VMM Standard Library User Guide

Version C-2009.06
June 2009
VMM Version 1.7 (Synopsys)

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

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

Chapter 5 - VMM Standard Library Customization

Appendix A - Standard Library Classes

1

VMM Standard Library Update

This document is currently a specification for the updates made to the VMM Standard Library as described in Appendix A of the *Verification Methodology Manual for SystemVerilog* and the Errata located at http://vmm-sv.org.

Appendix A, *Standard Library Classes*, of this user's guide, specifies additional classes and class members.

2

Shorthand Macros

The implementation of an extension of the `vmm_data`, `vmm_xactor`, `vmm_subenv` and `vmm_env` classes requires the implementation of many methods (for example, `vmm_data::compare()`, `vmm_data::copy()`, `packing`, `vmm_env::start()`, etc...). Although you only need to implement these methods once, they may be cumbersome to maintain. They are also cumbersome to implement for trivial class extensions.

However, a set of shorthand macros exist to help reduce the amount of code required to implement or use VMM class extensions. These shorthand macros provide a default implementation of all methods for specified data members.

The shorthand macros are specified inside the class specification, after the declaration of the data members. It starts with the `‘vmm_data_member_begin()`, `‘vmm_env_member_begin()`, `‘vmm_subenv_member_begin()` or `‘vmm_xactor_member_begin()` macro and ends with the

corresponding `‘vmm_data_member_end()`, `‘vmm_env_member_end()`, `‘vmm_subenv_member_end()` or `‘vmm_xactor_member_end()` macro. The `vmm_data` shorthand macro section must then be followed by a `‘vmm_data_byte_size()` macro. In between, there must be **only** corresponding `vmm_*_member_*()` shorthand data member macro calls. The order in which the shorthand data member macros are invoked will determine the order in which data members are printed, compared, copied, packed and unpacked.

The data member macros are type-specific. You must use the macro that corresponds to the type of the data member named in its argument. The available data member macros can be found in the section,“vmm_data” on page A-135, “vmm_env” on page A-198, “vmm_subenv” on page A-600 and “vmm_xactor” on page A-663.

Example 2-1 Transaction Implemented Using Shorthand Macros

```
class eth_frame extends vmm_data;
   rand bit [47:0] da;
   rand bit [47:0] sa;
   rand bit [15:0] len_typ;
   rand bit [7:0]  data [];
   rand bit [31:0] fcs;

   ‘vmm_data_member_begin(eth_frame)
      ‘vmm_data_member_scalar(da, DO_ALL)
      ‘vmm_data_member_scalar(sa, DO_ALL)
      ‘vmm_data_member_scalar(len_typ, DO_ALL)
      ‘vmm_data_member_scalar_array(data, DO_ALL)
      ‘vmm_data_member_scalar(fcs,
                              DO_ALL-DO_PACK-DO_UNPACK)
   ‘vmm_data_member_end(eth_frame)
   ‘vmm_data_byte_size(1500, this.len_typ + 16)

   constraint valid_frame {
      fcs == 0;
   }
endclass
```

The use of shorthand macros remain fully backward compliant with classes implemented with explicitly specified methods. You may choose to implement one class using the shorthand macros and another by explicitly implementing all of the methods.

User-defined Implementations

If the shorthand macros are used, then **all** `vmm_data` virtual methods are provided with a default implementation. If it is necessary to provide a different, explicitly coded implementation for one of these methods or data member, it can be implemented using one of two approaches.

User-defined Method Implementation

If most of the default method implementation is suitable, except for one or two specific methods, it is possible to specify a user-defined implementation for those exception methods. The user-defined behavior is specified by implementing one of the following methods, corresponding to the `vmm_data` class method, whose default behavior is not suitable.

```
vmm_data::do_psdisplay()
vmm_data::do_is_valid()
vmm_data::do_allocate()
vmm_data::do_copy()
vmm_data::do_compare()
vmm_data::do_byte_size()
vmm_data::do_max_byte_size()
vmm_data::do_byte_pack()
vmm_data::do_byte_unpack()
```

```
vmm_env::do_psdisplay()
vmm_env::do_start()
vmm_env::do_stop()
vmm_env::do_vote()

vmm_subenv::do_psdisplay()
vmm_subenv::do_start()
vmm_subenv::do_stop()
vmm_env::do_vote()

vmm_xactor::do_psdisplay()
vmm_xactor::do_start_xactor()
vmm_xactor::do_stop_xactor()
vmm_xactor::do_reset_xactor()
```

The following example shows how to replace the default implementation of the `vmm_data::is_valid()` method by implementing the `do_is_valid()` method. All other methods will use the default implementation provided by the shorthand macros.

Example 2-2 Overloaded Default Method Implementation

```
class eth_frame extends vmm_data;
   rand bit [47:0] da;
   rand bit [47:0] sa;
   rand bit [15:0] len_typ;
   rand bit [7:0]  data [];
   rand bit [31:0] fcs;

   `vmm_data_member_begin(eth_frame)
      `vmm_data_member_scalar(da, DO_ALL)
      `vmm_data_member_scalar(sa, DO_ALL)
      `vmm_data_member_scalar(len_typ, DO_ALL)
      `vmm_data_member_scalar_array(data, DO_ALL)
      `vmm_data_member_scalar(fcs,
                              DO_ALL-DO_PACK-DO_UNPACK)
   `vmm_data_member_end(eth_frame)
   `vmm_data_byte_size(1500, this.len_typ+16)
```

```
   virtual bit function do_is_valid(bit silent = 1,
                                    int kind   = -1);
      if (len_typ < 48)
         return 0;
      if (len_typ < 1500 && len_typ != data.size())
         return 0;
      if (len_typ > 1500 && len_typ < 'h0600)
         return 0;

      return 1;
   endfunction

   constraint valid_frame {
      fcs == 0;
   }
endclass
```

To effectively implement these methods, you must use the shorthand macros. However, if you do not use the shorthand macros (for example, all of the class methods are to be explicitly implemented), you must implement, the normal `psdisplay()`, `is_valid()`, `allocate()`, `copy()`, `compare()`, `byte_size()`, `max_byte_size()`, `byte_pack()` and `byte_unpack()` (for example, not their `do_*` counterparts).

User-defined Member Default Implementation

If the unsuitable implementation in the default method pertains to a specific data member, it is possible to provide a user-defined default implementation for that member. The user-defined implementation will be woven with the other default implementations to create the overall default implementation for all virtual methods.

User-defined vmm_data Member Default Implementation

For the `vmm_data` class, this is accomplished by using the `"'vmm_data_member_user_defined()"` macro and implementing a function named `do_`*`membername`*`()`. You **must** implement this function using the following pattern:

```
function bit do_name(
     input vmm_data::do_what_e do_what,
     input string prefix,
     ref string image,
     input classname rhs,
     input int kind,
     ref int offset,
     ref logic [7:0] pack[],
     const ref logic [7:0] unpack());

   do_name = 1; // Success, abort by returning 0

   case (do_what)
      DO_PRINT: begin
         // Add to the 'image' variable, using 'prefix'
      end
      DO_COPY: begin
         // Copy from 'this' to 'rhs'
      end
      DO_COMPARE: begin
         // Compare 'this' to 'rhs'
         // Put mismatch description in 'image'
         // Returns 0 on mismatch
      end
      DO_PACK: begin
         // Pack into 'pack' starting at 'offset'
         // Update 'offset' to end of 'pack'
      end
      DO_UNPACK: begin
         // Unpack from 'unpack' starting at 'offset'
         // Update 'offset' to start of next unpacked data
      end
    endcase

endfunction
```

The following example shows how the default method implementation for the da member can be user-specified to display an IP address using the dot form instead of a hexadecimal value, as provided by the default implementation.

