_sb_ds::stream_id()” method to map a packet to the proper input and expected streams. However, if the input or expected stream a packet belongs to cannot be determined by the content of the packet itself, the appropriate stream identifier must be explicitly specified when adding, removing or comparing a packet in the scoreboard. Example 2-5 shows an example of explictly mapping packets into various input and expected streams based on the stream identifier of the transactor inserting the packet into the scoreboard.

Example 2-5 Explicitly mapping packets to streams

```
class mac_to_sb extends eth_mac_callbacks;
   vmm_sb_ds sb;
   ...
   virtual task post_tx(eth_mac   xactor,
                        eth_frame fr);
      this.sb.insert(.pkt          (fr),
                     .inp_stream_id(xactor.stream_id));
   endtask

   virtual function void post_rx(eth_mac   xactor,
                                 eth_frame fr);
      this.sb.expect_in_order(.pkt           (fr),
                              .exp_stream_id(xactor.stream_id));
   endfunction
endclass
```

Other than providing mappings to the appropriate input or expected stream, multiple-stream scoreboards operate and are used in the exact same way as single-stream scoreboards, as shown in section titled "Single stream" on page 2-5.

Multiple expected stream, single input stream

A Single Input, Multiple Expect (SIME) data stream device takes packets from a single source and demultiplexes them onto one or more destinations. Checking the response of such a design involves making sure that all packets were observed on the proper expected stream. Whether or not the ordering of the packets is maintained by the design is a separate question answered by using the appropriate response checking method, as described in section titled "Single stream" on page 2-5.

The data stream scoreboard foundation class supports the checking of a SIME function where individual packets in the expected response corresponding to each input packet is found on a single expected stream. If the same expected packet is to be found on several or all expected streams, the functionality of the scoreboard and its comparison functions must be extended as described in section titled "User-defined behavior" on page 2-30.

Example 2-6 shows an example of mapping packets into 256 expected streams based on the packet destination address. For input stream identifiers, the default stream identifier is returned reguardless of the content of the packet.

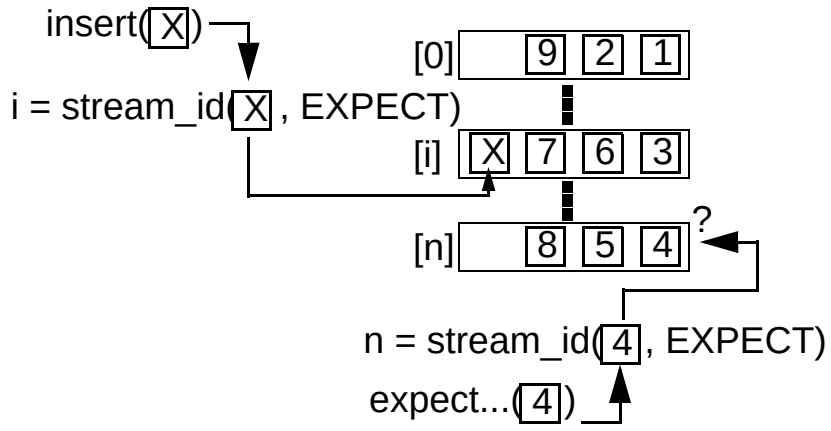
Example 2-6 Mapping packets to multiple output streams

```
class multi_stream_eth_sb extends vmm_sb_ds;

   virtual function int stream_id(vmm_data pkt,
     kind_e   kind);
       eth_frame fr;

       $cast(fr, pkt);
       if (kind == EXPECT) begin
          return fr.dst[7:0];
       end
       return super.stream_id(pkt, kind);
```

```
   endfunction: stream_id

endclass: multi_stream_eth_sb
```

Figure 2-11 illustrates how the incoming packets are stored then later compared against observed packets in the “vmm_sb_ds” class. There is a queue for each expected stream. Queues are dynamically created when a new expected stream identifier is seen, unless streams have been predefined using the “vmm_sb_ds::define_stream()” method. Incoming packets are appended to the queue corresponding to the expected stream identifer they are mapped to (in this illustration by the “vmm_sb_ds::stream_id()” method because an expected stream identifier has not been explicitly specified) and stored in order of arrival. Observed packets are compared against the packets found in the queue corresponding to the expected stream identifier the observed packet is mapped to (in this illustration through the “vmm_sb_ds::stream_id()” method).

Figure 2-11 SIME Stream Scoreboard



Single expected stream, multiple input stream

A Multiple Input, Single Expect (MISE) data stream device takes packets from multiple sources and multiplexes them onto a single destination. Checking the response of such a design usually involves making sure that all packets were observed on the expected stream in the proper order of the respective input stream. Whether or not the ordering of the packets is maintained by the design is a separate question answered by using the appropriate response checking method, as described in section titled "Single stream" on page 2-5.

The data stream scoreboard foundation class supports the checking of a MISE function where individual packets in the expected response corresponding to each input packet are multiplexed on the expected stream without any priority or fairness. If the expected packets from different input streams are to be found in a specific relative order on the expected stream, the functionality of the scoreboard and its comparison functions must be extended as described in section titled "User-defined behavior" on page 2-30.

Example 2-7 shows an example of mapping packets into 256 input streams based on the packet source address. For expected stream identifiers, the default stream identifier is returned regardless of the content of the packet.

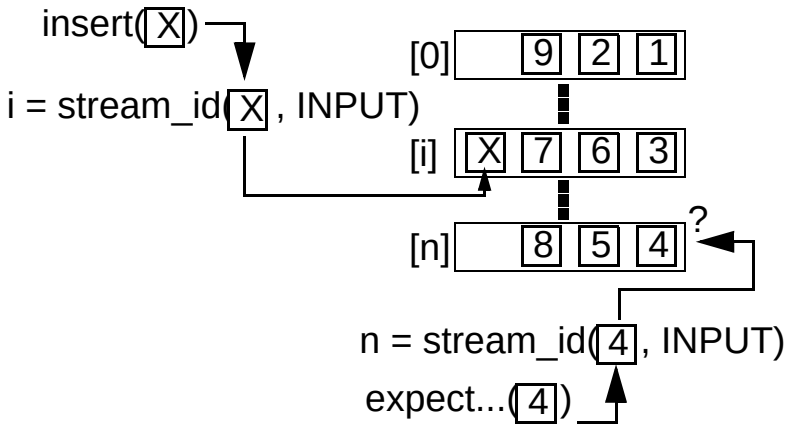
Example 2-7 Mapping packets to multiple input streams

```
class multi_stream_eth_sb extends vmm_sb_ds;

   virtual function int stream_id(vmm_data pkt,
    kind_e   kind);
      eth_frame fr;

      $cast(fr, pkt);
      if (kind == INPUT) begin
         return fr.src[7:0];
      end
```

```
      return super.stream_id(pkt, kind);
   endfunction: stream_id

endclass: multi_stream_eth_sb
```

Figure 2-11 illustrates how the incoming packets are stored then later compared against observed packets in the “vmm_sb_ds” class. There is a queue for each input stream. Queues are dynamically created when a new input stream identifier is seen, unless streams have been predefined using the “vmm_sb_ds::define_stream()” method. Incoming packets are appended to the queue corresponding to the input stream identifer they are mapped to (in this illustration by the “vmm_sb_ds::stream_id()” method because an input stream identifier has not been explicitly specified) and stored in order of arrival. Observed packets are compared against the packets found in the queue corresponding to the input stream identifier the observed packet is mapped to (in this illustration through the “vmm_sb_ds::stream_id()” method).

Figure 2-12 MISE Stream Scoreboard



If the input stream of an observed packet cannot be determined, it will be necessary to search all of the queues in the scoreboard to check if the observed packet is expected. Example 2-8 shows how

such a search is accomplished, expecting a strict in-order response within a given input stream. The “vmm_sb_ds::expect_with losses()” or “vmm_sb_ds::expect_out_of_order()” method can be used if different ordering is expected.

Example 2-8 Checking against unknown input stream

```
function bit expect_in_any(vmm_data, pkt);
   vmm_sb_ds_iter in_str = sb.new_sb_iter();
   while (in_str.next()) begin
      if (sb.expect_in_order(.pkt           (pkt),
                  .inp_stream_id(in_str.inp_stream_id()),
                  .exp_stream_id(in_str.exp_stream_id()),
                      .silent      (1)) != null) return 1;
   end
  `vmm_error(log, {"Unexpected packet: \n", pkt.psdisplay("
")});
   return 0;
endfunction
```

Multiple input and expected streams

A Multiple Input, Multiple Expected (MIME) data stream device takes packets from a multiple source and routes them onto one or more destinations. Checking the response of such a design involves making sure that all packets were observed on the proper expected stream in the proper order relative to other expected packets from the same input stream. Whether or not the ordering of the packets is maintained by the design is a separate question answered by using the appropriate response checking method, as described in section titled "Single stream" on page 2-5.

The data stream scoreboard foundation class supports the checking of a MIME function where individual packets in the expected response corresponding to each input packet is found on a single expected stream. If an the same expected packet is to be found on

several or all expected streams, the functionality of the scoreboard and its comparison functions must be extended as described in section titled "User-defined behavior" on page 2-30.

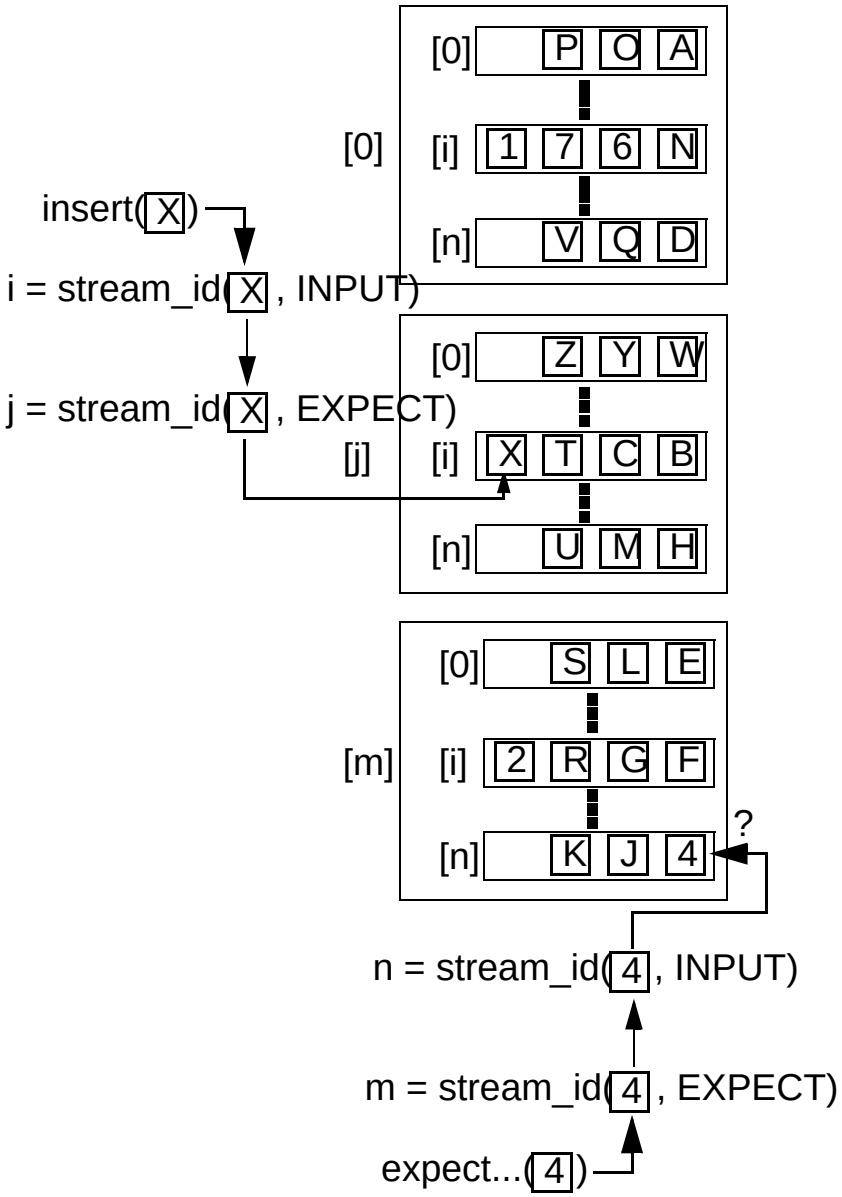
Furthermore, the MIME function, as supported by the data stream scoreboard foundation class, also assumes that individual packets in the expected response corresponding to each input packet are multiplexed on their respective expected stream without any priority or fairness. If the expected packets from different input streams are to be found in a specific relative order on an expected stream, the functionality of the scoreboard and its comparison functions must be extended as described in section titled "User-defined behavior" on page 2-30.

Example 2-4 shows an example of mapping packets into 256 input streams and 256 expected streams based on the packet source and destination addresses respectively.

Figure 2-11 illustrates how the incoming packets are stored then later compared against observed packets in the “vmm_sb_ds” class. There is a queue for each input stream within each expected stream. Queues are dynamically created when a new stream identifier is seen, unless streams have been predefined using the “vmm_sb_ds::define_stream()” method. Incoming packets are appended to the queue corresponding to the input and expected stream identifers they are mapped to (in this illustration by the “vmm_sb_ds::stream_id()” method because stream identifiers have not been explicitly specified) and stored in order of arrival. Observed packets are compared against the packets found in the queue

corresponding to the stream identifiers the observed packet is mapped to (in this illustration through the “vmm_sb_ds::stream_id()” method).

Figure 2-13 MIME Stream Scoreboard



If the input stream of an observed packet cannot be determined, it will be necessary to search all of the input stream queues in the scoreboard corresponding to a specific expected stream to check if the observed packet is expected. Example 2-9 shows how such a search is accomplished, expecting a strict in-order response within a given input stream. The “vmm_sb_ds::expect_with losses()” or “vmm_sb_ds::expect_out_of_order()” method can be used if different ordering is expected.

Example 2-9 Checking against unknown input stream within a known expected stream

```
function bit expect_in_any(vmm_data, pkt);
   vmm_sb_ds_iter in_str;

   in_str = sb.new_sb_iter(.exp_stream_id(
     sb.stream_id(pkt, vmm_sb_ds::EXPECT)));

   while(in_str.next()) begin
      if (sb.expect_in_order(.pkt           (pkt),
                  .inp_stream_id(in_str.inp_stream_id()),
                  .exp_stream_id(in_str.exp_stream_id()),
                      .silent      (1)) != null) return 1;
   end
  `vmm_error(log, $psprintf("Unexpected packet on port #%0d:
\n%s",
                             in_str.out_stream_id(),
                             pkt.psdisplay("   ")));
   return 0;
endfunction
```

Transformations

So far, inputs were assumed to be simply moved from an input to an output. But designs often transform the data flowing through them. Input packets are transformed into output packets. Input values are used to compute output values.

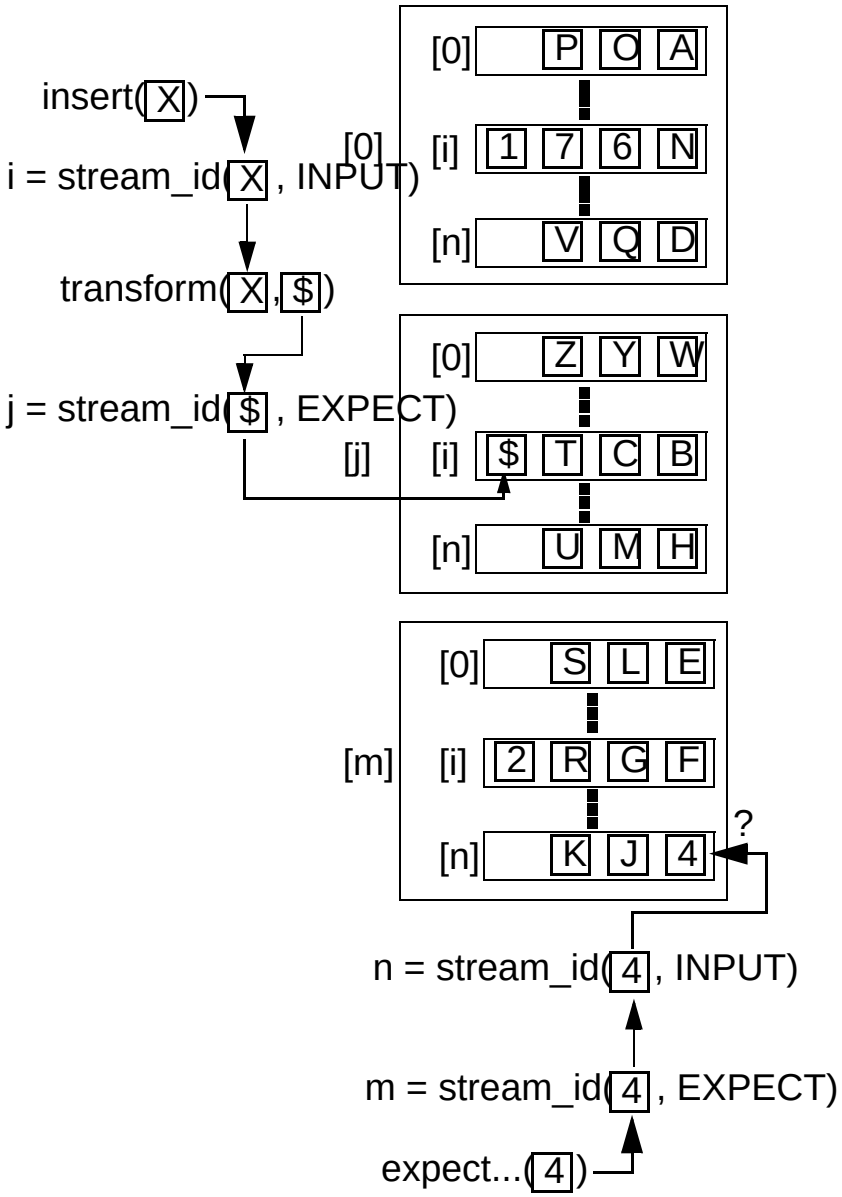
This section deals with how input packets are transformed into expected packets through a user-defined transfer function. All examples in this section assume a single-input, single-expected stream of packets. The concepts and examples shown in this section can be combined with the concepts and techniques shown in other sections—such as multiple streams—to create a scoreboard that matches your design.

The VMM data stream scoreboard foundation class handles arbitrary transformation of input packets into expected packets. A single input packet can be transformed into none, one or many expected packets. For example, a DSP function would produce one expected response for each input packet. A segmentation function would produce many expected packets for each input packet. And a reassembly function would produce an expected packet only every few input packets.

As illustrated in Figure 2-10, the transformation of input packets into expected packets is defined by the “vmm_sb_ds::transform()” method. By default, this method does not modify the input packet and the expected packet is identical to the input packet. This method can be user-extended to model the transformation of input packets

into the corresponding expected packets. The expected packets are then appended to their respective queue in the scoreboard, as described in the previous section and illustrated in Figure 2-14.

Figure 2-14 Multi-stream Transformation Scoreboard



The expected packets need not be of the same type as the input packets.

Other than providing a transformation algorithm from an input stream into one or more expected streams, transformation scoreboards operate and are used in the exact same way as single-stream or multiple-stream scoreboards.

One-to-one transformation

A one-to-one transformation device takes individual input packets and transforms them into individual expected packets. Checking the response of such a design involves making sure that proper packets were observed. Whether or not the ordering of the packets is maintained by the design or how packets are assigned to specific expected streams are separate questions answered by using the appropriate response checking method, as described in section titled "Single stream" on page 2-5 and identifying the proper streams a packet belongs to as described in section titled "Multiple streams" on page 2-12.

Example 2-10 shows an example of transforming input samples into expected samples using a digital filter. It is important to ensure that the proper state information is maintained between invocation of the “vmm_sb_ds::transform()” method. Static configuration information should be provided through the scoreboard constructor or via a "reconfigure()" method.

Example 2-10 Transforming one input packet into one expected packet

```
class dsp_sb extends vmm_sb_ds;

   local float a[];
   local float b[];
   local float z[];
```

```
   function new(ref float a[],
                ref float b[]);
      super.new("DSP Function");
      if (a.size() != b.size()) `vmm_fatal(log, "...");
      this.a = new [a.size()] (a);
      this.b = new [a.size()] (b);
      this.z = new (a.size()];
      foreach (this.z[i]) this.z[i] = 0.0;
   endfunction: new

   virtual function bit transform(input  vmm_data in_pkt,
                                  output vmm_data out_pkts[]);
      sample d;
      float  zi = 0.0;

      $cast(d, in_pkt);

      zi = d.value;
      d = new;
      d.value = 0.0;
      for (int i = this.a.size()-1; i > 0; i--) begin
         zi      += this.a[i] * this.z[i];
         d.value += this.b[i] * this.z[i];
         this.z[i] = this.z[i-1];
      end
      this.z[0] = zi * this.a[0];
      d.value = (d.value + this.z[0]) * this.b[0];

      out_pkts = new [1];
      out_pkts[0] = d;

      return 1;
   endfunction: transform

endclass: dsp_sb
```

One-to-many transformation

A one-to-many transformation device takes individual input packets and transforms them into multiple expected packets. Checking the response of such a design involves making sure that the proper

packets were observed. Whether or not the ordering of the packets is maintained by the design or how packets are assigned to specific expected streams are separate questions answered by using the appropriate response checking method, as described in section titled "Single stream" on page 2-5 and identifying the proper streams a packet belongs to as described in section titled "Multiple streams" on page 2-12.

Example 2-11 shows an example of segmenting IP frames into IP segments. Static configuration information should be provided through the scoreboard constructor or via a "reconfigure()" method.

Example 2-11 Transforming an input packet into multiple expected packets

```
class ip_seg_sb extends vmm_sb_ds;

   local int MTU;

   function new(int MTU);
      this.MTU = MTU;
   endfunction

   virtual function bit transform(input  vmm_data in_pkt,
                                  output vmm_data out_pkts[]);
      ip_frame ip, seg;
      bit [7:0] bytes[];
      int n, i;

      $cast(ip, in_pkt);
      n = ip.byte_size();
      if (n <= this.MTU) begin
         out_pkts = new [1] ({ip});
         return 1;
      end
      if (ip.DF) return 0;

      ip.byte_pack(bytes);
      out_pkts = new [((n-20)/(this.MTU-20)) + 1];
      n = 0; i = 20;
      while (i < bytes.size()) begin
         seg = new;
         ...
```

```
            out_pkts[n++] = seg;
         end

         return 1;
      endfunction: transform

endclass: ip_seg_sb
```

Many-to-one transformation

A many-to-one transformation device combines multiple input packets—possibly received out of order—and combines them into a single expected packet. Checking the response of such a design involves making sure that the proper combined packets were observed. Whether or not the ordering of the packets is maintained by the design or how packets are assigned to specific expected streams are separate questions answered by using the appropriate response checking method, as described in section titled "Single stream" on page 2-5 and identifying the proper streams a packet belongs to as described in section titled "Multiple streams" on page 2-12.

Example 2-12 shows an example of decapsulating HDLC frames from a stream of ATM cells. Any state information that must be saved across inpust packets must be kept in local data members.

Example 2-12 Transforming multiple input packets into an expected packet

```
class hdlc_atm_decaps_sb extends vmm_sb_ds;
   local bit [7:0] bytes[];
   local bit escape = 0;

   virtual function bit transform(input  vmm_data in_pkt,
                                  output vmm_data out_pkts[]);
      atm_cell atm;
      $cast(atm, in_pkt);
      foreach (atm.payload[i]) begin
         bit [7:0] b = atm.payload[i];
```

```
            if (b == 8'h7E) begin
               hdlc_frame hdlc = new;
               hdlc.byte_unpack(bytes);
               out_pkts = new [1] ({hdlc});
               bytes.delete();
               return 1;
            end

            if (b == 8'h7D) begin
               escape = 1;
               continue;
            end

            if (escape) begin
               case (b)
               8'h5E: b = 8'h7E;
               7'h5D: b = 8'h7D;
               endcase
               escape = 0;
            end

            bytes = new [bytes.size() + 1] (bytes);
            bytes[$] = b;
         end
         return 1;
      endfunction: transform

endclass: hdlc_atm_decaps_sb
```

User-defined behavior

The functionality of the data stream scoreboard described so far supports relatively simple expectation functions. Should a different expectation function be necessary, the data stream scoreboard foundation classes provide functionality that enables any user-defined expectation function to be written.

Also, the functionality described so far only supports single-destination for input streams, with no predictive data loss. Some designs require that an expected packet be expected on multiple expected streams. Others require that an expected packet be removed from an expected stream. The foundation classes provide the necessary functionality to match the requirements of the design under verification.

Iterators

The actual data structure used to implement the data stream scoreboard is not detailed in this user's guide. In fact, it is entirely private to the implementation of the foundation classes. This allows the user interface to be removed from the scoreboard implementation.

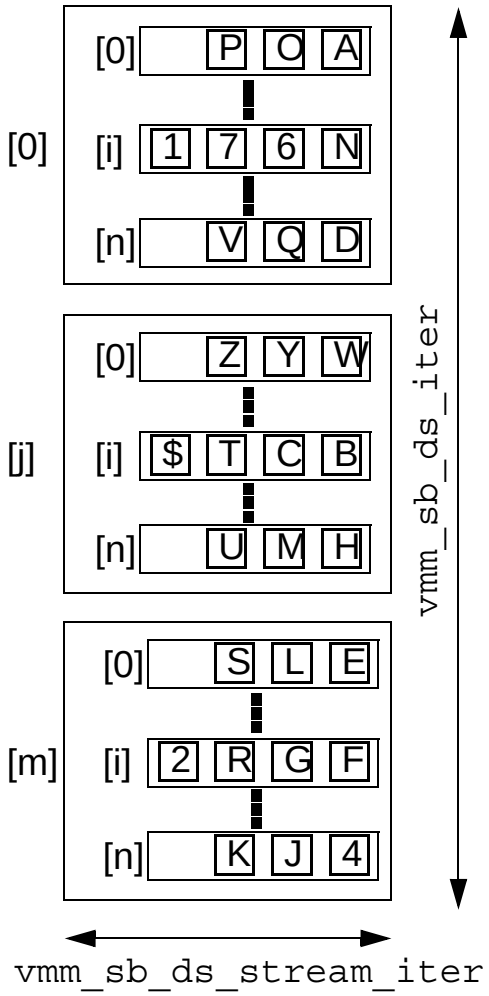
However, implementing user-defined functionality requires that the entire content of the scoreboard be made available so it can be searched and modified. This can be done without exposing the underlying implementation through *iterators*.

Iterators are objects that know how to traverse and navigate the implementation of the scoreboard. They provide high-level methods for moving through the scoreboard and modifying its content at the location of the iterator.

There are two kinds of iterators: scoreboard iterators—the "vmm_sb_ds_iter" class—that move from one stream of expected packets to another, and stream iterators—the

"vmm_sb_ds_stream_iter" class—that move from one expected packet to another on a single packet stream. The different motions of the iterators is illustrated in Figure 2-15.

Figure 2-15 Iterator Motions



The constructors for the iterators are not documented because they cannot be created on their own. They have to be created by a scoreboard using the "vmm_sb_ds::new_sb_iter()" or "vmm_sb_ds::new_stream_iter()" methods or an scoreboard iterator

using the “vmm_sb_ds_iter::new_stream_iter()” method. Iterators are created to operator on a specific scoreboard or stream and cannot be relocated to another scoreboard or stream.

Locating a packet

The complexity of locating a packet depends on the a priori restrictions you can impose on its whereabouts. The more you can assume about the location of a packet, the easier and more efficient it will be to find it.

Locating a packet always requires using two iterators: one to locate the appropriate stream, the other to locate the packet within the stream. Example 2-13 shows how these "nested" iterators are used to locate a specific packet in the scoreboard.

Example 2-13 Locating a packet in the scoreboard

```
class my_sb extends vmm_sb_ds;
   protected function vmm_data locate(vmm_data pkt);
      vmm_sb_ds_iter scan_sb = this.new_sb_iter();

      while (scan_sb.next()) begin
         vmm_sb_ds_stream_iter scan_str =
scan_sb.new_stream_iter();
         while (scan_str.next()) begin
            vmm_data is_it = scan_str.data();
            string diff;
            if (is_it.quick_compare(pkt) &&
                is_it.compare(pkt, diff)) return is_it;
         end
      end
      return null;
   endfunction: locate
endclass
```

If the packet is known to be located on a specific input stream for a specific expected stream, the "outer" iterator can be short-circuited and the "inner" created directly on the relevant stream. Example 2-14 shows how.

Example 2-14 Locating a packet in a known stream

```
class my_sb extends vmm_sb_ds;
   protected function vmm_data locate(vmm_data pkt);
      vmm_sb_ds_stream_iter scan;

      scan = this.new_stream_iter(this.stream_id(pkt,
                                       vmm_sb_ds::EXPECT),
                                  this.stream_id(pkt,
                                       vmm_sb_ds::INPUT));
      while (scan.next()) begin
         vmm_data is_it = scan.data();
         string diff;
         if (is_it.quick_compare(pkt) &&
             is_it.compare(pkt, diff)) return is_it;
      end
      return null;
   endfunction: locate
endclass
```

Inserting a packet

A packet can be inserted before or after the current position of an iterator using the “vmm_sb_ds_stream_iter::prepend()” or “vmm_sb_ds_stream_iter::append()” methods respectively. "Before" refers to "earlier" in the stream whereas "after" refers to "later" in the stream. An earlier packet will be expected before a later packet. Example 2-15 shows how an input packet can be multicasted on all expected streams. Notice how the extension of the “vmm_sb_ds::transform()” method never returns an expected packet

to avoid it from being added automatically to an expected stream (and thus duplicated) by the default behavior of the scoreboard foundation class.

Example 2-15 Multicasting a packet on all output streams

```
class my_sb extends vmm_sb_ds;
virtual function bit transform(input  vmm_data in_pkt,
                               output vmm_data out_pkts[]);
      vmm_sb_ds_iter scan_sb = this.new_sb_iter();

      while (scan_sb.next()) begin
         vmm_sb_ds_stream_iter scan_str =
scan_sb.new_stream_iter();
         scan_str.last();
         scan_str.append(in_pkt);
      end
      out_pkts = new [0]; // Avoid duplication
   endfunction: locate
endclass
```

Discarding a packet

The packet an interator is currently located on can be deleted using the “vmm_sb_ds_stream_iter::delete()” method. Alternatively, all packets located before or after the current position of the iterator can be deleted by using the “vmm_sb_ds_stream_iter::preflush()” or “vmm_sb_ds_stream_iter::postflush()” methods respectively. "Before" refers to "earlier" in the stream whereas "after" refers to "later" in the stream. An earlier packet will be expected before a later packet. Example 2-16 shows how all packets after a packet to be found of greater priority are deleted for a specific output streams.

Example 2-16 Dropping subsequent lower priority packets

```
class my_sb extends vmm_sb_ds;
virtual function bit transform(input  vmm_data in_pkt,
                               output vmm_data out_pkts[]);
      vmm_sb_ds_stream_iter scan;
```

```
      my_packet inp, outp;

      $cast(inp, in_pkt);

      scan = this.new_stream_iter(this.stream_id(pkt,
                                       vmm_sb_ds::EXPECT),
                                  this.stream_id(pkt,
                                       vmm_sb_ds::INPUT));
      while (scan.next()) begin
         $cast(outp, scan.data());

         if (outp.priority() < inp.priority())
scan.postflush();
      end
      out_pkts = new [0];
      out_pkts[0] = in_pkt;
   endfunction: locate
endclass
```

Integrating Scoreboards

Stimulus and observed response packets must be reported to the scoreboard for prediction and comparison against expectations. There are various ways this can be accomplished.

The advantage of using the VMM scoreboarding foundation classes is that they can take advantage of predefined scoreboard integration points in the VMM infrastructure.

In all cases, it is important to understand if a reported packet instance can be directly added to the scoreboard or if a copy mut first be done. If an instance is added directly, it is possible that it may be modified by other transactors in the verification environmnt or reused to describe subsequent packets unrelated to the first one.

All the examples in the section will copy the packets prior to submission to the scoreboard. Although safe, copying packets increases run-time and memory consumption. If using instances directly is acceptable, simply remove the copy operation.

Integrating with callback extensions

Any scoreboard can be integrated with a transactor—be it a master transactor or a passive monitor—using the callback methods provided by that transactor. But because of its non-specific nature and flexibility of application, it is also the most verbose. Example 2-17 shows how a scoreboard can be integrated using a "post transaction" callback method in a master transactor.

Example 2-17 Integrating a scoreboard via callback extension

```
class ahb_to_sb extends ahb_master_callbacks;
   my_sb sb;

   function new(my_sb sb);
      this.sb = sb;
   endfunction

   virtual task post_tr(ahb_master xactor,
                        ahb_tr     tr);
      vmm_data cpy = tr.copy();
      this.sb.insert(cpy);
   endtask
endclass

class my_env extends vmm_env;
   my_sb sb;
   ahb_master ahb;

   virtual function build();
      super.build();

      this.sb = new();
      this.ahb = new(...);
```

```
      begin
         ahb_to_sb cb = new(this.sb);
         this.ahb.append_callback(cb);
      end
   endfunction: build
endclass
```

Scoreboards should be integrated using callback methods that are invoked at the end of a transaction execution. This will ensure that a scoreboard will see the final result of a transaction.