Example 2-3 User-defined Member Default Implementation

```
class eth_frame extends vmm_data;
   rand bit [47:0] da;
   rand bit [47:0] sa;
   rand bit [15:0] len_typ;
   rand bit [7:0]  data [];
   rand bit [31:0] fcs;

   ‘vmm_data_member_begin(eth_frame)
      ‘vmm_data_member_user_defined(da, DO_ALL)
      ‘vmm_data_member_scalar(sa, DO_ALL)
      ‘vmm_data_member_scalar(len_typ, DO_ALL)
      ‘vmm_data_member_scalar_array(data, DO_ALL)
      ‘vmm_data_member_scalar(fcs,
                              DO_ALL-DO_PACK-DO_UNPACK)
   ‘vmm_data_member_end(eth_frame)
   ‘vmm_data_byte_size(1500, this.len+16)

function bit do_da(
    input vmm_data::do_what_e do_what,
    input string prefix,
    ref string image,
    input eth_frame rhs,
    input int kind,
    ref int offset,
    ref logic [7:0] pack[],
    const ref logic [7:0]unpack());

   do_da = 1; // Success, abort by returning 0

   case (do_what)
      DO_PRINT: begin
        $sformat(image, "%s\n%s   DA = %h.%h.%h.%h.%h.%h",
                 this.da[47.40], this.da[39.32],
                 this.da[31:24], this.da[23:16],
                 this.da[15: 8], this.da[ 7: 0]);
      end
      DO_COPY: begin
         rhs.da = this.da;
```

```
      end
      DO_COMPARE: begin
         if (this.da != rhs.da) begin
             $sformat(image, "this.da (%h.%h.%h.%h.%h.%h)
!= to.da (%h.%h.%h.%h.%h.%h)",
                      this.da[47.40], this.da[39.32],
                      this.da[31:24], this.da[23:16],
                      this.da[15: 8], this.da[ 7: 0],
                      rhs.da[47.40], rhs.da[39.32],
                      rhs.da[31:24], rhs.da[23:16],
                      rhs.da[15: 8], rhs.da[ 7: 0]);
            return 0;
         end
      end
      DO_PACK: begin
         if (pack.size() < offset + 6)
            pack = new [offset + 6] (pack);
        {pack[offset  ], pack[offset+1], pack[offset+2],
         pack[offset+3], pack[offset+4], pack[offset+5]} =
            this.da;
         offset += 6;
      end
      DO_UNPACK: begin
         if (unpack.size() < offset + 6) return 0;
         this.da = {unpack[offset  ], unpack[offset+1],
                    unpack[offset+2], unpack[offset+3],
                    unpack[offset+4], unpack[offset+5]};
         offset += 6;
      end
    endcase

endfunction


   constraint valid_frame {
      fcs == 0;
   }
endclass
```

Note that you **must** provide a default implementation for all possible operations (`print`, `compare`, `copy`, `pack` and `unpack`). It is not possible to execute the default implementation that would have otherwise be

provided by the other type-specific shorthand macros. However, it is acceptable to leave the implementation for an operation empty if it not going to be used or has no functional effect.

User-defined vmm_env or vmm_subenv Member Default Implementation

For the vmm_env and vmm_subenv classes, it is accomplished by using the "`vmm_env_member_user_defined()`" or "`vmm_subenv_member_user_defined()`" macro respectively and implementing a function named "do_*membername*(). This function **must** be implemented using the following pattern:

```
function bit do_name(vmm_env::do_what_e do_what);
   do_name = 1; // Success, abort by returning 0

   case (do_what)
      DO_PRINT: begin
         // Add to the 'this.__vmm_image' variable,
         // using 'this.__vmm_prefix'
      end
      DO_VOTE: begin
         // Register with this.end_vote
      end
      DO_START: begin
         // vmm_[sub]env::start() operations.
         // If blocking:
         this.__vmm_forks++;
         fork
         begin
           // Blocking statements...
           this.__vmm_forks--;
         end
         join_none
      end
      DO_STOP: begin
         // vmm_[sub]env::stop() operations.
         // If blocking:
         this.__vmm_forks++;
         fork
         begin
```

```
            // Blocking statements...
            this.__vmm_forks--;
         end
         join_none
      end
   endcase
endfunction
```

The following example shows how the default method implementation for the `'ahb'` transactor can be augmented through an additional user-specified default implementation. In Example 2-4, the default consensus registration for the transactor is augmented with the additional registration of the transactor's input channel.

Example 2-4 Augmenting a Default Implementation

```
class ahb_subenv extends vmm_subenv;
   ahb_master ahb;

   `vmm_subenv_member_begin(ahb_env)
      `vmm_subenv_member_xactor(ahb, DO_ALL)
      `vmm_subenv_member_user_defined(ahb_more)
   `vmm_subenv_member_end(ahb_subenv)

function bit do_ahb_more(vmm_subenv::do_what_e do_what);
   case (do_what)
      DO_VOTE: begin
         this.end_test.register_channel(this.ahb.in_chan);
      end
   endcase
   return 1;
endfunction

endclass
```

Note that a default implementation **must** be provided for all possible operations (print, consensus registration, start and stop). It is not possible to execute the default implementation that would otherwise be provided by the other type-specific shorthand macros. However, it is acceptable to leave the implementation for an operation empty if it not going to be used or has no functional effect.

User-defined vmm_xactor Member Default Implementation

For the vmm_xactor class, it is accomplished by using the "'vmm_xactor_member_user_defined()" macro and implementing a function named "do_*membername*(). This function **must** be implemented using the following pattern:

```
function bit do_name(vmm_xactor::do_what_e do_what,
                     vmm_xactor::reset_e   rst_typ);
   do_name = 1; // Success, abort by returning 0

   case (do_what)
      DO_PRINT: begin
         // Add to the 'this.__vmm_image' variable,
         // using 'this.__vmm_prefix'
      end
      DO_START: begin
         // vmm_xactor::start_xactor() operations.
      end
      DO_START: begin
         // vmm_xactor::stop_xactor() operations.
      end
      DO_RESET: begin
         // vmm_xactor::reset_xactor() operations.
      end
   endcase
endfunction
```

Note that a default implementation **must** be provided for all possible operations (print, consensus registration, start and stop). It is not possible to execute the default implementation that would have otherwise be provided by the other type-specific shorthand macros. However, it is acceptable to leave the implementation for an operation empty if it not going to be used or has no functional effect.

Unsupported Data Types

You can use the user-defined default implementation macro for data members of a type not currently supported by a pre-defined shorthand macro. For example, should a member be an instance of a user-defined class not extended from the `vmm_data` class, it is necessary to use the user-defined default member implementation to perform the correct `display`, `copy`, and `compare` operations for that class.

The following example shows how you can use a user-defined default implementation with a user-defined class with no `display`, `copy`, or `compare` methods.

Example 2-5 Class Member Default Implementation

```
class vlan_tag;
   rand bit [ 2:0] pri;
   rand bit        cfi;
   rand bit [11:0] tag;
endclass

class eth_frame extends vmm_data;
   rand bit [47:0] da;
   rand bit [47:0] sa;
   rand bit [15:0] len_typ;
   rand vlan_tag   vlan;
   rand bit [7:0]  data [];
   rand bit [31:0] fcs;

   `vmm_data_member_begin(eth_frame)
      `vmm_data_member_scalar(da, DO_ALL)
      `vmm_data_member_scalar(sa, DO_ALL)
      `vmm_data_member_scalar(len_typ, DO_ALL)
      `vmm_data_member_user_defined(vlan, DO_ALL)
      `vmm_data_member_scalar_array(data, DO_ALL)
      `vmm_data_member_scalar(fcs,
                              DO_ALL-DO_PACK-DO_UNPACK)
   `vmm_data_member_end(eth_frame)
   `vmm_data_byte_size(1500, this.len+16)
```

```
function bit do_vlan(
    input vmm_data::do_what_e do_what,
    input string prefix,
    ref string image,
    input eth_frame rhs,
    input int kind,
    ref int offset,
    ref logic [7:0] pack[],
    const ref logic [7:0] unpack());

   do_da = 1; // Success, abort by returning 0

   case (do_what)
      DO_PRINT: begin
         if (this.vlan == null) return 1;
         $sformat(image, "%s\n%s   VLAN: %0d/%b (%h)",
                  this.pri, this.cfi, this.tag);
      end
      DO_COPY: begin
         rhs.vlan = (this.vlan == null) ? null
                    : new this.vlan;
      end
      DO_COMPARE: begin
         if (this.vlan == null && rhs.vlan == null)
            return 1;
         if (this.vlan == null) begin
            image = "No VLAN on this but found on to";
            return 0;
         end
         if (this.rhs == null) begin
            image = "VLAN on this but not on to";
            return 0;
         end
         if (this.vlan.pri != rhs.vlan.pri) begin
          $sformat(image, "this.vlan.pri (%0d) != to.vlan.pri
(%0d)",
                   this.vlan.pri, rhs.vlan.pri);
            return 0;
         end
         if (this.vlan.cfi != rhs.vlan.cfi) begin
          $sformat(image, "this.vlan.cfi (%b) != to.vlan.cfi
(%b)",
                   this.vlan.cfi, rhs.vlan.cfi);
            return 0;
         end
         if (this.vlan.tag != rhs.vlan.tag) begin
```

```
         $sformat(image, "this.vlan.tag (%h) != to.vlan.tag
(%h)",
                     this.vlan.tag, rhs.vlan.tag);
            return 0;
         end
      end
      DO_PACK: begin
         if (this.vlan == null) return 1;
         if (pack.size() < offset + 4)
            pack = new [offset + 4 (pack);
         {pack[offset  ], pack[offset+1] = 'h8100;
         {pack[offset+2], pack[offset+3] =
            {this.vlan.pri, this.vlan.cfi, this.vlan.tag};
         offset += 4;
      end
      DO_UNPACK: begin
         if (unpack.size() < offset + 4) return 1;
         if ({unpack[offset], unpack[offset+1]}
             != 'h8100) return 1;
         this.vlan = new;
         {this.vlan.pri, this.vlan.cfi, this.vlan.tag} =
            {unpack[offset+2], pack[unoffset+3];
         offset += 4;
      end
    endcase

endfunction



   constraint valid_frame {
      fcs == 0;
   }
endclass
```

Virtual interfaces are user-defined types and are not directly supported by a corresponding shorthand macro. Example 2-6 shows how the default method implementation for a transactor can deal with a physical interface through an additional user-specified default implementation.

Example 2-6 Dealing with physical interfaces

```
class my_bfm extends vmm_xactor;
   virtual interface my_ifc sigs;
   ...
   ‘vmm_xactor_member_begin(my_bfm)
       ‘vmm_xactor_member_user_defined(sigs)
   ‘vmm_xactor_member_end(my_bfm)

function bit do_sigs(vmm_xactor::do_what_e do_what,
                     vmm_xactor::reset_e   rst_typ);
   case (do_what)
      DO_RESET: begin
         this.sigs.dat_out <= ’0;
      end
   endcase
   return 1;
endfunction

endclass
```

3

Constructing Sub-environments

The VMM promotes the design of transactors and self-checking structures so you can reuse them in different environments. For example, you can construct system-level verification environments of the same basic components used to construct block-level environments.

When you construct a system-level environment using the same basic components used to construct block-level environments, VMM arranges, combines and connects these same basic components, the same way. For example, a block-level self-checking structure—complete with stimulus and response monitors and scoreboard—may be identical in the system-level environment, if the system-level, self-checking mechanism, consists of checking the behavior of the individual blocks which compose it. Similarly, different block-level environments may have a need for similar combinations of basic components, for example, a complete TCP/IP stimulus stack.

You can minimize the overall effort and maintenance, if you construct block and system-level environments by reusing complex testbench structures, which already provide a significant portion of the required functionality.

In this chapter, a "sub-environment" refers to a subset of a verification environment that is reusable in a different verification environment. Sub-environments are not individual transactors. Sub-environments are composed of two or more interconnected transactors, potentially linked to additional elements such as a scoreboard, a file I/O mechanism or a response generator, implementing a specific functionality.

This chapter contains the following sections:

Architecting Sub-environments

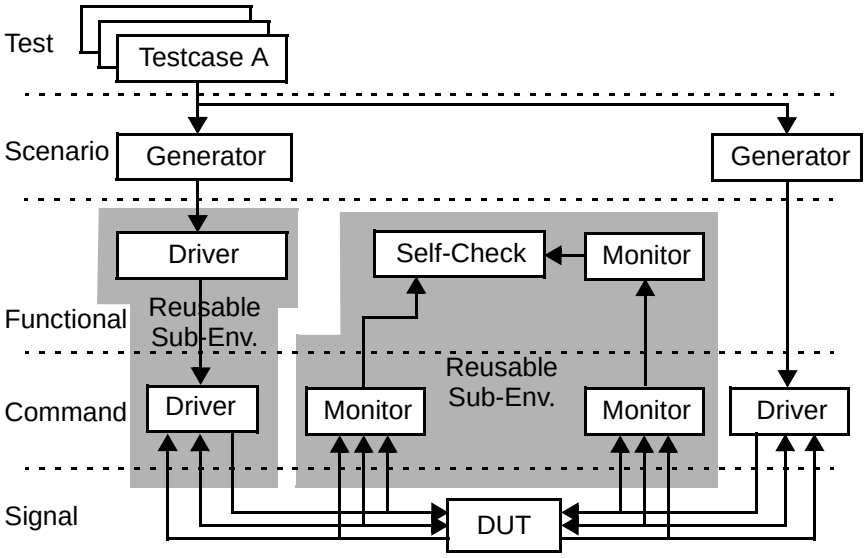
You must identify and architect reusable sub-environments when you first design and architect a verification environment. Like any other reusable components, reusable sub-environments will not happen by accident nor after the fact. Neither can you reuse sub-environments if a verification environment is not designed to take advantage of it.

The remainder of this section will present some guidelines and hints to help identify the architect reusable sub-environments.

A sub-environment may span multiple verification environment layers. The VMM defines different abstraction layers in verification environments. These layers are more logical than structural. Although a transactor or basic verification component will typically sit in a single layer, a sub-environment can encompass transactors and components in different layers.

For example, Figure 3-1 shows a layered verification environment. The self-checking and stimulus protocol stack structures, which can be made into reusable sub-environments, span two of those layers.

Figure 3-1 Layered Verification Environment Architecture



A sub-environment may have transaction-level interfaces.

It would be wrong to read too much in Figure 3-1. Although the depicted sub-environments have physical-level interfaces, a reusable sub-environment can also have transaction-level input and outputs, as shown in Figure 3-2.

A physical-level interface is limited to monitoring signal-level activity on a specific physical bus. A transaction-level interface can be fed using a different monitor, extracting the same transactions transported on a different physical bus. It can also be fed from a driver transactor, as shown in Figure 3-3, eliminating or delaying the need to develop a command-layer monitor if none is readily available.