Integrating with extended vmm_xactor

The VMM Standard Library can be optionally extended to facilitate the integration of scoreboards based on the VMM data stream scoreboard foundation classes. The scoreboard integration methods described below are added to the `vmm_xactor` class when the symbol ‘VMM_SB_DS_IN_STDLIB is defined.

A transactor based on the extended `vmm_xactor` class may call the `vmm_xactor::inp_vmm_ds_sb()` method after executing a transaction and the `vmm_xactor::exp_vmm_sb_ds()` method after observing a transaction. These methods are usually invoked after the post-transaction callback method that would be used to integrate a scoreboard using the generic callback extension mechanism as described above. Example 2-18 shows how the AHB master transactor would need to be modified to call this new scoreboard integration method.

Example 2-18 Scoreboard integration point in a transactor

```
class ahb_master extends vmm_xactor;
   ...
   virtual task main();
      super.main();
      forever begin
         wait_if_stopped_or_empty(this.in_chan);
```

```
         this.in_chan.activate(tr);
         ...
        `vmm_callback(ahb_master_callbacks, post_tr(this,
tr));
         inp_vmm_sb_ds(tr);
      end
   endtask
endclass: ahb_master
```

If a transactor supports the data stream scoreboard integration method, a scoreboard based on the data stream scoreboard foundation classes can be integrated very easily as shown in Example 2-19. Compare with Example 2-17 to appreciate the difference.

Example 2-19 Scoreboard integration with exhanced transactor

```
class my_env extends vmm_env;
   my_sb sb;
   ahb_master ahb;

   virtual function build();
      super.build();

      this.sb = new();
      this.ahb = new(...);

      this.ahb.register_vmm_sb_ds(this.sb);
   endfunction: build
endclass
```

Integrating with vmm_channel

Any scoreboard can be integrated with a monitor by sinking the output channel used to report observed transactions. However, by sinking the output channel, no other transactor can be connected to that same channel, often requiring the use of a `vmm_broadcast` component. Example 2-20 shows how a scoreboard can be integrated by sinking the output of a channel.

Example 2-20 Integrating a scoreboard by sinking an output channel

```
class my_env extends vmm_env;
   my_sb sb;
   ahb_monitor ahb;

   virtual function build();
      super.build();

      this.sb = new();
      this.ahb = new(...);
   endfunction: build

   virtual task start();
      super.start();

      this.ahb.start_xactor();
      fork
         forever begin
            ahb_tr tr;
            this.ahb.out_chan.get(tr);
            this.sb.expect_in_order(tr);
         end
      join_none
   endfunction: build
endclass
```

VMM channels offer the tee() method to sample the transactions flowing through it. This method should be reserved for implementing functional coverage. The VMM Standard Library can be optionally extended to facilitate the integration of scoreboards based on the VMM data stream scoreboard foundation classes. The scoreboard integration methods described below are added to the `vmm_channel` class when the symbol ‘VMM_SB_DS_IN_STDLIB is defined.

The extended VMM channel provides an unobtrusive mechanism that can be used to tap the flow of transactions through a channel and forward it to a scoreboard. By simply registering a data stream

scoreboard with a channel, all transactions removed from it will be forwarded to that scoreboard at the time of removal. Example 2-21 shows how this simplifies the integration process.

Example 2-21 Integrating a scoreboard by registration with a channel

```
class my_env extends vmm_env;
   my_sb sb;
   ahb_monitor ahb;

   virtual function build();
      super.build();

      this.sb = new();
      this.ahb = new(...);
      this.ahb.out_chan.register_vmm_sb_ds(this.sb,
vmm_sb_ds::EXPECT,
                                           vmm_sb_ds::IN_ORDER);
   endfunction: build
endclass
```

Integrating with vmm_notify

Any scoreboard can be integrated with a notification by waiting for its indication then sampling the associated status. Example 2-20 shows how a scoreboard can be integrated by waiting for a notification to be indicated.

Example 2-22 Integrating a scoreboard by notification indication

```
class my_env extends vmm_env;
   my_sb sb;
   ahb_monitor ahb;

   virtual function build();
      super.build();

      this.sb = new();
      this.ahb = new(...);
   endfunction: build
```

```
   virtual task start();
      super.start();

      this.ahb.start_xactor();
      fork
         forever begin
          this.ahb.notify.wait_for(ahb_monitor::OBSERVED);
            this.sb.expect_in_order(
           this.ahb.notify.status(ahb_monitor::OBSERVED));
         end
      join_none
   endfunction: build
endclass
```

Waiting for a notification indication is subject to the same event ordering and scheduling limitation than waiting for any other SystemVerilog event has. Should a notification be indicated multiple times within the same timestep with different status, not all indications and status might be seen. This is usually not a problem when indications are generated by a physical level transactor where transaction-level events are separated in time. But integrating a scoreboard on a transaction-level model using notifications may cause some reported transactions to be missed. Example 2-20 shows how notification indication callbacks can be used to catch all and each indications.

Example 2-23 Integrating a scoreboard by notification callback

```
class ahb_to_sb extends vmm_notify_callbacks;
   my_sb sb;

   function new(my_sb sb);
      this.sb = sb;
   endfunction

   virtual function void indicated(vmm_data status);
      vmm_data cpy = status.copy();
      this.sb.insert(cpy);
   endfunction
endclass
```

```
class my_env extends vmm_env;
   my_sb sb;
   ahb_monitor ahb;

   virtual function build();
      super.build();

      this.sb = new();
      this.ahb = new(...);

      begin
         ahb_to_sb cb = new(this.sb);

this.ahb.notify.append_callback(ahb_monitor::OBSERVED,
cb);
      end
   endfunction: build
endclass
```

The VMM Standard Library can be optionally extended to facilitate the integration of scoreboards based on the VMM data stream scoreboard foundation classes. The scoreboard integration methods described below are added to the `vmm_notify` class when the symbol ‘VMM_SB_DS_IN_STDLIB is defined.

The extended VMM notification service provides an unobtrusive mechanism that can be used catch the status of all notification indications and forward it to a scoreboard. By simply registering a data stream scoreboard with a notification, all status indications will be forwarded to that scoreboard at the time of indication. Example 2-21 shows how this simplifies the integration process.

Example 2-24 Integrating a scoreboard by registration with a notification

```
class my_env extends vmm_env;
   my_sb sb;
   ahb_monitor ahb;

   virtual function build();
      super.build();
```

```
      this.sb = new();
      this.ahb = new(...);

this.ahb.notify.register_vmm_sb_ds(ahb_monitor::OBSERVED,
                                   this.sb, vmm_sb_ds::EXPECT,
                                         vmm_sb_ds::IN_ORDER);
   endfunction: build
endclass
```

A

Scoreboarding Support Classes

This appendix provides detailed documentation of the classes that are used to implement and support self-checking structures.

The OpenVera and SystemVerilog classes have identical functionality and features. There are thus documented together. The heading used to introduce a method uses the SystemVerilog name. The OpenVera name will be identical except for the few cases where `a _t` suffix is appended to indicate that it may be a blocking method.

Usage examples are usually specified in a single language but that should not deter users of the other language as they would be almost identical. This document prefers to provide more different examples than almost identical examples in each language.

The classes are documented in alphabetical order. The methods in each class are documented in a logical order, where methods that accomplish similar results are documented sequentially. A summary

of all available methods with cross references to the page where their detailed documentation can be found is provided at the beginning of each class specification.

Class Summary

vmm_sb_ds

This class implements a generic data stream scoreboard. A single instance of this class is used to check the proper transformation, multiplexing and ordering of multiple data streams.

For guidelines on using and extending this class to match a specific application, see the section titled “Data Streams” on page 2-3.

The documentation for this class uses the term "packet" to describe a data item to be inserted or checked in the scoreboard. The term is used for as a convenience as does not imply that the class is limited to data streams composed of packets. It is suitable for any stream of data, composed of frames, fragments, bus cycles, transfers, etc...

Summary

vmm_sb_ds::new()

Create a new instance of a data stream scoreboard.

SystemVerilog

```
function new(string name);
```

OpenVera

```
task new(string name);
```

Description

Create an instance of a data stream scoreboard with the specified name.

The specified name is used as the instance name of the message interface found in the “vmm_sb_ds::log” class property.

Examples

Example A-1

```
class my_sb extends vmm_sb_ds;
. . .
   function new(string name);
      super.new(name);
   endfunction
   . . .
endclass

class my_env extends vmm_env;
   . . .
   my_sb sb = new("Simple");
   . . .
endclass
```

vmm_sb_ds::log

Message service interface for the data stream scoreboard.

SystemVerilog

```
vmm_log log;
```

OpenVera

```
rvm_log log;
```

Description

Message service interface used to issue messages from this data stream scoreboard. The name of the interface is hardcoded as "Data Stream Scoreboard". The instance name of the interface is the name of the scoreboard specified in the constructor. These names may be modified afterward using the vmm_log::set_name() or vmm_log::set_instance() methods.

Examples

Example A-2

```
class my_sb extends vmm_sb_ds;
   vmm_log log;
   function new();
      super.new(". . .");
      log = new("MY","SCBD");
      `vmm_note(this.log, "Log method of vmm_sb_ds");
      . . .
   endfunction
   . . .
endclass
```

vmm_sb_ds::notify

Notification service interface for the data stream scoreboard.

SystemVerilog

```
vmm_notify notify;
```

OpenVera

```
rvm_notify notify;
```

Description

Notification service interface used to indicate notifications from this data stream scoreboard. Notifications are indicated after any callback methods.

A scoreboard normally operates in zero-time. it is thus possible that notifications may be indicated multiple times during the same timestep if the corresponding methods are called multiple times within the same timestep. If it is important that each and every indication be caught, it is necessary to use an extension of the vmm_notify_callbacks::indicated() method registered with this notification service interface. See the VMM Standard Library User's Guide for more details.

The following notifications are indicated under the specified circumstances:

vmm_sb_ds::INSERTED
An expected packet has been inserted in the scoreboard. The status is an instance of "vmm_sb_ds_pkts", describing the inserted packet. This notification is indicated only if the

"vmm_sb_ds::insert()" method is used. It is not indicated if a packet is inserted directly into a stream using the "vmm_sb_ds_stream_iter::prepend()" or "vmm_sb_ds_stream_iter::append()" methods.

vmm_sb_ds::EMPTY
An ON/OFF indication indicating whether the scoreboard is empty or not. When indicated, the scoreboard is empty.

vmm_sb_ds::MATCHED
An expected packet has been matched and removed from the scoreboard. The status is an instance of "vmm_sb_ds_pkts", describing the matched packet.

vmm_sb_ds::MISMATCHED
An expected packet has been mis-matched by the "vmm_sb_ds::expect_with losses()" method and removed from the scoreboard. The status is an instance of "vmm_sb_ds_pkts", describing the mis-matched observed (pkts[0]) and (pkts[1]) expected packet.

vmm_sb_ds::DROPPED
One or more expected packets have been assumed lost by the "vmm_sb_ds::expect_with losses()" method and removed from the scoreboard. The status is an instance of "vmm_sb_ds_pkts", describing the dropped packet(s).

vmm_sb_ds::NOT_FOUND
An observed packet has not been found in the scoreboard. The status is an instance of "vmm_sb_ds_pkts", describing the packet not found.

vmm_sb_ds::ORPHANED
One or more expected packets are left over in the scoreboard. This notification is indicated only the first time the "vmm_sb_ds::get_n_orphaned()" method is used. The status is an instance of "vmm_sb_ds_pkts", describing the orphaned packet(s). Because orphaned packets can be from different streams, the input

and expected stream identifiers are invalid. Use the "vmm_sb_ds_callbacks::orphaned()" method if it is necessary to know orphaned packets on a per-stream basis.

Examples

Example A-3

```
//Example for vmm_sb_ds::INSERTED
vmm_sb_ds_pkts my_pkt;
. . .
sb.insert(pkt,vmm_sb_ds::INPUT,ip_id,exp_id);
. . .
//NOTE: You can use vmm_notify_callbacks::indicated also.
my_pkt = sb.notify.status(vmm_sb_ds::INSERTED);
. . .

//Example for vmm_sb_ds::EMPTY
class my_env extends vmm_env;
   . . .
   task wait_for_end();
      . . .
      sb.notify.wait_for(vmm_sb_ds::EMPTY);
      . . .
   endtask
   . . .
endclass

//Example for vmm_sb_ds::MATCHED
. . .
sb.insert(pkt,vmm_sb_ds::INPUT,ip_id,exp_id);
. . .
exp_pkt = sb.expect_in_order(pkt,ip_id,exp_id);
. . .
my_pkt = sb.notify.status(vmm_sb_ds::MATCHED);
. . .

//Example for vmm_sb_ds::MISMATCHED & vmm_sb_ds::DROPPED
. . .
vmm_sb_ds_pkts my_pkt;
sb.insert(pkt,vmm_sb_ds::INPUT,ip_id,exp_id);
```

```
. . .
exp_pkt =
sb.expect_with_losses(pkt,matched,lost,ip_id,exp_id);
. . .
my_pkt = sb.notify.status(vmm_sb_ds::MISMATCHED);
//Use my_pkt.pkts[0] and my_pkt.pkts[1] to get he mismatched
packets.
//OR
my_pkt = sb.notify.status(vmm_sb_ds::DROPPED);
. . .

//Example for vmm_sb_ds::NOT_FOUND
//Refer previous examples
. . .
my_pkt = sb.notify.status(vmm_sb_ds::NOT_FOUND);
. . .

//Example for vmm_sb_ds::ORPHANED
. . .
orphaned_pkts = sb.get_n_orphaned();
. . .
my_pkt = sb.notify.status(vmm_sb_ds::ORPHANED);
//For callback example refer A-84
. . .
```

vmm_sb_ds::kind_e

Symbolic values for packet kind.

SystemVerilog

```
typedef enum {EITHER, INPUT, EXPECT} kind_e;
```

OpenVera

```
typedef enum {EITHER, INPUT, EXPECT} kind_e;
```

Description

These symbols are used to specify the kind or purpose of a packet to the scoreboard functionality.

vmm_sb_ds::EITHER
Specify that the direction of the packet is not relevant for the operation.

vmm_sb_ds::INPUT
Specify that the packet is an input packet going into the design under verification and a corresponding response is to be expected.

vmm_sb_ds::EXPECT
Specify that the packet is an expected response packet coming out of the design under verification and has been observed or is to be expected.

Examples

Example A-4

```
class my_sb extends vmm_sb_ds;
   . . .
```

```
  virtual function int stream_id(vmm_data pkt, kind_e kind);
      my_pkt tr;
      $cast(tr, pkt);
      if (kind == INPUT) begin
        return 1 + super.stream_id(pkt, kind);
      end
      `vmm_note(this.log,"Kind method of vmm_sb_ds");
   endfunction
   . . .
endclass
```

vmm_sb_ds::stream_id()

Return a stream identifier for a packet.

SystemVerilog

```
virtual function int stream_id(vmm_data pkt,
    kind_e   kind = EITHER)
```

OpenVera

```
virtual function integer stream_id(rvm_data pkt,
    kind_e   kind = EITHER)
```

Description

Return a non-negative stream identifier corresponding to the specified packet and the specified packet kind. This method can be used to determine the stream a packet belongs to based on the packet's content, such as a source or destination address.

By default, the direction is ignored and the value 0 is returned.

Examples

Example A-5

```
class my_sb extends vmm_sb_ds;
   . . .
  virtual function int stream_id(vmm_data pkt, kind_e kind);
      my_pkt tr;
      $cast(tr, pkt);
      if (kind == INPUT) begin
        return 1 + super.stream_id(pkt, kind);
      end
      if (kind == EXPECT) begin
        return super.stream_id(pkt, kind);
```

```
      end
      if (kind == EITHER) begin
        return super.stream_id(pkt, kind);
      end
      `vmm_note(this.log,"Stream_id method of vmm_sb_ds");
   endfunction
   . . .
endclass

class my_master_cbs extends vmm_xactor_callbacks;
   . . .
endclass: my_master_cbs

class my_master_to_sb extends my_master_cbs;
   . . .
   id = sb.stream_id(pkt,vmm_sb_ds::INPUT);
   `vmm_note(this.log,
   $psprintf("Stream_id()= %0h method of
vmm_sb_ds",stream_id));
   . . .
endclass
```

vmm_sb_ds::define_stream()

Pre-define a packet stream.

SystemVerilog

```
function void define_stream(int    stream_id,
     string descr = ""
     kind_e kind = EITHER)
```

OpenVera

```
task define_stream(integer stream_id,
     string  descr = ""
     kind_e  kind = EITHER)
```

Description

Pre-define a data stream and associate the optional description with the specified stream. The identifier must be a non-negative number.