Figure 3-2 Sub-environment with Transaction-level Interface

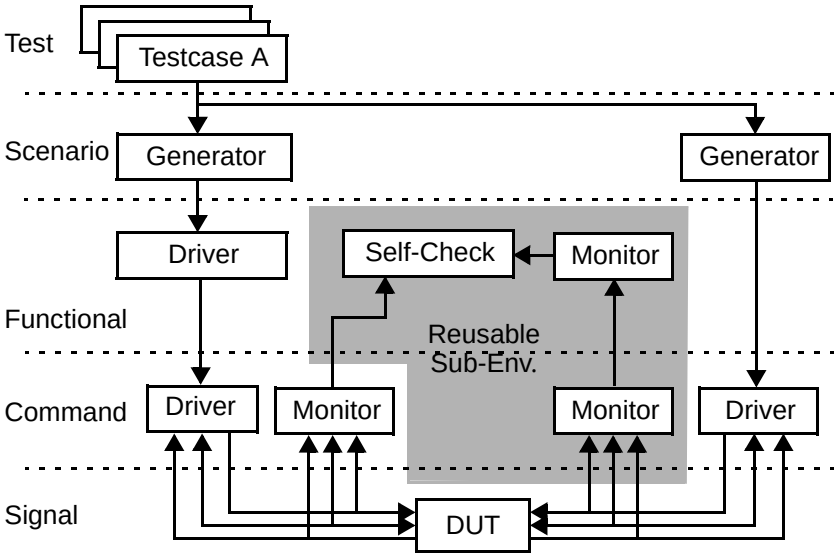
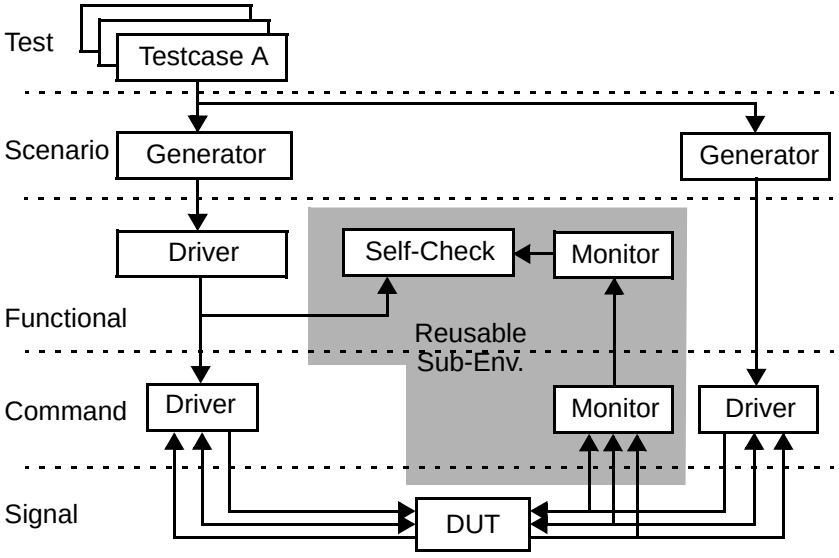
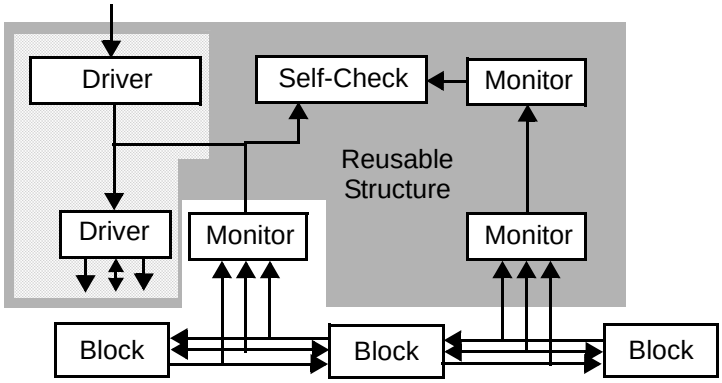


Figure 3-3 Sub-environment Interfaced to Driver Transactor



The structure of a sub-environment can be configurable. Instead of creating two sub-environment, as shown in Figure 3-3, you can create a single sub-environment which you can configure with or without the protocol stimulus stack. In a block-level environment, you would configure the sub-environment with the protocol stimulus stack. In a system-level environment, another block within the system provides the stimulus, therefore, you would configure the sub-environment without the protocol stimulus stack, as shown in Figure 3-4.

Figure 3-4 Configurable Sub-environment in System-level Environment



There are different ways you can specify the configuration of a sub-environment. The following section, *Implementing Sub-environments*, describes the various techniques.

Implementing Sub-environments

This section provides guidelines and techniques for implementing reusable sub-environments that you can reuse across different verification environments, or instantiate multiple times in the same verification environment.

Sub-environments shall be encapsulated using the "vmm_subenv" base class or a derivative.

This base class provides generic functionality required by most sub-environments. It also provides, through virtual methods, standard interfaces for functionality that the sub-environments must provide.

Furthermore, using a common base class for all sub-environments makes it easy to identify their nature and boundaries. Also, a common base class allows the development of generic functionality to deal with a collection of sub-environments. For example, an environment could maintain an array of references to all of the sub-environments it contains to easily start and stop all of them.

Example 3-1 Sub-environment Declaration

```
class mii_eth_frame_sb extends vmm_subenv;
   ...
endclass
```

You should derive sub-environment classes from VMM_SUBENV.

By default, VMM defines this pre-processor symbol to `vmm_subenv`. You may choose to provide your own sub-environment base class, derived from the "`vmm_subenv`" base class, to provide additional organization-specific functionality associated with the particular applications or methods used by the organization.

By redefining the value of the macro from the command line, a sub-environment can thus be derived from an organization-specific base class, even if it comes from outside the organization.

Example 3-2 Retargetable Sub-environment Declaration

```
class mii_eth_frame_sb extends ‘VMM_SUBENV;
   ...
endclass
```

Example 3-3 Retargeting Sub-environment Declarations

```
% vcs +define+VMM_SUBENV=my_subenv ...
```

All transaction-level interfaces shall be defined as VMM channels constructor arguments.

This process documents the transaction-level connectivity of the sub-environment. These channels can then be directly connected to the appropriate transactors inside the sub-environment.

For example, the input to the self-checking block in the sub-environment shown in Figure 3-2 is a transaction-level interface that you could define as shown in Example 3-4.

Example 3-4 Transaction-level Interface Definition

```
function new(eth_frame_channel tx_frames_in, ...);
   ...
endfunction: new
```

All physical-level interfaces shall be defined as virtual modport constructor arguments.

This process documents the physical-level connectivity of the sub-environment. These virtual modports can then be directly connected to the appropriate transactors inside the sub-environment.

For example, the input to the monitors in the sub-environment shown in Figure 3-1 are physical-level interface that could be defined as shown in Example 3-5.

Example 3-5 Physical-Level Interface Definition

```
function new(   virtual ahb_bus.passive tx_frame,
                virtual mii_phy.passive rx_frame,
             ...);
   ...
endfunction: new
```

A `null` value for a channel or virtual modport constructor argument shall be interpreted as an unconnected interface.

You can specify a *null* literal value for transaction-level interfaces (`vmm_channel` references) and physical-level interfaces (virtual modports). You must interpret this as an open connection and thus an unused interface.

The consequences of having an unconnected interface must be appropriately handled by the sub-environment. An unconnected interface may cause the sub-environment to configure itself accordingly. For example, the output channel of a monitor may be sunk if the corresponding output channel is unconnected and a command-level transactor may not be instantiated—or simply not started—if the corresponding virtual modport is unconnected.

If an interface cannot be left unconnected, an error message must be issued.

Sub-environments should have a reference to a configuration descriptor as a constructor argument

Sub-environments may be configurable in more ways than simply leaving interfaces unconnected. A sub-environment configuration descriptor contains class properties for configuring the sub-environment itself. In addition, the transactors they encapsulate are most likely configurable themselves. A sub-environment configuration descriptor would typically contain a configuration descriptor class property for each encapsulated transactor with its own configuration descriptor.

The sub-environment configuration descriptor would typically be randomized in the `vmm_env::gen_cfg()` method extension for the environment containing the reusable structure. The randomized (or directed) value would then be passed to the constructor of the sub-environment in the extension of the `vmm_env::build()` step.

Example 3-6 Sub-environment Configuration Descriptor

```
class mii_eth_frame_sb_cfg;
   rand ahb_cfg ahb;
   rand mii_cfg mii;
endclass: mii_eth_frame_sb_cfg

class mii_eth_frame_sb extends vmm_subenv;
   function new(mii_eth_frame_sb_cfg cfg, ...);
      ...
   endfunction: new
 endclass: mii_eth_frame_sb
```

There shall be a task named `configure()` to configure the sub-environment and the portion of the DUT associated with the sub-environment.

If the functionality of the sub-environment is configurable, then the sub-environment, and the portion of the DUT that corresponds to the functionality it verifies, must also be configured accordingly.

If the sub-environment and associated DUT functionality is not configurable, this method must still exist to document that fact.

Example 3-7 Sub-environment DUT Configuration Method

```
class mii_eth_frame_sb extends vmm_subenv;
   ...
   task configure(...);
      ...
      super.configured();
   endtask: configure
endclass: mii_eth_frame_sb
```

There is no virtual method in the "vmm_subenv" base class corresponding to this task because it will likely require different arguments for different sub-environments.

The `configure()` method shall call the "vmm_subenv::configured()" method upon successful completion.

The "vmm_subenv::configured()" method is used to confirm to the base class that it—and its associated DUT functionality—has been properly configured and it can be started. If this method is not invoked, the "vmm_subenv::start()" method will issue an error.

The `configure()` method shall configure the DUT through a register abstraction layer.

A sub-environment associated with a specific block-level DUT may be reused in a system-level environment where the corresponding block is no longer directly accessible. The address, physical bus or hierarchical path used to program registers in the block-level DUT may be different than the ones used to originally develop the reusable structure.

A register abstraction layer allows registers and memories in a block-level DUT to be accessed in their current state, regardless of their actual physical context. The appropriate register abstraction interface must then be passed as an argument to the *`configure()`* task.

Example 3-8 Configuring through Register Abstraction Layer

```
class mii_eth_frame_sb extends vmm_subenv;
   ...
   task configure(ral_mac_block blk);
      if (this.cfg.mii.duplex) blk.duplex.set(1);
      else blk.duplex.set(0);
      ...
      if (blk.update() != vmm_rw::IS_OK) begin
         ...
         return;
      end
      super.configured();
   endtask: configure
endclass: mii_eth_frame_sb
```

Extensions of the “vmm_subenv” base class implement the “vmm_subenv::start()”, “vmm_subenv::stop()” and “vmm_subenv::cleanup()” virtual methods.

These methods implement the corresponding generic steps that must be performed to successfully simulate a testcase that includes the sub-environment. They must be overloaded to perform each step as required by the sub-environment. Even if a method does not need to be extended for a particular sub-environment, it should be extended anyway—and left empty—to explicitly document that fact.

These methods must then be called in their corresponding simulation step method in the extension of the vmm_env base class where a sub-environment is used.

Extensions of the , “vmm_subenv::stop()” and “vmm_subenv::cleanup()” virtual methods shall call their base implementation first.

The implementation of these methods in the base class manages the sequence in which these methods must be invoked. They will report an error if a sub-environment it not properly used.

Example 3-9 Extending a Simulation Step Method

```
class mii_eth_frame_sb extends vmm_subenv;
   ...
   virtual task start();
      super.start();
      this.mii.start_xactor();
      ...
   endtask: start
   ...
enclass: mii_eth_frame_sb
```

Extensions of the “vmm_subenv” base class may implement the “vmm_subenv::report()” virtual method.

This method is designed to implement any status, coverage or statistical reporting of information collected by the sub-environment. The default implementation is empty.

Extensions of the “vmm_subenv::report()” method shall not report on the success or failure of the simulation.

Extensions of this method should not be used to determine the pass or fail status of the simulation. This should be left to the vmm_env::report() method of the environment instantiating the sub-environment. If an error is detected that causes the failure of the

simulation, it should be reported through a vmm_log error message in the "vmm_subenv::cleanup()" method. The message service will record the error message and fail the simulation accordingly.

A reference to a vmm_consensus instance will be provided via the constructor and used to indicate that the sub-environment has reached its end-of-test condition.

The sub-environment must be able to participate in the decision of whether or not to end the simulation. This decision must take into account other sub-environments, the overall verification environments and the testcase itself. The vmm_consensus utility class offers a well-defined service for collaboration upon deciding when a test is complete and simulation can be halted.

How a sub-environment determines if the test can end or not is specific to the sub-environment itself. It can be implemented in various ways:

1. Fork threads in the extension of the "vmm_subenv::start()" method to watch for conditions, such as a generator being done, to consent or disagree to end the test.
2. Have the self-checking structure consent to the end of the test once a pre-determined condition, such as a specific number of observed transactions, has been observed.
3. Register all transactors and channels in the sub-environment with the vmm_consensus instance to consent to the end of test when all transactors are idle and all channels are empty.

4

Using Scenarios

The atomic generator creates a stream of individually-randomized transactions. This is fine for creating broad-spectrum stimulus, but corner cases would likely require a more constrained sequence of transactions. Scenarios are short sequences of transactions that are directed or mutually constrained, or a combination of both.

This chapter describes how to specify single-stream and multi-stream scenarios, both random and directed, as well as hierarchical scenarios.

Appendix A includes detailed documentation for `vmm_scenario_gen` and `vmm_scenario::define_scenario()`, `vmm_ms_scenario_gen` and `vmm_ms_scenario`.

Architecture of the Generators

The scenario generators and multi-stream scenario generators are transactors that repeatedly select a scenario from a set of available scenarios, randomize it, then execute it. Once a scenario is executed, the total number of transactions created by the scenario is added to the total number of transactions generated by the generator, and the number of generated scenarios is incremented. This process is repeated until the maximum number of scenarios, or transaction descriptors to generate, has been reached or exceeded.

By default, the single-stream scenario generator provides only one scenario: a scenario that randomizes then applies just one transaction. Functionally, the default behavior of the single-stream scenario generator is equivalent to that of the atomic generator. Single-stream scenarios must be registered with a single-stream scenario generator to produce different stimulus. Note that the performance of the default-configuration single-stream scenario generator is significantly lower than the atomic generator because of the overhead associated with selecting, randomizing and applying scenarios. It should not be used as a replacement of the atomic generator.

By default, the multi-stream scenario generator does not provide any scenarios. Multi-stream scenarios must be registered with a multi-stream scenario generator to produce stimulus.

Scenarios are registered to the desired scenario generator instance via the `vmm_scenario_gen::register_scenario()` or `vmm_ms_scenario_gen::register_ms_scenario()` method. This allows specific generators to generate the desired stimulus

sequence and no other. Should a scenario have to be registered with multiple instances of scenario generators, the transactor iterator can be used, as shown in Example 4-1.

Example 4-1 Registering a scenario with multiple generators

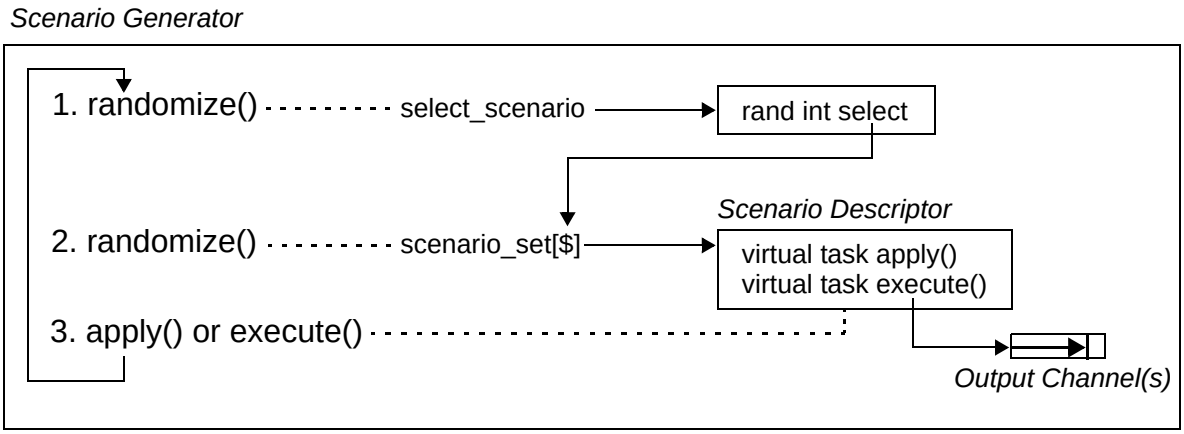
```
‘foreach_vmm_xactor(ahb_scenario_gen, "/./", "/./") begin
   my_ahb_scenario sc = new();
   xact.register_scenario(sc);
end
```

Scenario Selection

As illustrated in Figure 4-1, a generator selects the next scenario to generate, among all of the scenarios registered with it, by randomizing its `vmm_scenario_gen::select_scenario` or `vmm_ms_scenario_gen::select_scenario` class property. The selected scenario is identified by the final value of the `vmm_scenario_election::select` or `vmm_ms_scenario_election::select`. It is interpreted by the generator as the index in the

`vmm_scenario_gen::scenario_set[$]` or `vmm_ms_scenario_gen::scenario_set[$]` of the scenario to generate.

Figure 4-1 Scenario selection and execution process



By default, the `vmm_scenario_election::round_robin` and `vmm_ms_scenario_election::round_robin` constraint blocks constrains the selection process to a round-robin order. By turning off this constraint block, the scenario selection process can be made completely random.

Example 4-2 Making the scenario selection random

```
env.gen[2].select.round_robin.constraint_mode(0);
```

The instance of the `vmm_scenario_election` or `vmm_ms_scenario_election` class in the `vmm_scenario_gen::select_scenario` or `vmm_ms_scenario_gen::select_scenario` class property can be replaced to create a different selection process. Various state variables are available to help procedurally or randomly determine the next scenario to execute.