Use this method to pre-defined stream identifiers. Any subsequently specified stream identifier that has not been pre-defined will be considered invalid. If streams are defined as "EITHER" kind, then they must all be defined as "EITHER" and each expected stream has one and only one input stream.

If this method is never used, streams will be dynamically created as needed whenever a new stream identifier is observed.

Examples

Example A-6

```
class my_sb extends vmm_sb_ds;
   . . .
```

```
   function new();
      super.new(". . .");
      this.define_stream(0, "SRC" , INPUT);
      this.define_stream(0, "DST 0", EXPECT);
      this.define_stream(1, "DST 1", EXPECT);
      `vmm_note(this.log,
      $psprintf("Define_stream method of vmm_sb_ds"));
   endfunction
   . . .
endclass
```

vmm_sb_ds::insert()

Insert a packet into the scoreboard

SystemVerilog

```
virtual function bit insert(vmm_data pkt,
     kind_e   kind          = INPUT,
     int      exp_stream_id = -1,
     int      inp_stream_id = -1)
```

OpenVera

```
virtual function bit insert(rvm_data pkt,
     kind_e   kind          = INPUT,
     integer  exp_stream_id = -1,
     integer  inp_stream_id = -1)
```

Description

Insert the specified packet into the scoreboard and returns TRUE if the insertion was succesful.

If the specified kind is "INPUT", the packet is considered a stimulus packet and will be first transformed by calling the "vmm_sb_ds::transform()" method then the resulting expected packet(s) inserted in the scoreboard, using the "vmm_sb_ds::insert()" method with an "EXPECT" kind.

If the specified direction is "EXPECT", the packet is considered an expected response packet. It will be appended, as-is, to the expected packet queue corresponding to the input stream identifier (if an input stream identifier is not specified, it will be determined by calling the "vmm_sb_ds::stream_id()" with an "INPUT" direction) in the group of queues corresponding to the expected stream identifier

(if an expected stream identifier is not specified, it will be determined by calling the “vmm_sb_ds::stream_id()” with an "EXPECT" direction).

It is invalid to call this method with a kind specified as "EITHER".

The effect on existing iterators is unspecified.

Examples

Example A-7

```
class my_master_to_sb extends my_master_cbs;//Refer Example
A-5
   my_sb sb;
   int unsigned id;
   . . .
   function new(my_sb sb);
      this.sb = sb;
   endfunction
   . . .
   id = sb.stream_id(pkt,vmm_sb_ds::INPUT);
   void'(this.sb.insert(pkt,vmm_sb_ds::INPUT,id,id));
   `vmm_note(this.log,"Insert method of vmm_sb_ds");
   . . .
endclass
```

vmm_sb_ds::remove()

Remove a packet from the scoreboard

SystemVerilog

```
virtual function bit remove(vmm_data pkt,
     kind_e   kind          = INPUT,
     int      exp_stream_id = -1,
     int      inp_stream_id = -1)
```

OpenVera

```
virtual function bit remove(rvm_data pkt,
     kind_e   kind          = INPUT,
     integer  exp_stream_id = -1,
     integer  inp_stream_id = -1)
```

Description

Remove the specified packet from the scoreboard and returns TRUE if the corresponding packets were succesfully found in the scoreboard then removed.

If the specified direction is "INPUT", the packet is considered a stimulus packet and will be first transformed by calling the “vmm_sb_ds::transform()” method then the resulting expected packet(s) removed from the scoreboard, using the “vmm_sb_ds::remove()” method with an "EXPECT" direction. If an input packet is transformed into multiple expected packets, TRUE is returned if all expected packets were successfully removed. If one or more expected packets where not removed, FALSE is returned and whatever expected packets that could be removed are removed.

If the specified direction is "EXPECT", the packet is considered an expected response packet. The first packet that compares to the specified packet in the expected packet queue corresponding to the specified input stream identifier (if not specified, it will be determined by calling the “vmm_sb_ds::stream_id()” with an "INPUT" direction) in the group of queues corresponding to the specified expected stream identifier (if not specified, it will be determined by calling the “vmm_sb_ds::stream_id()” with an "EXPECT" direction) is removed and TRUE is returned. If no matching packet is found, FALSE is returned.

It is invalid to call this method with a kind specified as "EITHER".

The effect on existing iterators is unspecified.

Examples

Example A-8

```
class my_master_to_sb extends my_master_cbs;//Refer Example
A-5
   int pkt_cnt;
   bit is_rmv;
   . . .
   void'(this.sb.insert(pkt,vmm_sb_ds::INPUT));
   if(pkt_cnt == 2) begin
      is_rmv = sb.remove(cycle,vmm_sb_ds::INPUT);
      if(is_rmv == 1)
       `vmm_note(this.log,
      $psprintf("Removed %0b method of vmm_sb_ds",is_rmv));
   end
   pkt_cnt ++;
   . . .
endclass
```

vmm_sb_ds::transform()

Transform an input packet into an expected response.

SystemVerilog

```
virtual function bit transform(input  vmm_data inp_pkt,
      output vmm_data exp_pkts[])
```

OpenVera

```
virtual function bit transform(    rvm_data inp_pkt,
     var rvm_data exp_pkts[*])
```

Description

Transform the specified stimulus packet into the corresponding response and returns TRUE if the transformation was succesful. A valid response can be composed of zero, one or several packets.

By default, returns the input packet, unmodified.

Examples

Example A-9

```
class my_sb extends vmm_sb_ds;
   . . .
   virtual function bit transform(input  vmm_data in_pkt,
                                  output vmm_data out_pkts[]);
      my_pkt p, q;
      $cast(p, in_pkt);
      out_pkts = new [p.no_of_pkt];
      foreach (out_pkts[i]) begin
         q = new();
         out_pkts[i] = q;
      end
```

```
   endfunction: transform
   . . .
endclass
class my_master_to_sb extends my_master_cbs;
   . . .
   //transform method automatically called by insert method
   void'(this.sb.insert(pkt,vmm_sb_ds::INPUT));
   `vmm_note(this.log,"transform method of vmm_sb_ds");
   . . .
endclass
```

vmm_sb_ds::match()

Match two packets.

SystemVerilog

```
virtual function bit match(vmm_data actual,
     vmm_data expected)
```

OpenVera

```
virtual function bit match(rvm_data actual,
     rvm_data expected)
```

Description

Match the two specified packets and return TRUE if they match and FALSE if they definitely do not match.

By default, calls “vmm_sb_ds::quick_compare()”.

Examples

Example A-10

```
class my_master_to_sb extends my_master_cbs;//Refer Example
A-5
   . . .
   my_pkt pkt_q[$];
   bit is_cmp;
   . . .
   void'(this.sb.insert(pkt,vmm_sb_ds::INPUT));
   is_cmp = sb.match(pkt_q[0],pkt);
   if(is_cmp == 1)
      `vmm_note(this.log,$psprintf("Match %0b (method of
vmm_sb_ds)",is_cmp));
   end
   . . .
```

```
endclass
```

vmm_sb_ds::quick_compare()

Quickly compare two packets.

SystemVerilog

```
virtual function bit quick_compare(vmm_data actual,
    vmm_data expected)
```

OpenVera

```
virtual function bit quick_compare(rvm_data actual,
    rvm_data expected)
```

Description

Do a quick comparison of the two specified packets and return TRUE if they potentially match and FALSE if they definitely do not match.

By default, returns TRUE.

Examples

Example A-11

```
class my_master_to_sb extends my_master_cbs;//Refer Example
A-5
   . . .
   void'(this.sb.insert(pkt,vmm_sb_ds::INPUT));
   is_cmp = sb.quick_compare(pkt_q[0],pkt);
   if(is_cmp == 1'b1)
    `vmm_note(this.log,$psprintf("Quick_compare %0b method
of vmm_sb_ds",
                 is_cmp));
   . . .
endclass
```

vmm_sb_ds::compare()

Compare two packets.

SystemVerilog

```
virtual function bit compare(vmm_data actual,
    vmm_data expected)
```

OpenVera

```
virtual function bit compare(rvm_data actual,
    rvm_data expected)
```

Description

Compare the two specified packets and return TRUE if they definitively match and return FALSE otherwise.

By default, calls “vmm_sb_ds::quick_compare()” followed by "actual.compare(expected)" if the quick comparison succeeded.

Examples

Example A-12

```
class my_pkt extends vmm_data;
   . . .
   virtual function bit compare(input  vmm_data to,
                                output string   diff,
                               input  int      kind = -1);
      my_pkt tr;
      . . .
      if (this.addr !== tr.addr) begin
         $sformat(diff, "Addr 0x%h !== 0x%h", this.addr,
tr.addr);
         return 0;
```

```
      end
      return 1;
      . . .
   endfunction: compare
   . . .
endclass

class my_master_to_sb extends my_master_cbs;//Refer Example
A-5
   . . .
   void'(this.sb.insert(pkt,vmm_sb_ds::INPUT));
   is_cmp = sb.compare(pkt_q[0],pkt);
   if(is_cmp == 1'b1)
      `vmm_note(this.log,$psprintf("Compare %b (method of
vmm_sb_ds)",is_cmp));
   . . .
endclass
```

vmm_sb_ds::expect_in_order()

Check a packet against the next expected packet.

SystemVerilog

```
virtual function vmm_data expect_in_order(vmm_data pkt,
     int      exp_stream_id = -1,
     int        inp_stream_id = -1,
     bit      silent = 0)
```

OpenVera

```
virtual function rvm_data expect_in_order(rvm_data pkt,
     integer  exp_stream_id = -1,
     integer  inp_stream_id = -1,
     bit      silent = 0)
```

Description

Check if a the specified packet compares to the packet at the front of the queue that corresponds to the specified input stream of the packet (if not specified, it will be determined by the “vmm_sb_ds::stream_id()” method with an "INPUT" direction) in the group of queues corresponding to the specified expected stream of the packet (if not specified, it will be determined by the “vmm_sb_ds::stream_id()” method with an "EXPECT" direction).

If the comparison is successful, the packet at the front of the queue is removed and returned. Otherwise, NULL is returned. The scoreboard statistics are updated based on the result.

If the "silent" parameter is TRUE, no error message is issued and the scoreboard statistics are not updated.

The effect on existing iterators is undefined.

Examples

Example A-13

```
class my_master_to_sb extends my_master_cbs;
   . . .
   sb.insert(pkt);
   . . .
   p = sb.expect_in_order(pkt);
   `vmm_note(log, "Expect_in_order() method of vmm_sb_ds");
   if (p == null) `vmm_error(log,"Null packet found");
   else if (p != pkt_q[0])
   `vmm_error(log, $psprintf("The packet not in order %s",
              p.psdisplay(" ")));
   . . .
endclass
```

vmm_sb_ds::expect_with losses()

Check a packet against the expected packet sequence.

SystemVerilog

```
virtual function bit expect_with_losses(
     input  vmm_data pkt,
     output vmm_data matched,
     output vmm_data lost[],
     input  int      exp_stream_id = -1,
     input  int      inp_stream_id = -1,
     input  bit      silent        = 0)
```

OpenVera

```
virtual function bit expect_with_losses(
     rvm_data     pkt,
     var rvm_data matched,
     var rvm_data lost[*],
     integer      exp_stream_id = -1,
     integer      inp_stream_id = -1,
     bit          silent        = 0)
```

Description

Important: This method requires that the "vmm_sb_ds::quick_compare()" method be properly overloaded to function properly. The default implementation will cause the first packet in the stream to always be matched.

Check if a the specified packet quickly compares (using the "vmm_sb_ds::quick_compare()""vmm_sb_ds::match()" method) to a packet in the queue that corresponds to the specified input stream of the packet (if not specified, it will be determined by the "vmm_sb_ds::stream_id()" method with an "INPUT" direction) in the

group of queues corresponding to the specified expected stream of the packet (if not specified, it will be determined by the “vmm_sb_ds::stream_id()” method with an "EXPECT" direction).

If no quick-matching packet is found, the number of packets not found is incremented.

If the quick comparison is successful, the matching packet is returned in the "matched" argument. All expected packets in the queue located in front of the matching packet are then removed from the scoreboard and returned in the "lost" argument . The number of packets assumed to have been lost is then added to the lost packet count.

If the matching packet is then fully compared to the expected packet (using the vmm_data::compare() method). If the packet fully matches, the number of matched packet is incremented otherwise the number of mismatched packets count is incremented.

Returns TRUE only if a packet that fully compares with the expected packet was found.

If the "silent" parameter is TRUE, no error message is issued and the scoreboard statistics are not updated.

The effect on existing iterators is undefined.

Examples

Example A-14

```
program test;
   . . .
   sb.insert(pkt);
   . . .
   if (!sb.expect_with_losses(pkt, p, lost)) begin
```

```
      `vmm_error(log, $psprintf("Packet was not matched"));
   end
  if (p == null) `vmm_error(log, $psprintf("Packet was not
found"));
   else if (p != pkt_q[0])
   `vmm_error(log, $psprintf("The wrong packet was
returned:\n%s",
              p.psdisplay(" ")));
   if (lost.size() != 0) begin
      `vmm_error(log, "Packets were lost");
      foreach (lost[i]) lost[i].display(" ");
   end
   . . .
endprogram
```

vmm_sb_ds::expect_out_of_order()

Check a packet against the expected packet stream.

SystemVerilog

```
virtual function vmm_data expect_out_of_order(
     vmm_data pkt,
     int      exp_stream_id = -1,
     int      inp_stream_id = -1,
     bit      silent        = 0)
```

OpenVera

```
virtual function rvm_data expect_out_of_order(
     rvm_data pkt,
     integer  exp_stream_id = -1,
     integer  inp_stream_id = -1,
     bit      silent        = 0)
```

Description

Check if a the specified packet compares to a packet anywhere in the queue that corresponds to the specified input stream of the packet (if not specified, it will be determined by the “vmm_sb_ds::stream_id()” method with an "INPUT" direction) in the group of queues corresponding to the specified expected stream of the packet (if not specified, it will be determined by the “vmm_sb_ds::stream_id()” method with an "EXPECT" direction).

If the comparison is successful, the matching packet is removed and returned. Otherwise, NULL is returned. The scoreboard statistics are updated based on the result.

If the "silent" parameter is TRUE, no error message is issued and the scoreboard statistics are not updated.

The effect on existing iterators is undefined.

Examples

Example A-15

```
program test;
   . . .
   pkt = new();
   sb.insert(pkt);
   . . .
   vmm_data p;
   pkt = new();
   p = sb.expect_out_of_order(pkt);
   `vmm_note(log, "Expect_out_of_order() method of
vmm_sb_ds");
   if (p == null) `vmm_error(log,"Null packet found");
   else if (p != pkt_q[i])
  `vmm_error(log, $psprintf(". . .  %s",p.psdisplay(" ")));
   . . .
endprogram
```

vmm_sb_ds::flush()

Reset the scoreboard.

SystemVerilog

```
virtual function void flush()
```

OpenVera

```
virtual task flush()
```

Description

Flush the entire content of the scoreboard and reset the scoreboard statistics.

The effect on existing iterators is undefined.

Examples

Example A-16

```
class my_env extends vmm_env; //Refer Example A-7
   my_master mst;
   . . .
   virtual task wait_for_end();
      super.wait_for_end();
      //Before Flush
      sb,report();
      sb.flush();
      `vmm_note(this.log,"Flush method of vmm_sb_ds");
      //After Flush
      sb,report();
      . . .
   endtask
   ...
```

```
endclass
```

vmm_sb_ds::new_sb_iter()

Create a scoreboard iterator.

SystemVerilog

```
function vmm_sb_ds_iter new_sb_iter(int exp_stream_id = -1,
     int inp_stream_id = -1)
```

OpenVera

```
function vmm_sb_ds_iter new_sb_iter(integer exp_stream_id =
-1,
     integer inp_stream_id = -1)
```

Description

Create and return a scoreboard iterator object that can iterate over all of the currently-defined streams of expected packets in the scoreboard.

If an expected stream identifier is specified, only the streams (one per input stream, if any) of expected packets for that expected stream identifier are iterated on.

If an input stream identifier is specified, only the streams (one per expected stream, if any) of expected packets for that input stream identifier are iterated on.

Examples

Example A-17

```
class my_env extends vmm_env;
   . . .
   vmm_sb_ds_iter sb_iter;
```

```
   . . .
   sb_iter = sb.new_sb_iter();
   if(sb_iter == null);
      `vmm_error(log,("vmm_sb_ds::new_sb_iter() method is
not working.");
   . . .
endclass
```

vmm_sb_ds::new_stream_iter()

Create a stream iterator.