Single-Stream Scenarios

The single-stream scenario generator is a type-specific generator that is declared using the `vmm_scenario_gen() macro`, as shown in Example 4-3. This creates a class named *`class_name`*`_scenario_gen` where *`class_name`* is the name of the user-defined class supplied to the macro.

Example 4-3 Declaring a single-stream scenario generator

```
class eth_frame extends vmm_data;
   ...
endclass
‘vmm_channel(eth_frame)
‘vmm_scenario_gen(eth_frame, "Ethernet Frames")
```

The single-stream scenario generator is connected to a single output channel at construction time, or by assigning its `vmm_scenario_gen::out_chan` class property. All generated scenarios will be injected in that output channel.

Example 4-4 Instantiating a single-stream scenario generator

```
class tb_env extends vmm_env;
   eth_frame_scenario_gen gen;
   eth_frame_channel gen_to_bfm;
   ...
   virtual function void build();
      super.build();
      this.gen_to_bfm = new();
      this.gen = new("gen", 0, this.gen_to_bfm);
   endfunction
   ...
endclass
```

The macro also defines a single-stream scenario descriptor class named *`class_name`*`_scenario`. This class contains a type-specific array of transaction descriptors that are randomized according to the constraints in the scenario descriptor.

The macro predefines an atomic scenario in a class named *class_name*_atomic_scenario. An instance of this class will be registered by default with any instance of the corresponding single-stream scenario generator. This default scenario will need to be unregistered if it is not desired.

Random Scenarios

By default, single-stream scenarios are randomly generated. The combination of three things makes this happen:

- A single-stream scenario descriptors contains a *rand* array of user-defined transaction descriptors in the *class_name*_scenario::items[] class property.
- After being selected, the scenario descriptor is automatically randomized by the generator
- The default behavior of the *class_name*_scenario::apply() method copies the content of the *class_name*_scenario::items[] class property onto the generator's output channel.

As illustrated in Example 4-5, a random scenario is defined by extending the *class_name*_scenario class and providing constraints over the elements of the *class_name*_scenario::items[] class property. The maximum length of the scenario must also be specified by calling the vmm_scenario::define_scenario() method.

Example 4-5 Declaring a random single-stream scenario

```
class bad_eth_frames extends eth_frame_scenario;
   function new();
      this.define_scenario("Bad Frames", 10);
   endfunction
```

```
   constraint bad_eth_frames_valid {
      foreach (this.items) {
         this.items[i].fcs != 0;
      }
   }
endclass
```

Procedural Scenarios

Procedural—or directed—scenarios are specified by overloading the *class_name*_scenario::apply() method. The procedural scenario can be any user-defined code that puts transaction descriptors into the supplied output channel. The total number of procedurally generated transactions is then returned via the *n_insts* argument.

Note that it is important that *super.apply()* not be called, otherwise any transaction descriptor found in the *class_name*_scenario::items[] class property will also be injected into the output channel.

Random transactions can be created by using *rand* class properties (such as the predefined *class_name*_scenario::items[] class property), or by explicitly calling *randomize()* on local variables or non-random class properties.

Example 4-6 Declaring a procedural single-stream scenario

```
class collision extends eth_frame_scenario;
   virtual mii_if sigs;

   function new(virtual mii_if sigs);
      this.define_scenario("Collision", 1);
      this.sigs = sigs;
   endfunction
```

```
   virtual task apply(eth_frame_channel channel,
                      ref int unsigned  n_insts);
      @ (posedge this.sigs.crs);
      channel.put(this.items[0]);
      n_insts++;
   endtask
endclass
```

If the sequence of transactions generated by the scenario must not be interrupted by stimulus from another scenario (see “Multi-Stream Scenarios” on page 4-11), an output channel may be taken for exclusive use until it is explicitly released. If the channel is not currently taken by another scenario, it will be immediately reserved for the exclusive use of this scenario descriptor. If the channel is currently taken by another scenario, the execution of this scenario descriptor will be suspended until the channel becomes available.

Example 4-7 Ensuring a transaction order in a single-stream scenario

```
class dot_dot_dot extends eth_frame_scenario;
   function new();
      this.define_scenario("Exclusive", 0);
   endfunction

   virtual task apply(eth_frame_channel channel,
                      ref int unsigned  n_insts);
      eth_frame fr;

      fr = new;
      fr.randomize() with {...};
      channel.grab(this);
      repeat (3) begin
         channel.put(fr.copy(), .grabber(this));
      end
      channel.ungrab(this);
      n_insts += 3;
   endtask
endclass
```

Hierarchical Scenarios

Scenarios can also be described hierarchically by composing them of lower-level scenarios. A hierarchical scenario is a procedural scenario. The lower-level scenario descriptors are simply instantiated in the higher-level scenario descriptor. The higher-level scenario's *apply()* method calls the lower-level scenario's respective *apply()* method in the appropriate sequence.

Example 4-8 Declaring a hierarchical single-stream scenario

```
class bad_frames_then_collision extends eth_frame_scenario;
   rand bad_eth_frames bad;
   rand collision      col;

   function new(virtual mii_if sigs);
      this.define_scenario("Bad+Collision", 0);
      this.bad = new();
      this.col = new(sigs);
   endfunction

   virtual task apply(eth_frame_channel channel,
                      ref int unsigned  n_insts);
      this.bad.apply(channel, n_insts);
      this.col.apply(channel, n_insts);
   endtask
endclass
```

Hierarchical scenarios are registered, like any other scenarios. If the sub-scenarios are relevant top-level scenarios, they also need to be registered to become available for selection.

Example 4-9 Registering hierarchical and flat scenarios

```
‘foreach_vmm_xactor(eth_frame_scenario_gen,
                    "/./", "/./") begin
   mii_phy                    phy;
   if ($cast(phy, xact.out_chan.get_consumer())) begin
      bad_frames_then_collision btc = new(phy.sigs);
      bad_eth_frames            bad = new;
```

```
      xact.register_scenario("Bad then Col", btc);
      xact.register_scenario("Bad Burst", bad);
   end
end
```

To prevent deadlock situations, a channel that is currently taken by a higher-level scenario is available to be taken by any of its lower-level scenarios. To make the exclusive use of an output channel from a higher-level scenario available to a lower-level scenario, it is necessary to specify that the higher-level scenario instance is a parent of the lower-level scenario.

Example 4-10 Preventing deadlocks in taking the output channel

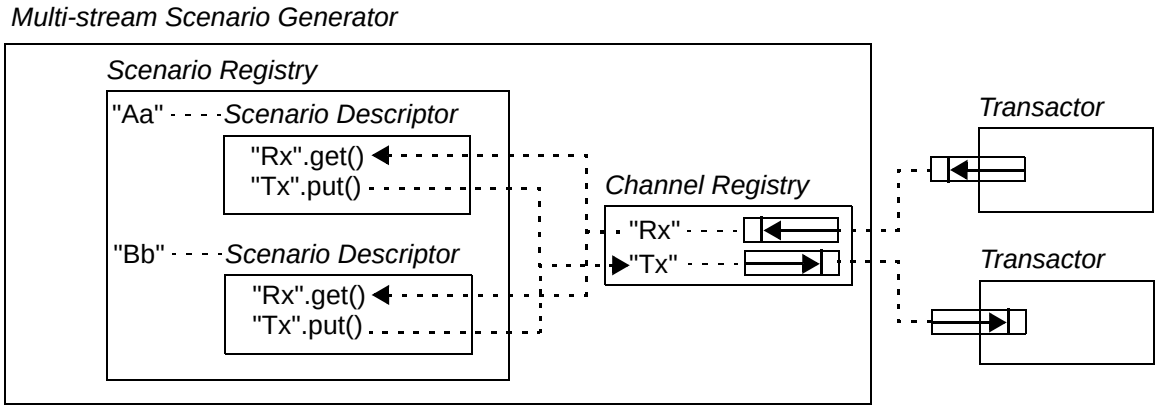
```
class bad_frames_then_collision extends eth_frame_scenario;
   rand dot_dot_dot    ddd;
   rand bad_eth_frames bad;
   rand collision      col;

   function new(virtual mii_if sigs);
      this.define_scenario("Bad+Collision", 0);
      this.ddd = new();
      this.bad = new();
      this.col = new(sigs);

      this.ddd.set_parent_scenario(this);
      this.bad.set_parent_scenario(this);
      this.col.set_parent_scenario(this);
   endfunction

   virtual task apply(eth_frame_channel channel,
                      ref int unsigned  n_insts);
      channel.grab(this);
      this.bad.apply(channel, n_insts);
      this.col.apply(channel, n_insts);
      channel.ungrab(this);
   endtask
endclass
```

Multi-Stream Scenarios

Multi-stream scenarios are able to inject stimulus on multiple output channels. Unlike single-stream scenarios, multi-stream scenarios are not implicitly injected in a channel. They must be explicitly defined by extending their `vmm_ms_scenario::execute()` method. That is not to say that random multi-stream scenarios are not possible! A random multi-stream scenario can be implemented by defining properties as "rand" or by calling "randomize()" from within the `vmm_ms_scenario::execute()` method.

As illustrated in Figure 4-2, multi-stream scenarios interact with channels identified by logical names. This allows the same scenario to be executed on a different set of channels. Channels are associated with a logical name by registering them with an instance of a multi-stream scenario generator, using the `vmm_ms_scenario_gen::register_channel()` method. The channel associated with a logical name is obtained from within the `vmm_ms_scenario::execute()` method by calling the `vmm_ms_scenario::get_channel()` method.

Figure 4-2 Channels in multi-stream scenarios



Example 4-11 Registering logical channel pairs

```
foreach (this.ms_gen[i]) begin
   this.ms_gen[i].register_channel("Tx",
                                   this.bfm[i].tx_chan);
   this.ms_gen[i].register_channel("Rx",
                                   this.bfm[i].rx_chan);
end
```

A multi-stream scenario need not generate stimulus on multiple output channels. A single-channel multi-stream scenario can be used to describe a single-stream scenario. Similarly, a multi-stream scenario generator may be connected to only one output channel, effectively emulating a single-stream scenario generator. The performance of a multi-stream scenario generator used in a single-stream application should be comparable to the performance of a single-stream scenario generator.

Procedural Scenarios

Multi-stream scenarios are procedural scenarios because they do not contain the pre-defined functionality of random scenarios. The only randomization that occurs implicitly is the randomization of the

multi-stream scenario descriptor before it is executed. The execution of the procedural scenario could include further randomization of local variables and data members.

A multi-stream scenario is specified by overloading the `vmm_ms_scenario::execute()` task in an extension of the `vmm_ms_scenario` class. Each multi-stream scenario is specified as a separate class extension. The execution of this task constitutes the multi-stream scenario. It is up to the task to create or randomize transaction descriptors then copy them in the appropriate channels. It is important that a proper factory pattern be used when implementing random scenarios so the transaction descriptor may be further constrained.

Example 4-12 A simple multi-stream scenario

```
class simple_scenario extends vmm_ms_scenario;
   rand ahb_cycle ahb;
        ocp_cycle ocp;

   function new(vmm_scenario parent = null);
      super.new(parent);
      this.ahb = new;
      this.ocp = new;
   endfunction

   virtual task execute(ref int n);
      vmm_channel ocp_chan = this.get_channel("OCP");
      vmm_channel ahb_chan = this.get_channel("AHB");
      fork
         begin
            this.ocp.randomize();
            ocp_chan.put(this.ocp.copy());
         end
         // this.ahb will be randomized when this
         // class is randomized by the generator
         ahb_chan.put(this.ahb.copy());
      join
      n += 2;
   endtask
endclass
```

A multi-stream scenario generator can be connected to any channel instance in the testbench environment. But such a connection does not prevent other transactors to concurrently inject transactions to a channel. A scenario is not guaranteed exclusive access to an output channel. Multiple threads in the same scenario may inject transactions in the same channel. Or an other generator may be actively generating its own stream of transaction in a channel, concurrently with the multi-stream generator.

If a multi-stream scenario requires exclusive access to a channel, to ensure that its specific sequence of transactions is not interrupted or mixed with a sequence from another thread in the same scenario or from another transactor, it must first grab the channel. Once grabbed, all other potential producers on the channel will be blocked from injecting transactions in the channel until it will have been explicitly ungrabbed. When injecting transactions in a potentially grabbed channel, a reference to the scenario currently injecting the transaction must be supplied to *grabber* argument of the `vmm_channel::put()` or `vmm_channel::sneak()` methods.

Example 4-13 A multi-stream scenario with exclusive channel access

```
class exclusive_access extends vmm_ms_scenario;
   rand ahb_cycle ahb;

   function new(vmm_scenario parent = null);
      super.new(parent);
      this.ahb = new;
   endfunction

   virtual task execute(ref int n);
      vmm_channel chan = this.get_channel("AHB");
      chan.grab(this);
      repeat (10) chan.put(this.ahb, .grabber(this));
      chan.ungrab(this);
      n += 2;
   endtask
endclass
```

Hierarchical Scenarios

Multi-stream scenarios can be composed of other single-stream and multi-stream scenarios. There are two kinds of hierarchical scenarios: "contained" and "distributed".

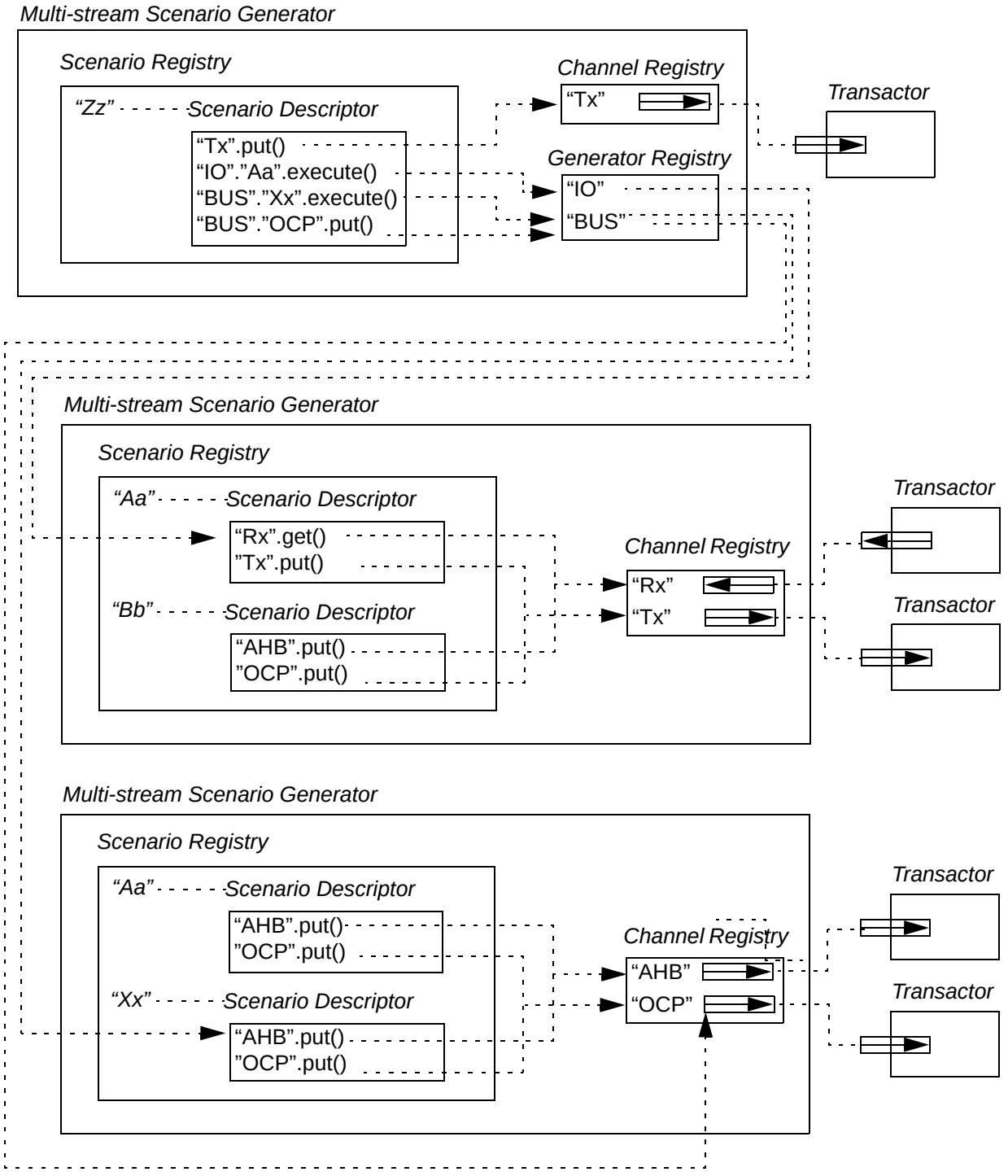
A contained multi-stream scenario is entirely described and executed by a multi-stream scenario descriptor. It executes within the context of a single multi-stream scenario generator, as illustrated in Figure 4-2. The sub-scenarios in a contained hierarchical scenario execute on the same logical channels as the top-level scenario.