SystemVerilog

```
function vmm_sb_ds_stream_iter new_stream_iter(int
exp_stream_id = -1,
     int inp_stream_id = -1)
```

OpenVera

```
function vmm_sb_ds_stream_iter new_stream_iter(
     integer exp_stream_id = -1,
     integer inp_stream_id = -1)
```

Description

Create and return a stream iterator object that can iterate over all of the expected packet in the specified stream.

If there is only one stream of expected packets, the expected stream identifier need not be specified.

If there is only one input stream of expected packets for the specified expected stream, the input stream identifier need not be specified.

Examples

Example A-18

```
class my_env extends vmm_env;
   . . .
   vmm_sb_ds_iter sb_iter;
   . . .
   sb_iter.stream_iter = sb.new_stream_iter();
   if(sb_iter.stream_iter == null);
```

```
     `vmm_error(log,("vmm_sb_ds::new_stream_iter() method
is not working.");
   . . .
endclass
```

vmm_sb_ds::prepend_callback()

Prepends a callback extension instance.

SystemVerilog

```
function void prepend_callback(vmm_sb_ds_callbacks sb)
```

OpenVera

```
task prepend_callback(vmm_sb_ds_callbacks sb)
```

Description

Prepends the specified callback extension instance to the registered callbacks for this scoreboard. Callbacks are invoked in the order of registration.

Examples

Example A-19

```
program my_test;
  class my_sb_ds_callbacks extends vmm_sb_ds_callbacks;
     . . .
  endclass
  initial
  begin
     my_sb_ds_callbacks cb;
     my_env env = new;
     . . .
     cb = new();
     env.sb.prepend_callback(cb);
    `vmm_note(log,"Prepend_callback() method of vmm_sb_ds");
     . . .
     env.run();
  end
```

```
endprogram
```

vmm_sb_ds::append_callback()

Appends a callback extension instance.

SystemVerilog

```
function void append_callback(vmm_sb_ds_callbacks sb)
```

OpenVera

```
task append_callback(vmm_sb_ds_callbacks sb)
```

Description

Appends the specified callback extension instance to the registered callbacks for this scoreboard. Callbacks are invoked in the order of registration.

Examples

Example A-20

```
program my_test;
  class my_sb_ds_callbacks extends vmm_sb_ds_callbacks;
     . . .
  endclass

  initial
  begin
     my_sb_ds_callbacks cb;
     my_env env = new;
     . . .
     cb = new();
     env.sb.append_callback(cb);
    `vmm_note(log,"Append_callback() method of vmm_sb_ds");
     . . .
     env.run();
```

```
  end
endprogram
```

vmm_sb_ds::unregister_callback()

Removes a callback extension instance.

SystemVerilog

```
function void unregister_callback(vmm_sb_ds_callbacks sb)
```

OpenVera

```
task unregister_callback(vmm_sb_ds_callbacks sb)
```

Description

Removes the specified callback extension instance from the registered callbacks for this scoreboard. A warning message is issued if the callback instance has not been previously registered.

Examples

Example A-21

```
program my_test;
  class my_sb_ds_callbacks extends vmm_sb_ds_callbacks;
     . . .
  endclass

  initial
  begin
     my_sb_ds_callbacks cb;
     my_env env = new;
     . . .
     cb = new();
     env.sb.append_callback(cb);
     //Can't register same callback moe than once.
     env.sb.append_callback(cb); //Wrong Way
    env.sb.unregister_callback(cb); //Now you can register
```

```
callback once again
     `vmm_note(log,"Unregister_callback() method of
vmm_sb_ds");
     . . .
     env.run();
  end
endprogram
```

vmm_sb_ds::get_n_inserted()

Total number of inserted EXPECT packets.

SystemVerilog

```
function int get_n_inserted(int exp_stream_id = -1,
     int inp_stream_id = -1)
```

OpenVera

```
function integer get_n_inserted(integer exp_stream_id = -1,
     integer inp_stream_id = -1)
```

Description

Return the total number of expected packets that have been inserted in the scoreboard.

If an expected stream identifier is specified, returns the total number of inserted expected packeds for that expected stream.

If an input stream identifier is specified, returns the total number of inserted expected packets for that input stream.

Examples

Example A-22

```
class my_env extends vmm_env; //Refer Example A-7
   . . .
   virtual function void build();
      . . .
   endfunction
   . . .
   `vmm_note(this.log,
   $psprintf("Get_n_inserted : %0d (method of vmm_sb_ds)",
```

```
   sb.get_n_inserted()));
   . . .
endclass
```

vmm_sb_ds::get_n_pending()

Total number of EXPECT packets still in the scoreboard.

SystemVerilog

```
function int get_n_pending(int exp_stream_id = -1,
    int inp_stream_id = -1)
```

OpenVera

```
function integer get_n_pending(integer exp_stream_id = -1,
    integer inp_stream_id = -1)
```

Description

Return the total number of expected packets still in the scoreboard.

If an expected stream identifier is specified, returns the total number of pending expected packeds for that expected stream.

If an input stream identifier is specified, returns the total number of pending expected packets for that input stream.

Examples

Example A-23

```
class my_env extends vmm_env; //Refer Example A-7
   . . .
   `vmm_note(this.log,
   $psprintf("Get_n_pending : %0d (method of vmm_sb_ds)",
   sb.get_n_pending()));
   . . .
endclass
```

vmm_sb_ds::get_n_matched()

Total number of matched packets.

SystemVerilog

```
function int get_n_matched(int exp_stream_id = -1,
     int inp_stream_id = -1)
```

OpenVera

```
function integer get_n_matched(integer exp_stream_id = -1,
     integer inp_stream_id = -1)
```

Description

Return the total number of matched expected packets.

If an expected stream identifier is specified, returns the total number of matched expected packeds for that expected stream.

If an input stream identifier is specified, returns the total number of matched expected packets for that input stream.

Examples

Example A-24

```
class my_env extends vmm_env; //Refer Example A-7
   . . .
   `vmm_note(this.log,
   $psprintf("Get_n_matched : %0d (method of vmm_sb_ds)",
   sb.get_n_matched()));
   . . .
endclass
```

vmm_sb_ds::get_n_mismatched()

Total number of mis-matched packets.

SystemVerilog

```
function int get_n_mismatched(int exp_stream_id = -1,
     int inp_stream_id = -1)
```

OpenVera

```
function integer get_n_mismatched(integer exp_stream_id = -
1,
     integer inp_stream_id = -1)
```

Description

Return the total number of mismatched expected packets as identified by the “vmm_sb_ds::expect_with losses()” method.

If an expected stream identifier is specified, returns the total number of mismatched expected packeds for that expected stream.

If an input stream identifier is specified, returns the total number of mismatched expected packets for that input stream.

Examples

Example A-25

```
class my_env extends vmm_env; //Refer Example A-7
   . . .
   `vmm_note(this.log,
  $psprintf("Get_n_mismatched : %0d (method of vmm_sb_ds)",
   sb.get_n_mismatched()));
   . . .
endclass
```

vmm_sb_ds::get_n_dropped()

Total number of dropped packets.

SystemVerilog

```
function int get_n_dropped(int exp_stream_id = -1,
     int inp_stream_id = -1)
```

OpenVera

```
function integer get_n_dropped(integer exp_stream_id = -1,
     integer inp_stream_id = -1)
```

Description

Return the total number of packets assumed dropped.

If an expected stream identifier is specified, returns the total number of dropped packeds for that expected stream.

If an input stream identifier is specified, returns the total number of dropped packets for that input stream.

Examples

Example A-26

```
class my_env extends vmm_env; //Refer Example A-7
   . . .
   `vmm_note(this.log,
   $psprintf("Get_n_dropped : %0d (method of vmm_sb_ds)",
   sb.get_n_dropped()));
   . . .
endclass
```

vmm_sb_ds::get_n_not_found()

Total number of unexpected packets.

SystemVerilog

```
function int get_n_not_found(int exp_stream_id = -1,
     int inp_stream_id = -1)
```

OpenVera

```
function integer get_n_not_found(integer exp_stream_id = -1,
     integer inp_stream_id = -1)
```

Description

Return the total number of packets that were not found in the scoreboard and thus assumed to be unexpected.

If an expected stream identifier is specified, returns the total number of unexpected packeds for that expected stream.

If an input stream identifier is specified, returns the total number of unexpected packets for that input stream.

Examples

Example A-27

```
class my_env extends vmm_env; //Refer Example A-7
   . . .
   `vmm_note(this.log,
   $psprintf("Get_n_orphaned : %0d (method of vmm_sb_ds)",
   sb.get_n_orphaned()));
   . . .
endclass
```

vmm_sb_ds::get_n_orphaned()

Total number of leftover packets.

SystemVerilog

```
function int get_n_orphaned(int exp_stream_id = -1,
     int inp_stream_id = -1)
```

OpenVera

```
function integer get_n_orphaned(integer exp_stream_id = -1,
     integer inp_stream_id = -1)
```

Description

Return the total number of expected packets remaining in the scoreboard.

If an expected stream identifier is specified, returns the total number of orphaned packeds for that expected stream.

If an input stream identifier is specified, returns the total number of orphaned packets for that input stream.

All pending expected packets in the scoreboard in the specified streams at the time this method is called are assumed to be orphaned. This method should be called at the end of simulation only.

This method indicates the **`vmm_sb_ds::ORPHANED`** notification the first time this method is called.

Examples

Example A-28

```
class my_env extends vmm_env; //Refer Example A-7
   . . .
   `vmm_note(this.log,
   $psprintf("Get_n_not_found : %0d (method of vmm_sb_ds)",
   sb.get_n_not_found()));
   . . .
endclass
```

vmm_sb_ds::report()

Report scoreboard statistics.

SystemVerilog

```
virtual function void report(int exp_stream_id = -1,
     int inp_stream_id = -1)
```

OpenVera

```
virtual task report(integer exp_stream_id = -1,
     integer inp_stream_id = -1)
```

Description

Report the statistics recorded by the scoreboard. By default, reports the total number of transactions matched, mismatched, dropped, not found and orphaned.

If an expected stream identifier is specified, report for that expected stream.

If an input stream identifier is specified, report for that input stream.

The total number of packets reported as not found may be greater than the sum of the packets reported as not found for the individual streams. These additional packets were not found because the specified stream did not exist.

Examples

Example A-29

```
class my_env extends vmm_env; //Refer Example A-7
   . . .
```

```
    `vmm_note(this.log,"Report method of vmm_sb_ds");
    sb.report();
    . . .
endclass
```

vmm_sb_ds::describe()

List and describe all streams.

SystemVerilog

```
function void describe();
```

OpenVera

```
task describe();
```

Description

Display the list of defined streams in the scoreboard along with any description previously provided via the "vmm_sb_ds::define_stream()" method.

To be displayed, a stream must exists and therefore must have seen at least one packet. If a stream has never had any traffic, it is not displayed. Simply defining a stream using "vmm_sb_ds::define_stream()" is not sufficient to include it in the description.

Examples

Example A-30

```
class my_master_to_sb extends my_master_cbs;//Refer Example
A-5
   . . .
   this.define_stream(0, "SRC" , INPUT);
   . . .
   sb.describe();
   `vmm_note(this.log,"Describe method of vmm_sb_ds");
   . . .
```

```
endclass
```

vmm_sb_ds::display()

Dump the scoreboard content.

SystemVerilog

```
virtual function void display(string prefix = "")
```

OpenVera

```
virtual task display(string prefix = "")
```

Description

Dump the content of the entire scoreboard on the standard output in a human-readable format.

Examples

Example A-31

```
class my_master_to_sb extends my_master_cbs;//Refer Example
A-5
   . . .
   void'(this.sb.insert(pkt,vmm_sb_ds::INPUT));
   sb.display("Display  Method : ");
   . . .
endclass
```

vmm_sb_ds_iter

This class implements an iterator that will iterate over all input streams in the scoreboard.

This class is used in combination with the "vmm_sb_ds_stream_iter" class to walk the content of the scoreboard to perform a user-defined checking operation.

Instances of this class are created by the "vmm_sb_ds::new_sb_iter()" method. This method defines which set of input streams will be iterated on. Iterators are created in the invalid state.

Streams are iterated on in increasing input stream identifiers first, then increasing expected stream identifiers second.

Summary

vmm_sb_ds_iter::first()

Reset the iterator to the first stream.

SystemVerilog

```
function bit first();
```

OpenVera

```
function bit first();
```

Description

Reset the iterator to the first applicable input stream. Returns TRUE if at least one such stream exists. Returns FALSE if no such stream exists.

Examples

Example A-32

```
. . .
my_sb sb = new("Simple");
vmm_sb_ds_iter sb_iter;
. . .
sb_iter = sb.new_sb_iter();
. . .
//Actually we are inserting more packets. But to use
vmm_sb_ds_iter::first method
//at least one packet must be present in the sb stream.
sb.insert(pkt);

if(sb_iter.first())
   `vmm_note(log,"Stream Iterator is reseted and set at the
1st stream position.");
. . .
```

vmm_sb_ds_iter::is_ok()

Check if the iterator is on a valid stream.

SystemVerilog

```
function bit is_ok();
```

OpenVera

```
function bit is_ok();
```

Description

Returns TRUE if the iterator is currently positioned on a valid stream. Returns FALSE if the iterator has be moved beyond the streams or no streams exist.

Examples

Example A-33

```
vmm_sb_ds_iter scan = sb.new_sb_iter();
for (scan.first(); scan.is_ok(); scan.next()) begin
   ...
end
```

vmm_sb_ds_iter::next()

Move the iterator to the next applicable stream.

SystemVerilog

```
function bit next();
```

OpenVera

```
function bit next();
```

Description

Move the iterator to the next applicable stream. Returns TRUE if a subsequent stream exists. If no subsequent stream exists, the iterator is invalidated and FALSE is returned.

If the iterator was invalid, this method is identical to calling “vmm_sb_ds_iter::first()”.

Examples

Example A-34

```
. . .
sb_iter = sb.new_sb_iter();
. . .
sb.insert(pkt);
if(sb_iter.first()) begin
   if(sb_iter.next())
      `vmm_note(log,"Stream Iterator is set to next stream
position.");
end
. . .
```

vmm_sb_ds_iter::last()

Reset the iterator to the last stream.

SystemVerilog

```
function bit last();
```

OpenVera

```
function bit last();
```

Description

Reset the iterator to the last applicable input stream. Returns TRUE if at least one such stream exists. Returns FALSE if no such stream exists.

Examples

Example A-35

```
. . .
sb_iter = sb.new_sb_iter();
. . .
sb.insert(pkt);
if(sb_iter.first()) begin
   void'(sb_iter.last());
  `vmm_note(log,$psprintf({"Stream Iterator is set at last
[%0d] stream ",
                "position."},sb_iter.pos());
end
. . .
```

vmm_sb_ds_iter::prev()

Move the iterator to the previous applicable stream.

SystemVerilog

```
function bit prev();
```

OpenVera

```
function bit prev();
```

Description

Move the iterator to the previous applicable stream. Returns TRUE if a previous stream exists. If no previous stream exists, the iterator is invalidated and FALSE is returned.

If the iterator was invalid, this method is identical to calling “vmm_sb_ds_iter::last()”.

Examples

Example A-36

```
. . .
sb_iter = sb.new_sb_iter();
. . .
sb.insert(pkt);
if(sb_iter.last()) begin
   if(sb_iter.prev())
      `vmm_note(log,"Stream Iterator is set at previous
stream position.");
end
. . .
```

vmm_sb_ds_iter::length()

Number of streams.

SystemVerilog

```
function int length();
```

OpenVera

```
function integer length();
```

Description

Return the number of streams that can be iterated on.

Examples

Example A-37

```
sb.insert(pkt);
`vmm_note(log,$psprintf("Length of the stream ::
%0d",sb_iter.length()));
. . .
```

vmm_sb_ds_iter::pos()

Position of the iterator.

SystemVerilog

```
function int pos();
```

OpenVera

```
function integer pos();
```

Description

Return the current position of the iterator. Returns -1 if the iterator is currently invalid.

Examples

Example A-38

```
. . .
sb.insert(pkt);
//First we need to reset the iterator
sb_iter.first();
`vmm_note(log,$psprintf("Position of the stream ::
%0d",sb_iter.pos()));
. . .
```

vmm_sb_ds_iter::inp_stream_id()

Input stream identifier of the current stream.

SystemVerilog

```
function int inp_stream_id();
```

OpenVera

```
function integer inp_stream_id();
```

Description

Return the input stream identifier of the stream currently iterated on. Returns -1 if there is no applicable streams available to iterate on.

Examples

Example A-39

```
. . .
sb.insert(pkt);
sb_iter.first();
`vmm_note(log,$psprintf("Input Stream Id ::
%0d",sb_iter.inp_stream_id()));
. . .
```

vmm_sb_ds_iter::exp_stream_id()

Expected stream identifier of the current stream.

SystemVerilog

```
function int exp_stream_id();
```

OpenVera

```
function integer exp_stream_id();
```

Description

Return the expected stream identifier of the stream currently iterated on. Returns -1 if there is no applicable streams available to iterate on.

Examples

Example A-40

```
. . .
sb.insert(pkt);
sb_iter.first();
`vmm_note(log,$psprintf("Expected Stream Id ::
%0d",sb_iter.exp_stream_id()));
. . .
```

vmm_sb_ds_iter::describe()

Stream description of the current stream.

SystemVerilog

```
function string describe();
```

OpenVera

```
function string describe();
```

Description

Return the description of the stream of expected packets currently iterated on.

The description depends on how the input and expected streams were defined. If they were defined as separate streams, the description will be the combination of both descriptions, as a stream of expected packet is a point-to-point packet flow. If they were defined using EITHER, a single description is returned.

If no description has been previously provided via the “vmm_sb_ds::define_stream()” method, an empty string is returned.

Examples

Example A-41

```
. . .
sb.insert(pkt);
sb_iter.first();
`vmm_note(log,$psprintf("Stream Description ::
%0s",sb_iter.describe()));
. . .
```

vmm_sb_ds_iter::get_n_inserted()

Total number of inserted EXPECT packets.

SystemVerilog

```
function int get_n_inserted()
```

OpenVera

```
function integer get_n_inserted()
```

Description

Return the total number of expected packets that have been inserted in the stream.

Examples

Example A-42

```
. . .
sb.insert(pkt);
sb_iter.first();
`vmm_note(log,$psprintf("Inserted packets ::
%0d",sb_iter.get_n_inserted()));
. . .
```

vmm_sb_ds_iter::get_n_pending()

Total number of pending packets.

SystemVerilog

```
function int get_n_pending()
```

OpenVera

```
function integer get_n_pending()
```

Description

Return the total number of pending expected packets still in the stream.

Examples

Example A-43

```
. . .
sb.insert(pkt);
sb_iter.first();
`vmm_note(log,$psprintf("Pending packets ::
%0d",sb_iter.get_n_pending()));
. . .
```

vmm_sb_ds_iter::get_n_matched()

Total number of matched packets.

SystemVerilog

```
function int get_n_matched()
```

OpenVera

```
function integer get_n_matched()
```

Description

Return the total number of matched expected packets in the stream.

Examples

Example A-44

```
. . .
sb.insert(pkt);
sb_iter.first();
`vmm_note(log,$psprintf("Matched packets ::
%0d",sb_iter.get_n_matched()));
. . .
```

vmm_sb_ds_iter::get_n_mismatched()

Total number of mis-matched packets.

SystemVerilog

```
function int get_n_mismatched()
```

OpenVera

```
function integer get_n_mismatched()
```

Description

Return the total number of mismatched expected packets as identified by the “vmm_sb_ds::expect_with losses()” method.

Examples

Example A-45

```
. . .
sb.insert(pkt);
sb_iter.first();
`vmm_note(log,$psprintf("Mismatched packets ::
%0d",sb_iter.get_n_mismatched()));
. . .
```

vmm_sb_ds_iter::get_n_dropped()

Total number of dropped packets.

SystemVerilog

```
function int get_n_dropped()
```

OpenVera

```
function integer get_n_dropped()
```

Description

Return the total number of packets assumed dropped.

Examples

Example A-46

```
. . .
sb.insert(pkt);
sb_iter.first();
`vmm_note(log,$psprintf("Dropped ackets ::
%0d",sb_iter.get_n_dropped());
. . .
```

vmm_sb_ds_iter::get_n_not_found()

Total number of unexpected packets.

SystemVerilog

```
function int get_n_not_found()
```

OpenVera

```
function integer get_n_not_found()
```

Description

Return the total number of packets that were not found in the stream and thus assumed to be unexpected.

Examples

Example A-47

```
. . .
sb.insert(pkt);
sb_iter.first();
`vmm_note(log,$psprintf("Not Found packets ::
%0d",sb_iter.get_n_not_found()));
. . .
```

vmm_sb_ds_iter::get_n_orphaned()

Total number of leftover packets.

SystemVerilog

```
function int get_n_orphaned()
```

OpenVera

```
function integer get_n_orphaned()
```

Description

Return the total number of expected packets remaining in the stream.

Examples

Example A-48

```
. . .
sb.insert(pkt);
sb_iter.first();
`vmm_note(log,$psprintf("Orphaned packets ::
%0d",sb_iter.get_n_orphaned()));
. . .
```

vmm_sb_ds_iter::incr_n_inserted()

Adjust the total number of inserted EXPECT packets.

SystemVerilog

```
function int incr_n_inserted(int delta)
```

OpenVera

```
function integer incr_n_inserted(integer delta)
```

Description

Adjust and return the total number of expected packets that have been inserted in the stream by the specified value. Specify a negative adjustment value to decrease the total number. The final adjusted value cannot be less than zero.

Examples

Example A-49

```
. . .
int adjust;
sb.insert(pkt);
sb_iter.first();
`vmm_note(log,$psprintf("Adjusted Inserted packet :: %0d",
              sb_iter.incr_n_inserted()));
. . .
```

vmm_sb_ds_iter::incr_n_matched()

Adjust the total number of matched packets.

SystemVerilog

```
function int incr_n_matched(int delta)
```

OpenVera

```
function integer incr_n_matched(int delta)
```

Description

Adjust and return the total number of matched expected packets in the stream. Specify a negative adjustment value to decrease the total number. The final adjusted value cannot be less than zero.

Examples

Example A-50

```
. . .
int adjust;
sb.insert(pkt);
sb_iter.first();
`vmm_note(log,$psprintf("Adjusted Matched packet :: %0d",
              sb_iter.incr_n_matched()));
. . .
```

vmm_sb_ds_iter::incr_n_mismatched()

Adjust the total number of mis-matched packets.

SystemVerilog

```
function int incr_n_mismatched(int delta)
```

OpenVera

```
function integer incr_n_mismatched(integer delta)
```

Description

Adjust and return the total number of mismatched expected packets as identified by the “vmm_sb_ds::expect_with losses()” method. Specify a negative adjustment value to decrease the total number. The final adjusted value cannot be less than zero.

Examples

Example A-51

```
. . .
int adjust;
sb.insert(pkt);
sb_iter.first();
`vmm_note(log,$psprintf("Adjusted Mismatched packet ::
%0d",
                sb_iter.incr_n_mismatched()));
. . .
```

vmm_sb_ds_iter::incr_n_dropped()

Adjust the total number of dropped packets.

SystemVerilog

```
function int incr_n_dropped(int delta)
```

OpenVera

```
function integer incr_n_dropped(integer delta)
```

Description

Adjust and return the total number of packets assumed dropped. Specify a negative adjustment value to decrease the total number. The final adjusted value cannot be less than zero.

Examples

Example A-52

```
. . .
int adjust;
sb.insert(pkt);
sb_iter.first();
`vmm_note(log,$psprintf("Adjusted Dropped packet :: %0d",
              sb_iter.incr_n_dropped()));
. . .
```

vmm_sb_ds_iter::incr_n_not_found()

Adjust the total number of unexpected packets.

SystemVerilog

```
function int incr_n_not_found(int delta)
```

OpenVera

```
function integer incr_n_not_found(integer delta)
```

Description

Adjust and return the total number of packets that were not found in the stream and thus assumed to be unexpected. Specify a negative adjustment value to decrease the total number. The final adjusted value cannot be less than zero.

Examples

Example A-53

```
. . .
int adjust;
sb.insert(pkt);
sb_iter.first();
`vmm_note(log,$psprintf("Adjusted Not Found packet :: %0d",
              sb_iter.incr_n_not_found()));
. . .
```

vmm_sb_ds_iter::copy()

Create a copy of this iterator.

SystemVerilog

```
function vmm_sb_ds_iter copy();
```

OpenVera

```
function vmm_sb_ds_iter copy();
```

Description

Return a copy of this iterator positioned on the same stream and with the same applicable stream configuration.

Examples

Example A-54

```
. . .
vmm_sb_ds_iter sb_iter;
vmm_sb_ds_iter sb_cpy;
. . .
sb_iter = sb.new_sb_iter();
sb_iter.first();
sb_cpy = sb_iter.copy();

`vmm_note(log,$psprintf("Get Inserted packet from copied
iterator:: %0d",
              sb_cpy.get_n_inserted()));
. . .
```

vmm_sb_ds_iter::stream_iter

Stream iterator.

SystemVerilog

```
vmm_sb_ds_stream_iter stream_iter;
```

OpenVera

```
vmm_sb_ds_stream_iter stream_iter;
```

Description

Pre-existing stream iterator to iterate on the packets in the stream this scoreboard iterator is currently on.

The stream iterator is invalidated every time the scoreboard iterator is moved.

Examples

Example A-55

```
for (scan_sb.first(); scan_sb.is_ok(); scan_sb.next())
begin
   if (scan_sb.exp_stream_id() < 10)
scan_sb.stream_iter.flush();
end
```

vmm_sb_ds_iter::new_stream_iter()

Create a stream iterator.

SystemVerilog

```
function vmm_sb_ds_stream_iter new_stream_iter();
```

OpenVera

```
function vmm_sb_ds_stream_iter new_stream_iter();
```

Description

Create and return a stream iterator on the stream being iterated on by this scoreboard iterator. Returns NULL of this scoreboard iterator is not currently on a valid stream.

Examples

Example A-56

```
. . .
vmm_sb_ds_iter sb_iter;
. . .
sb_iter = sb.new_sb_iter();
sb_iter.first();
sb_iter.stream_iter = sb_iter.new_stream_iter();
if (sb_iter.stream_iter == null)
   `vmm_error(log,("vmm_sb_ds_iter::new_stream_iter()
method is not working.");
. . .
```

vmm_sb_ds_iter::delete()

Delete the stream.

SystemVerilog

```
function int delete();
```

OpenVera

```
function integer delete();
```

Description

Flush the content of the stream being iterated on by this iterator and remove the stream from the list of known streams. Use “vmm_sb_ds::define_stream()” to create (or re-create) a stream.

Returns the number of packets that were flushed and moves the iterator to the next stream. Returns -1 if the iterator is not on a valid stream.

The effect on other existing iterators is undefined.

Examples

Example A-57

```
. . .
sb.insert(pkt);
sb_iter.first();
. . .
`vmm_note(log,$psprintf("Inserted Packets (Before Deleting
one stream):: %0d",
                sb_iter.get_n_inserted()));
sb_iter.delete();
```

```
`vmm_note(log,$psprintf("Inserted Packets (After Deleting
one stream):: %0d",
              sb_iter.get_n_inserted()));
. . .
```

vmm_sb_ds_iter::display()

Dump the content of the stream.

SystemVerilog

```
function void display(string prefix = "");
```

OpenVera

```
task display(string prefix = "");
```

Description

Dump the content of the stream iterated on the standard output in a human-readable format.

Examples

Example A-58

```
. . .
sb.insert(pkt);
sb_iter.first();
. . .
`vmm_note(log,"Contents Of the Packet::");
sb_iter.display();
. . .
```

vmm_sb_ds_stream_iter

This class implements an iterator that will iterate over all pending expected packets in a stream.

This class is used in combination with the "vmm_sb_ds_iter" class to walk the content of the scoreboard to perform a user-defined checking operation.

Instances of this class are created by the "vmm_sb_ds::new_stream_iter()" or "vmm_sb_ds_iter::new_stream_iter()" method. Iterators are created in the invalid state.

Packets are iterated on from first inserted to most-recently inserted.

Summary

vmm_sb_ds_stream_iter::first()

Reset the iterator to the first packet.

SystemVerilog

```
function bit first();
```

OpenVera

```
function bit first();
```

Description

Reset the iterator to the first packet in the stream. Returns TRUE if at least one such packet exists. Returns FALSE if no such packet exists.

Examples

Example A-59

```
. . .
sb_iter.stream_iter = sb.new_stream_iter();
sb.insert(pkt);
sb_iter.stream_iter.append(pkt);
if(sb_iter.stream_iter.first());
   `vmm_note(log,"Stream Iterator is reseted and set at the
1st stream position.");
. . .
```

vmm_sb_ds_stream_iter::is_ok()

Check if iterator is on a valid expected packet.

SystemVerilog

```
function bit is_ok();
```

OpenVera

```
function bit is_ok();
```

Description

Returns TRUE if the iterator is currently positioned on a valid expected packet. Returns FALSE if the iterator has be moved beyond the packets or no packets exist.

Examples

Example A-60

```
. . .
sb.insert(pkt);
sb_iter.stream_iter.append(pkt);
if(sb_iter.stream_iter.is_ok());
   `vmm_note(log,"Stream Iterator is on valid stream.");
. . .
```

vmm_sb_ds_stream_iter::next()

Move the iterator to the next packet in the stream.

SystemVerilog

```
function bit next();
```

OpenVera

```
function bit next();
```

Description

Move the iterator to the next packet in the stream. Returns TRUE if a subsequent packet exists. If no subsequent packet exists, the iterator is not moved and FALSE is returned.

If the iterator was invalid, this method is identical to calling “vmm_sb_ds_stream_iter::first()”.

Examples

Example A-61

```
. . .
sb.insert(pkt);
sb_iter.stream_iter.append(pkt);
sb_iter.stream_iter.first();
if(sb_iter.stream_iter.next());
   `vmm_note(log,$psprintf({"Stream Iterator is on %0d
position and next stream ",
             "is valid."},sb_iter.stream_iter.pos()));
. . .
```

vmm_sb_ds_stream_iter::last()

Reset the iterator to the last packet.

SystemVerilog

```
function bit last();
```

OpenVera

```
function bit last();
```

Description

Reset the iterator to the last packet in the stream. Returns TRUE if at least one such packet exists. Returns FALSE if no such packet exists.

Examples

Example A-62

```
. . .
sb.insert(pkt);
sb_iter.stream_iter.append(pkt);
if(sb_iter.stream_iter.last());
   `vmm_note(log,$psprintf({"Stream Iterator is on %0d
position and last stream ",
             "is valid."},sb_iter.stream_iter.pos()));
. . .
```

vmm_sb_ds_stream_iter::prev()

Move the iterator to the previous packet.

SystemVerilog

```
function bit prev();
```

OpenVera

```
function bit prev();
```

Description

Move the iterator to the previous packet in the stream. Returns TRUE if a previous packet exists. If no previous packet exists, the iterator is invalidated and FALSE is returned.

If the iterator was invalid, this method is identical to calling “vmm_sb_ds_stream_iter::first()”.

Examples

Example A-63

```
. . .
sb.insert(pkt);
sb_iter.stream_iter.append(pkt);
sb_iter.stream_iter.last();
if(sb_iter.stream_iter.prev());
   `vmm_note(log,$psprintf({"Stream Iterator is on %0d
position and previous ",
           "stream is valid."},sb_iter.stream_iter.pos()));
. . .
```

vmm_sb_ds_stream_iter::inp_stream_id()

Input stream identifier of the stream.

SystemVerilog

```
function int inp_stream_id();
```

OpenVera

```
function integer inp_stream_id();
```

Description

Return the input stream identifier of the stream.

Examples

Example A-64

```
. . .
sb.insert(pkt);
sb_iter.stream_iter.append(pkt);
sb_iter.stream_iter.first();
`vmm_note(log,$psprintf("Current packet inp_stream_id ::
%0d",
          sb_iter.stream_iter.inp_stream_id()));
. . .
```

vmm_sb_ds_stream_iter::exp_stream_id()

Expected stream identifier of the stream.

SystemVerilog

```
function int exp_stream_id();
```

OpenVera

```
function integer exp_stream_id();
```

Description

Return the expected stream identifier of the stream.

Examples

Example A-65

```
. . .
sb.insert(pkt);
sb_iter.stream_iter.append(pkt);
sb_iter.stream_iter.first();
`vmm_note(log,$psprintf("Current packet exp_stream_id ::
%0d",
          sb_iter.stream_iter.exp_stream_id()));
. . .
```

vmm_sb_ds_stream_iter::describe()

Stream description of the stream.

SystemVerilog

```
function string describe();
```

OpenVera

```
function string describe();
```

Description

Return the description of the stream. If no description has been previously provided via the “vmm_sb_ds::define_stream()” method, an empty string is returned.

Examples

Example A-66

```
. . .
sb.insert(pkt);
sb_iter.stream_iter.append(pkt);
sb_iter.stream_iter.first();
//Note :: Always return "No Description String"
`vmm_note(log,$psprintf("Packet Description :: %0s",
          sb_iter.stream_iter.describe()));
. . .
```

vmm_sb_ds_stream_iter::length()

Number of pending expected packets in the stream.

SystemVerilog

```
function int length();
```

OpenVera

```
function integer length();
```

Description

Return the number of packets in the stream.

Examples

Example A-67

```
. . .
sb.insert(pkt);
sb_iter.stream_iter.append(pkt);
sb_iter.stream_iter.first();
`vmm_note(log,$psprintf("Total packets ::
%0d",sb_iter.stream_iter.length()));
. . .
```

vmm_sb_ds_stream_iter::data()

The packet currently iterated on.

SystemVerilog

```
function vmm_data data();
```

OpenVera

```
function rvm_data data();
```

Description

Return the packet in the stream currently iterated on. Returns NULL if the iterator is not on a valid packet.

Examples

Example A-68

```
. . .
vmm_data curr_pkt;
my_data pkt_t;
sb.insert(pkt);
sb_iter.stream_iter.append(pkt);
sb_iter.stream_iter.first();
curr_pkt = sb_iter.stream_iter.data();
$cast(pkt_t,curr_pkt);
`vmm_note(log,$psprintf("%0s",pkt_t.psdisplay()));
. . .
```

vmm_sb_ds_stream_iter::pos()

Position of the iterator in the stream.

SystemVerilog

```
function int pos();
```

OpenVera

```
function integer pos();
```

Description

Return the position of the iterator in the stream. Returns 0 if it is on the first packet on the stream. Returns "vmm_sb_ds_stream_iter::length()"-1 if it is on the last packet in the stream. Returns -1 if the stream does not contain any packets.

Examples

Example A-69

```
. . .
sb.insert(pkt);
sb_iter.stream_iter.append(pkt);
sb_iter.stream_iter.first();
`vmm_note(log,$psprintf("Stream Iterator is on %0d
position",
          sb_iter.stream_iter.pos()));
. . .
```

vmm_sb_ds_stream_iter::find()

Find a packet in the stream.

SystemVerilog

```
function bit find(vmm_data pkt);
```

OpenVera

```
function bit find(rvm_data pkt);
```

Description

Locate the next packet matching the specified packet upward in the stream, starting with the packet currently iterated on. Returns TRUE if a matching packet exists and reposition the iterator on that packet. Returns FALSE and do not move the iterator otherwise.

Examples

Example A-70

```
. . .
sb.insert(pkt);
sb_iter.stream_iter.append(pkt);
sb_iter.stream_iter.first();
        if(sb_iter.stream_iter.find(pkt))
`vmm_note(log,$psprintf("Search Packet is on %0d position",
          sb_iter.stream_iter.pos()));
. . .
```

vmm_sb_ds_stream_iter::prepend()

Prepend a packet in the stream.

SystemVerilog

```
function void prepend(vmm_data pkt);
```

OpenVera

```
task prepend(rvm_data pkt);
```

Description

Insert the specified packet before the packet currently being iterated on. The position of the iterator is not modified.

If the iterator is in an invalid state, the packet is inserted at the beginning of the stream.

The effect on existing iterators on the same stream is undefined.

Examples

Example A-71

```
. . .
sb.insert(pkt);
sb_iter.stream_iter.prepend(pkt); //insert at the beginning
of the stream
`vmm_note(log,$psprintf("Current inserted Packet is on %0d
position",
          sb_iter.stream_iter.pos()));
. . .
```

vmm_sb_ds_stream_iter::append()

Append a packet in the stream.

SystemVerilog

```
function void append(vmm_data pkt);
```

OpenVera

```
task append(rvm_data pkt);
```

Description

Insert the specified packet after the packet currently being iterated on. The position of the iterator is not modified.

If the iterator is in an invalid state, the packet is added at the end of the stream.

The effect on existing iterators on the same stream is undefined.

Examples

Example A-72

```
. . .
sb.insert(pkt);
sb_iter.stream_iter.append(pkt); //insert at the end of the
stream
`vmm_note(log,$psprintf("Current inserted Packet is on %0d
position",
          sb_iter.stream_iter.pos()));
. . .
```

vmm_sb_ds_stream_iter::delete()

Delete the packet currently iterated on.

SystemVerilog

```
function vmm_data delete();
```

OpenVera

```
function rvm_data delete();
```

Description

Remove the packet currently iterated on and return it. The iterator is then moved to the next packet in the stream. Returns NULL if the iterator is not on a valid packet.

The effect on existing iterators on the same stream is undefined.

Examples

Example A-73

```
. . .
sb.insert(pkt);
sb_iter.stream_iter.append(pkt);
sb_iter.stream_iter.first();
del_pkt = sb_iter.stream_iter.delete();
`vmm_note(log,$psprintf("Deleted
Packet::%0s",del_pkt.psdisplay()));
. . .
```

vmm_sb_ds_stream_iter::flush()

Flush the stream.

SystemVerilog

```
function int flush();
```

OpenVera

```
function integer flush();
```

Description

Flush the content of the stream. Returns the number of packets that were flushed.

The effect on existing iterators on the same stream is undefined.

Examples

Example A-74

```
. . .
//Refer Example A-60
sb_iter.stream_iter.first();
flush_pkt = sb_iter.stream_iter.flush();
`vmm_note(log,$psprintf("Flushed Packet::%0d",flush_pkt));
. . .
```

vmm_sb_ds_stream_iter::preflush()

Flush previous packets.

SystemVerilog

```
function int preflush();
```

OpenVera

```
function integer preflush();
```

Description

Flush the packets that are before the packet currently iterated on in the stream. Returns the number of packets that were flushed. Returns -1 if the iterator is not on a valid packet.

The effect on existing iterators on the same stream is undefined.

Examples

Example A-75

```
. . .
//Refer Example A-60
sb_iter.stream_iter.first();
flush_pkt = sb_iter.stream_iter.preflush();
`vmm_note(log,$psprintf("Flushed Packet::%0d",flush_pkt));
`vmm_note(log,$psprintf("Now Total
Packets::%0d",sb_iter.stream_iter.length));
. . .
```

vmm_sb_ds_stream_iter::postflush()

Flush subsequent packets.

SystemVerilog

```
function int postflush();
```

OpenVera

```
function integer postflush();
```

Description

Flush the packets that are after the packet currently iterated on in the stream. Returns the number of packets that were flushed. Returns -1 if the iterator is not on a valid packet.

The effect on existing iterators on the same stream is undefined.

Examples

Example A-76

```
. . .
//Refer Example A-60
sb_iter.stream_iter.first();
flush_pkt = sb_iter.stream_iter.postflush();
`vmm_note(log,$psprintf("Flushed Packet::%0d",flush_pkt));
`vmm_note(log,$psprintf("Now Total
Packets::%0d",sb_iter.stream_iter.length));
. . .
```

vmm_sb_ds_stream_iter::copy()

Create a copy of this iterator.

SystemVerilog

```
function vmm_sb_ds_stream_iter copy();
```

OpenVera

```
function vmm_sb_ds_stream_iter copy();
```

Description

Return a copy of this iterator positioned on the same packet in the stream.

Examples

Example A-77

```
. . .
//Refer Example A-60
vmm_sb_ds_stream_iter cpy;
cpy = sb_iter.stream_iter.copy();
`vmm_note(log,$psprintf("Total Packets of Copied
Stream::%0d",cpy.length));
. . .
```

vmm_sb_ds_callbacks

This class is the facade for the callback methods available in the "vmm_sb_ds" class.

The documentation for this class uses the term "packet" to describe a data item inserted or checked in the scoreboard. The term is used for as a convenience as does not imply that the class is limited to data streams composed of packets. It is suitable for any stream of data, composed of frames, fragments, bus cycles, transfers, etc...

Summary

vmm_sb_ds_callbacks::pre_insert()

Pre-insertion callback method.

SystemVerilog

```
function void pre_insert(input vmm_sb_ds          sb,
     input vmm_data           pkt,
     input vmm_sb_ds::kind_e kind,
     ref   int               exp_stream_id,
     ref   int               inp_stream_id,
     ref   bit               drop);
```

OpenVera

```
task pre_insert(vmm_sb_ds          sb,
     rvm_data           pkt,
     vmm_sb_ds::kind_e kind,
     var integer        exp_stream_id,
     var integer        inp_stream_id
     var bit            drop);
```

Description

This callback method is called whenever a packet is inserted in the scoreboard. For an "INPUT" packet, this method is called before the packet is transformed into an expected packet and the value of "exp_stream_id" is invalid (-1). For an "EXPECT" packet, this method is called before the packet is actually inserted in the appropriate queue of expected packets.

Modifying the input or expected stream identifiers will cause the packet to be inserted in a different queue of expected packets. The value of "drop" is initialized to FALSE. If it is set to TRUE by a callback extension, the insertion process is aborted.

For "INPUT" packets, this method will be called twice: once as in "INPUT" packet and a second time as an "EXPECT" packet after the input packet has been transformed.

This callback method is called only if the “vmm_sb_ds::insert()” method is used. It is not called if a packet is inserted directly into a stream using the “vmm_sb_ds_stream_iter::prepend()” or “vmm_sb_ds_stream_iter::append()” methods.

Examples

Example A-78

```
program my_test;
  class my_sb_ds_callbacks extends vmm_sb_ds_callbacks;
     . . .
   virtual function void pre_insert(input vmm_sb_ds      sb,
                                    input vmm_data        pkt,
                                  input vmm_sb_ds::kind_e kind,
                             ref  int           exp_stream_id,
                             ref  int           inp_stream_id,
                                  ref  bit             drop);
          `vmm_note(log,"pre_insert Method is called for
vmm_sb_ds_callbacks");
          . . .
     endfunction
     . . .
  endclass

  initial
  begin
     my_sb_ds_callbacks cb;
     my_env env = new;
     . . .
     cb = new();
     env.sb.append_callback(cb);
     . . .
     env.run();
  end
```

```
endprogram
. . .
//This will call pre_insert and post_insert method
sb.insert(pkt,vmm_sb_ds::INPUT,ip_id,exp_id);
. . .
```

vmm_sb_ds_callbacks::post_insert()

Post-insertion callback method.

SystemVerilog

```
function void post_insert(vmm_sb_ds sb,
     vmm_data  pkt,
     int       exp_stream_id,
     int       inp_stream_id);
```

OpenVera

```
task post_insert(vmm_sb_ds sb,
     rvm_data  pkt,
     integer   exp_stream_id,
     integer   inp_stream_id);
```

Description

This callback method is called after an expected packet has been inserted in the scoreboard.

This callback method is called only if the "vmm_sb_ds::insert()" method is used. It is not called if a packet is inserted directly into a stream using the "vmm_sb_ds_stream_iter::prepend()" or "vmm_sb_ds_stream_iter::append()" methods.

Examples

Example A-79

```
program my_test;
  class my_sb_ds_callbacks extends vmm_sb_ds_callbacks;
     . . .
     virtual function void post_insert(vmm_sb_ds sb,
                                       vmm_data  pkt,
```

```
                                        int       exp_stream_id,
                                       int       inp_stream_id);
         `vmm_note(log,"post_insert Method is called for
vmm_sb_ds_callbacks");
         . . .
     endfunction
     . . .
  endclass

  initial
  begin
     my_sb_ds_callbacks cb;
     my_env env = new;
     . . .
     cb = new();
     env.sb.append_callback(cb);
     . . .
     env.run();
  end
endprogram
. . .
//This will call pre_insert and post_insert method
sb.insert(pkt,vmm_sb_ds::INPUT,ip_id,exp_id);
. . .
```

vmm_sb_ds_callbacks::matched()

Matched packet callback method.

SystemVerilog

```
function void matched(input  vmm_sb_ds sb,
     input  vmm_data  pkt,
     input  int       exp_stream_id,
     input  int       inp_stream_id,
     ref    int       count);
```

OpenVera

```
task matched(vmm_sb_ds   sb,
     rvm_data    pkt,
     integer     exp_stream_id,
     integer     inp_stream_id
     var integer count);
```

Description

This callback method is called after a packet has been matched and removed from the scoreboard.

The value of "count" is initialized to 1. Its final value, once the callback methods have been called, is used to increment the matched packet counter.

This callback method is called only if one of the "vmm_sb_ds::expect_in_order()", "vmm_sb_ds::expect_with losses()" or "vmm_sb_ds::expect_out_of_order()" methods is used. Any user-defined expect function should also invoke this callback method explicitly when a matching packet is found and removed.

Examples

Example A-80

```
program my_test;
  class my_sb_ds_callbacks extends vmm_sb_ds_callbacks;
     . . .
     virtual function void matched(input vmm_sb_ds sb,
                                   input vmm_data  pkt,
                                input int      exp_stream_id,
                                input int      inp_stream_id,
                                ref   int       count);
         `vmm_note(log,$psprintf("Matched Packet Count ::
%0d",count));
         . . .
     endfunction
     . . .
  endclass

  initial
  begin
     my_sb_ds_callbacks cb;
     my_env env = new;
     . . .
     cb = new();
     env.sb.append_callback(cb);
     . . .
     env.run();
  end
endprogram
. . .
sb.insert(pkt,vmm_sb_ds::INPUT,ip_id,exp_id);
. . .
exp_pkt = sb.expect_in_order(pkt,ip_id,exp_id);
//If packet is matched then called matched() method of
callback
. . .
```

vmm_sb_ds_callbacks::mismatched()

Mismatched packet callback method.

SystemVerilog

```
function void mismatched(input  vmm_sb_ds sb,
     input  vmm_data  pkt,
     input  int       exp_stream_id,
     input  int       inp_stream_id,
     ref    int       count);
```

OpenVera

```
task mismatched(vmm_sb_ds   sb,
     rvm_data    pkt,
     integer     exp_stream_id,
     integer     inp_stream_id
     var integer count);
```

Description

This callback method is called after a packet has been mismatched and removed from the scoreboard.

The value of "count" is initialized to 1. Its final value, once the callback methods have been called, is used to increment the mismatched packet counter.

This callback method is called only if the “vmm_sb_ds::expect_with losses()” method is used. Any user-defined expect function should also invoke this callback method explicitly when a mismatching packet is found and removed.

Examples

Example A-81

```
program my_test;
  class my_sb_ds_callbacks extends vmm_sb_ds_callbacks;
     . . .
     virtual function void mismatched(input vmm_sb_ds sb,
                                      input vmm_data  pkt,
                                input int      exp_stream_id,
                                input int      inp_stream_id,
                                      ref   int       count);
         `vmm_note(log,$psprintf("Mismatched Packet Count
:: %0d",count));
          . . .
     endfunction
     . . .
  endclass

  initial
  begin
     my_sb_ds_callbacks cb;
     my_env env = new;
     . . .
     cb = new();
     env.sb.append_callback(cb);
     . . .
     env.run();
  end
endprogram
. . .
sb.insert(pkt,vmm_sb_ds::INPUT,ip_id,exp_id);
. . .
exp_pkt =
sb.expect_with_losses(pkt,matched,lost,ip_id,exp_id);
//If packet is mismatched then called mismatched() method
of callback
. . .
```

vmm_sb_ds_callbacks::dropped()

Dropped packet(s) callback method.

SystemVerilog

```
function void matched(input  vmm_sb_ds sb,
     input  vmm_data  pkts[],
     input  int       exp_stream_id,
     input  int       inp_stream_id,
     ref    int       count);
```

OpenVera

```
task matched(vmm_sb_ds   sb,
     rvm_data    pkts[],
     integer     exp_stream_id,
     integer     inp_stream_id
     var integer count);
```

Description

This callback method is called after one more packets have been assumed lost and removed from the scoreboard. This method is not called if no packets are assumed lost.

The value of "count" is initialized to the number of lost packets. Its final value once the callback methods have been called is used to increment the lost packet counter.

This callback method is called only if the “vmm_sb_ds::expect_with losses()” method is used. Any user-defined expect function should also invoke this callback method explicitly when packets are assumed to have been lost.

Examples

Example A-82

```
program my_test;
  class my_sb_ds_callbacks extends vmm_sb_ds_callbacks;
     . . .
     virtual function void dropped(input vmm_sb_ds sb,
                                   input vmm_data  pkts[],
                                input int      exp_stream_id,
                                input int      inp_stream_id,
                                   ref   int       count);
         `vmm_note(log,$psprintf("Dropped Packet Count ::
%0d",count));
         . . .
     endfunction
     . . .
  endclass

  initial
  begin
     my_sb_ds_callbacks cb;
     my_env env = new;
     . . .
     cb = new();
     env.sb.append_callback(cb);
     . . .
     env.run();
  end
endprogram
. . .
sb.insert(pkt,vmm_sb_ds::INPUT,ip_id,exp_id);
. . .
exp_pkt =
sb.expect_with_losses(pkt,matched,lost,ip_id,exp_id);
//If packet is dropped then called dropped() method of
callback
. . .
```

vmm_sb_ds_callbacks::not_found()

Packet not found callback method.

SystemVerilog

```
function void not_found(input  vmm_sb_ds sb,
     input  vmm_data  pkt,
     input  int       exp_stream_id,
     input  int       inp_stream_id,
     ref    int       count);
```

OpenVera

```
task not_found(vmm_sb_ds   sb,
     rvm_data    pkt,
     integer     exp_stream_id,
     integer     inp_stream_id
     var integer count);
```

Description

This callback method is called after an observed packet has not been found in the scoreboard.

The value of "count" is initialized to 1. Its final value once the callback methods have been called is used to increment the packet not found counter.

This callback method is called only if one of the "vmm_sb_ds::expect_in_order()", "vmm_sb_ds::expect_with losses()" or "vmm_sb_ds::expect_out_of_order()" methods is used. Any user-defined expect function should also invoke this callback method explicitly when an observed packet is not found.

Examples

Example A-83

```
program my_test;
  class my_sb_ds_callbacks extends vmm_sb_ds_callbacks;
     . . .
     virtual function void not_found(input vmm_sb_ds sb,
                                     input vmm_data  pkt,
                                input int      exp_stream_id,
                                input int      inp_stream_id,
                                      ref   int       count);
        `vmm_note(log,$psprintf("Not Found Packet Count ::
%0d",count));
         . . .
     endfunction
     . . .
  endclass

  initial
  begin
     my_sb_ds_callbacks cb;
     my_env env = new;
     . . .
     cb = new();
     env.sb.append_callback(cb);
     . . .
     env.run();
  end
endprogram
. . .
sb.insert(pkt,vmm_sb_ds::INPUT,ip_id,exp_id);
. . .
exp_pkt = sb.expect_in_order(pkt,ip_id,exp_id);
//If packet is not found then called not_found() method of
callback
. . .
```

vmm_sb_ds_callbacks::orphaned()

Orphaned packet(s) callback method.

SystemVerilog

```
function void orphaned(input  vmm_sb_ds sb,
     input  vmm_data  pkts[],
     input  int       exp_stream_id,
     input  int       inp_stream_id,
     ref    int       count);
```

OpenVera

```
task orphaned(vmm_sb_ds   sb,
     rvm_data    pkts[],
     integer     exp_stream_id,
     integer     inp_stream_id
     var integer count);
```

Description

This callback method is called for each expected packet queue where one more packets have been left over in the scoreboard. This method is not called if no packets are orphaned.

The value of "count" is initialized to the number of orphaned packets. Its final value once the callback methods have been called is used to increment the orphaned packet counter.

This callback method is called only the first time the “vmm_sb_ds::get_n_orphaned()” method is used.

Examples

Example A-84

```
program my_test;
  class my_sb_ds_callbacks extends vmm_sb_ds_callbacks;
     . . .
     virtual function void orphaned(input vmm_sb_ds sb,
                                    input vmm_data  pkts[],
                                input int      exp_stream_id,
                                input int      inp_stream_id,
                                    ref   int       count);
        `vmm_note(log,$psprintf("Orphaned Packet Count ::
%0d",count));
         . . .
     endfunction
     . . .
  endclass

  initial
  begin
     my_sb_ds_callbacks cb;
     my_env env = new;
     . . .
     cb = new();
     env.sb.append_callback(cb);
     . . .
     env.run();
  end
endprogram
. . .
sb.insert(pkt,vmm_sb_ds::INPUT,ip_id,exp_id);
. . .
$psprintf("Get_n_orphaned : %0d method of
vmm_sb_ds",sb.get_n_orphaned()));
. . .
```

vmm_sb_ds_pkts

This class is used to describe one or more packets. Instances of this class are used as status information in notifications indicated through the “vmm_sb_ds::notify” property.

The documentation for this class uses the term "packet" to describe a data item inserted or checked in the scoreboard. The term is used for as a convenience as does not imply that the class is limited to data streams composed of packets. It is suitable for any stream of data, composed of frames, fragments, bus cycles, transfers, etc...

Summary

vmm_sb_ds_pkts::pkts

The packet(s) in question.

SystemVerilog

```
vmm_data pkts[$];
```

OpenVera

```
rvm_data pkts[$];
```

Description

The packet(s) that caused the notification to be indicated.

Examples

Example A-85

```
. . .
vmm_sb_ds_pkts sb_pkts;
vmm_data my_data;
my_pkt p;
sb.insert(pkt);
sb_pkts = new(pkt,vmm_sb_ds::INPUT,ip_id,exp_id);
my_data = sb_pkts.pkts[0];
$cast(p,my_data);
`vmm_note(log,$psprintf("vmm_sb_ds_pkts::pkts ==>
%0s",p.psdisplay()));
. . .
```

vmm_sb_ds_pkts::kind

The kind of packet(s) in question.

SystemVerilog

```
vmm_sb_ds::kind_e kind;
```

OpenVera

```
vmm_sb_ds::kind_e kind;
```

Description

The kind of packet(s) that caused the notification to be indicated.

Examples

Example A-86

```
. . .
vmm_sb_ds_pkts sb_pkts;
sb.insert(pkt);
sb_pkts = new(pkt,vmm_sb_ds::INPUT,ip_id,exp_id);
`vmm_note(log,$psprintf("vmm_sb_ds_pkts::kind ==>
%0s",sb_pkts.kind.name));
. . .
```

vmm_sb_ds_pkts::inp_stream_id

The input stream identifier of the packet(s) in question.

SystemVerilog

```
int inp_stream_id;
```

OpenVera

```
integer inp_stream_id;
```

Description

The input stream identifier of the packet(s) that caused the notification to be indicated.

Examples

Example A-87

```
. . .
vmm_sb_ds_pkts sb_pkts;
sb.insert(pkt);
sb_pkts = new(pkt,vmm_sb_ds::INPUT,ip_id,exp_id);
`vmm_note(log,$psprintf("vmm_sb_ds_pkts::inp_stream_id ==>
%0s",
          sb_pkts.inp_stream_id));
. . .
```

vmm_sb_ds_pkts::exp_stream_id

The expected stream identifier of the packet(s) in question.

SystemVerilog

```
int exp_stream_id;
```

OpenVera

```
integer exp_stream_id;
```

Description

The expected stream identifier of the packet(s) that caused the notification to be indicated.

Examples

Example A-88

```
. . .
vmm_sb_ds_pkts sb_pkts;
sb.insert(pkt);
sb_pkts = new(pkt,vmm_sb_ds::INPUT,ip_id,exp_id);
`vmm_note(log,$psprintf("vmm_sb_ds_pkts::exp_stream_id ==>
%0s",
          sb_pkts.exp_stream_id));
. . .
```

vmm_channel, vmm_notify, vmm_xactor

The following methods have been added to several components in the VMM Standard Library to facilitate the integration of data stream scoreboards with various verification environment components.

To eliminate the need for loading the scoreboard package code if it is not required, the following methods are only visible if the ‘VMM_SB_DS_IN_STDLIB symbol is defined

Summary

vmm_channel::register_vmm_sb_ds()

Register a data stream scoreboarde with a channel instance.

SystemVerilog

```
function void register_vmm_sb_ds(
     vmm_sb_ds              sb,
     vmm_sb_ds::kind_e      kind,
     vmm_sb_ds::ordering_e order = IN_ORDER);
```

OpenVera

```
task register_vmm_sb_ds(vmm_sb_ds              sb,
     vmm_sb_ds::kind_e      kind,
     vmm_sb_ds::ordering_e order = IN_ORDER);
```

Description

Register a data stream scoreboard with a channel instance with the specified direction and ordering. Transactions are automatically forwarded to all registered scoreboards when they are removed from the channel.

If *direction* is specified as `vmm_sb_ds::INPUT`, transactions will be automatically forwarded to the scoreboard by the channel using the "vmm_sb_ds::insert()" method.

If *direction* is specified as `vmm_sb_ds::EXPECT`, transactions will be automatically forwarded to the scoreboard using the method and the value specified for *order*, as shown in Table A-1:

Table A-1

Ordering	Method Called
`vmm_sb_ds::IN_ORDER`	`"vmm_sb_ds::expect_in_order()"`
`vmm_sb_ds::WITH_LOSSES`	`"vmm_sb_ds::expect_with losses()"`
`vmm_sb_ds::OUT_ORDER`	`"vmm_sb_ds::expect_out_of_order()"`

Example

Example A-89

```
class my_env extends vmm_env;
   . . .
   virtual function build();
      super.build();
      . . .
      this.my_xactor.out_chan.register_vmm_sb_ds(this.sb,
                    vmm_sb_ds::EXPECT,vmm_sb_ds::IN_ORDER);
      . . .
   endfunction
   . . .
endclass
. . .
forever begin
   my_tr tr;
   this.my_xactor.out_chan.get(tr);
   this.sb.expect_in_order(tr);
end
. . .
```

vmm_channel::unregister_vmm_sb_ds()

Unregister a data stream scoreboard.

SystemVerilog

```
function void unregister_vmm_sb_ds(vmm_sb_ds sb);
```

OpenVera

```
task unregister_vmm_sb_ds(vmm_sb_ds sb);
```

Description

Unregister the specified data stream scoreboard from the channel. An error is issued if the scoreboard was not previously registered with the channel.

Example

Example A-90

```
. . .
//Refer example A-1* (inp_vmm_sb_ds)
class my_env extends vmm_env;
   . . .

this.my_xactor.out_chan.register_vmm_sb_ds(this.sb,vmm_sb_
ds::EXPECT,
                                          vmm_sb_ds::IN_ORDER);
   . . .
   task wait_for_end();
      super.wait_for_end();
      . . .
    //After completed the process of the channel unregister
the scoreboard

this.my_xactor.out_chan.unregister_vmm_sb_ds(this.my_sb);
```

```
   endtask
   . . .
endclass
```

vmm_notify::register_vmm_sb_ds()

Register a data stream scoreboard with a notification.

SystemVerilog

```
function void register_vmm_sb_ds(
     int                   notification_id,
     vmm_sb_ds             sb,
     vmm_sb_ds::kind_e     kind,
     vmm_sb_ds::ordering_e order = IN_ORDER);
```

OpenVera

```
task register_vmm_sb_ds(
     integer               event_id,
     vmm_sb_ds             sb,
     vmm_sb_ds::kind_e     kind,
     vmm_sb_ds::ordering_e order = IN_ORDER);
```

Description

Register a data stream scoreboard with the notification service interface for the specified notification with the specified direction and ordering. The status descriptor specified in the `vmm_notify::indicate()` method is automatically forwarded to all registered scoreboard when the notification is indicated.

If *direction* is specified as `vmm_sb_ds::INPUT`, the status descriptor will be automatically forwarded to the scoreboard by the channel using the "`vmm_sb_ds::insert()`" method.

If *direction* is specified as `vmm_sb_ds::EXPECT`, the status information will be automatically forwarded to the scoreboard using method specified in Table A-1, according to the value specified for *order*.

Example

Example A-91

```
. . .
//Refer Example 2-22
class my_env extends vmm_env;
   . . .
   virtual function build();
      super.build();
      . . .

this.my_xactor.notify.register_vmm_sb_ds(my_xactor::OBSERV
ED,

this.sb,vmm_sb_ds::EXPECT,vmm_sb_ds::IN_ORDER);
      . . .
   endfunction
   . . .
endclass
```

vmm_notify::unregister_vmm_sb_ds()

Unregister a data stream scoreboard.

SystemVerilog

```
function void unregister_vmm_sb_ds(int notification_id,
    vmm_sb_ds sb);
```

OpenVera

```
task unregister_vmm_sb_ds(integer event_id,
    vmm_sb_ds sb);
```

Description

Unregister the specified data stream scoreboard from the notification service interface for the specified notification. An error is issued if the scoreboard was not previously registered with the specified notification.

Example

Example A-92

```
. . .
//Refer example A-3* (inp_vmm_sb_ds)
class my_env extends vmm_env;
   . . .

this.my_xactor.notify.register_vmm_sb_ds(my_xactor::OBSERV
ED,
           this.sb,vmm_sb_ds::EXPECT,vmm_sb_ds::IN_ORDER);
   . . .
   task wait_for_end();
      super.wait_for_end();
      . . .
```

```
      //After completed the process of the notification
unregister the scoreboard

this.my_xactor.notify.unregister_vmm_sb_ds(my_xactor::OBSE
RVED,this.my_sb);
   endtask
   . . .
endclass
```

vmm_xactor::inp_vmm_sb_ds()

Add input transaction to data stream scoreboards.

SystemVerilog

```
protected function void inp_vmm_sb_ds(vmm_data tr);
```

OpenVera

```
protected task inp_vmm_sb_ds(rvm_data tr);
```

Description

Inject the specified transaction descriptor as an input transaction in all input data stream scoreboards, that have been previously registered with this transactor. Input data scoreboards are registered using the "vmm_xactor::register_vmm_sb_ds()" method using the `vmm_sb_ds::INPUT` or `vmm_sb_ds::EITHER` direction.

This method may be called by a transactor at a judicious point in the execution of the transaction, usually after completion of all callback methods.

Example

Example A-93

```
class ahb_master  extends  vmm_xactor;
   ...
   `vmm_callback(ahb_master_cb,
                 post_tr(this, tr));
   this.inp_vmm_sb_ds(tr);
   in_chan.complete();
   ...
endclass
```

vmm_xactor::exp_vmm_sb_ds()

Check output transaction against data stream scoreboards.

SystemVerilog

```
protected function void exp_vmm_sb_ds(vmm_data tr);
```

OpenVera

```
protected task exp_vmm_sb_ds(rvm_data tr);
```

Description

Check the specified transaction descriptor against the expected transaction in all expected data stream scoreboards, that have been previously registered with this transactor. Expected data scoreboards are registered using the "vmm_xactor::register_vmm_sb_ds()" method using the vmm_sb_ds::EXPECT or vmm_sb_ds::EITHER direction.

Table A-1 specifies the expect method that is called depending on the order specified when registering the scoreboard.

This method may be called by a transactor at a judicious point in the execution of the transaction, usually after completion of all callback methods.

Example

Example A-94

```
class ahb_mon  extends  vmm_xactor;
   ...
   `vmm_callback(ahb_master_cb,
                 post_tr(this, tr));
```

```
    this.exp_vmm_sb_ds(tr);
    out_chan.sneak(tr);
    ...
endclass
```

vmm_xactor::register_vmm_sb_ds()

Register a data stream scoreboard.

SystemVerilog

```
function void register_vmm_sb_ds(
     vmm_sb_ds              sb,
     vmm_sb_ds::kind_e      kind,
     vmm_sb_ds::ordering_e order = IN_ORDER);
```

OpenVera

```
task register_vmm_sb_ds(vmm_sb_ds              sb,
     vmm_sb_ds::kind_e      kind,
     vmm_sb_ds::ordering_e order = IN_ORDER);
```

Description

Register a data stream scoreboard with the transactor and with the specified direction and ordering.

If *direction* is specified as `vmm_sb_ds::INPUT` or `vmm_sb_ds::EITHER`, input transactions will be automatically forwarded to the scoreboard by the transactor if it calls the "`vmm_xactor::inp_vmm_sb_ds()`" method.

If *direction* is specified as `vmm_sb_ds::EXPECT` or `vmm_sb_ds::EITHER`, observed or received transactions will be automatically forwarded to the scoreboard by the transactor if it calls the "`vmm_xactor::exp_vmm_sb_ds()`" method.

Example

Example A-95

```
. . .
//Refer example A-5 (inp_vmm_sb_ds)
class my_env extends vmm_env;
   . . .

this.my_xactor.register_vmm_sb_ds(this.sb,vmm_sb_ds::EXPEC
T,vmm_sb_ds::IN_ORDER);
   . . .
endclass
```

vmm_xactor::unregister_vmm_sb_ds()

Unregister a data stream scoreboard.

SystemVerilog

```
function void unregister_vmm_sb_ds(vmm_sb_ds sb);
```

OpenVera

```
task unregister_vmm_sb_ds(vmm_sb_ds sb);
```

Description

Unregister the specified data stream scoreboard from the transactor. An error is issued if the scoreboard was not previously registered with the transactor.

Example

Example A-96

```
. . .
//Refer example A-5 (inp_vmm_sb_ds)
class my_env extends vmm_env;
   . . .

this.my_xactor.register_vmm_sb_ds(this.sb,vmm_sb_ds::EXPEC
T,vmm_sb_ds::IN_ORDER);
   . . .
   task wait_for_end();
      super.wait_for_end();
      . . .
     //After completed the process of the xactor unregister
the scoreboard
      this.my_xactor.unregister_vmm_sb_ds(this.my_sb);
   endtask
   . . .
```

```
endclass
```