Example 4-14 Contained hierarchical multi-stream scenario

```
class contained extends vmm_ms_scenario;
   rand simple_scenario        simple;
   rand exclusive_access       excl;
   rand single_stream_scenario sss;

   function new(vmm_scenario parent = null);
      super.new(parent);
      this.simple = new(this);
      this.excl   = new(this);
      this.sss    = new();
      this.sss.set_parent_scenario(this);
   endfunction

   virtual task execute(ref int n);
      fork
         begin
            this.simple.execute(n);
            this.excl.execute(n);
         end
         this.sss.apply(this.get_channel("MII"), n);
      join
   endtask
endclass
```

A distributed multi-stream scenario is described and executed by multiple multi-stream scenario descriptors. Each multi-stream scenario descriptor executes within the context of the multi-stream scenario generator where it is registered, as shown in Figure 4-3. The sub-scenarios in a distributed hierarchical scenario execute on the logical channels as registered in the multi-stream scenario generator where they execute.

Figure 4-3 Distributed hierarchical multi-stream scenarios



Example 4-15 Distributed hierarchical multi-stream scenario

```
class distributed extends vmm_ms_scenario;
   rand simple_scenario        simple;

   function new(vmm_scenario parent = null);
      super.new(parent);
      this.simple = new(this);
   endfunction

   virtual task execute(ref int n);
      fork
         this.simple.execute(n);
         begin
            vmm_ms_scenario mii;
            mii = this.get_ms_scenario("MII_GEN",
                                       "Collision");
            if (mii != null) mii.execute(n);
         end
      join
   endtask
endclass
...
initial
begin
   vmm_ms_scenario_gen top_gen = new;

   // Assuming that somewhere, this registration happens
   // env.mii_gen.register_ms_scenario("Collision", ...);

   top_gen.register_ms_scenario_gen("MII_GEN",
                                    env.mii_gen);
   begin
      distributed d = new;
      this.top_gen.register_ms_scenario("Example", d);
   end
   ...
end
```

Of course, a distributed hierarchical scenario can be composed of contained hierarchical scenarios.

Implementing Multi-Stream Scenario Generation

Multi-stream scenarios extend the **`vmm_ms_scenario`** class and define the execution of the scenario **`execute()`** method. The content of the **`execute()`** method is controlled entirely by the user, which can be executing single/multiple transactions/scenarios. Execution can be single threaded, multi-threaded, reactive, and so on, depending on the user requirements. Then the scenario object must be registered with a multi-stream generator. This generator executes the registered scenario. Multiple scenarios can be registered to the same MS generator.

Creating and Executing a Multi-Stream Scenario

Create a scenario class and define the execute() method

Create a scenario class extending **`vmm_ms_scenario`** and define any properties as rand if they are intended to be randomized before the execution of the **`execute()`** method. Then define the **`execute()`** method as needed. YOu can update the *n* argument of the **`execute()`** method to keep track of the number of transactions executed by the generator to which this scenario is registered. The number of transactions is controlled by the **`stop_after_n_insts`** property of the generator.

```
class my_scenario extends vmm_ms_scenario;
    rand int NUM;
    int SCN_KIND = define_scenario("SCN", 0);
   constraint cst_num {
      NUM inside {[1:10]};
    }
   function new(vmm_ms_scenario parent = null);
     super.new(parent);
     trans = new();
   endfunction
```

```
   task execute(ref int n);
     for (int i=0; i<NUM; i++) begin
         $display(“This is a dummy transaction: %0d”, n);
         n++;
     end
  endtask
endclass
```

Register the scenario to multi-stream scenario generator

Instantiate the scenario created and register it to multi-stream scenario generator object through **`register_scenario()`** method. Generator randomizes the scenario and calls its **`execute()`** method. Any number of scenarios can be registered to a multi-stream generator. It executes scenarios in round robin order by default until either the **`stop_after_n_scenarios`** or **`stop_after_n_insts`** limit is reached.

```
my_scenario  scn = new();
vmm_ms_scenario_gen gen = new(“GEN”);
gen.register_ms_scenario(“MY_SCN”, scn);
gen.stop_after_n_scnearios = 5;
```

Using Logical Channels in Multi-Stream Scenarios

Usually, scenarios put/get transactions through VMM channels. To facilitate this, VMM channels can be registered to the multi-stream generator by name and can be accessed in the **`vmm_ms_scenario`** through its **`get_channel()`** method by specifying the same name.

```
class my_scenario extends vmm_ms_scenario;
   ….
   task execute(ref int n);
    my_trans tr = new;
     vmm_channel my_chan = get_channel(“MY_CHAN”);
     tr.randomize();
```

```
     my_chan.put(tr);
     tr.notify.wait_for(vmm_data::ENDED);
  endtask
endclass

my_scenario  scn = new();
my_trans_channel chan = new(“mychan”, “mychaninst”);
vmm_ms_scenario_gen gen = new(“GEN”);
gen.register_channel(“MY_CHAN”, chan);
gen.register_ms_scenario(“MY_SCN”, scn);
```

Simple Multi-Stream Scenario Generation

The following example shows the basic usage of a multi-stream scenario, where two different kinds of transactions are executed concurrently. You can change the order of execution as needed (dynamic, reactive, and so on).

```
program P;
`include "vmm.sv"
//ALU transaction extending vmm_data
class alu_trans extends vmm_data;
   typedef enum bit [2:0] {ADD=3'b000, SUB=3'b001,
MUL=3'b010, LS=3'b011, RS=3'b100} kind_t;
   rand kind_t kind;
   rand bit [3:0] a, b;
   rand bit [6:0] y;
`vmm_data_member_begin(alu_trans)
   `vmm_data_member_enum(kind, DO_ALL)
   `vmm_data_member_scalar(a, DO_ALL)
   `vmm_data_member_scalar(b, DO_ALL)
   `vmm_data_member_scalar(y, DO_ALL)
`vmm_data_member_end(alu_trans)
endclass
`vmm_channel(alu_trans)


//APB transaction extending vmm_data
class apb_trans extends vmm_data;
  typedef enum bit {READ=1'b0, WRITE=1'b1} kind_e;
```

```
  rand bit [31:0] addr;
  rand bit [31:0] data;
  rand kind_e     kind;
  `vmm_data_member_begin(apb_trans)
     `vmm_data_member_scalar(addr, DO_ALL)
     `vmm_data_member_scalar(data, DO_ALL)
     `vmm_data_member_enum(kind, DO_ALL)
  `vmm_data_member_end(apb_trans)
endclass
`vmm_channel(apb_trans)

//Multi-stream scenario with concurrent execution of 2
transactions of different //stream.
class my_scenario extends vmm_ms_scenario;
   alu_trans_channel alu_chan;
   apb_trans_channel apb_chan;
   rand apb_trans apb_tr; //Transaction gets randomized
                  //when this ms scenario gets randomized
   alu_trans      alu_tr;  //Transaction won't get
randomized
   int MY_SCN = define_scenario(“MY_SCN”, 0);

   function new(vmm_ms_scenario parent=null);
     super.new(parent);
     apb_tr = new();
   endfunction

   virtual task execute(ref int n);
     $cast(alu_chan, this.get_channel("ALU_SCN_CHAN"));
     $cast(apb_chan, this.get_channel("APB_SCN_CHAN"));
     fork
       begin
          alu_trans tr;
          $cast(tr, alu_tr.copy());
          tr.randomize();
          alu_chan.put(tr);
          n++; //Must update the number of transactions
       end
       begin
         apb_trans tr;
         $cast(tr, apb_tr.copy());
         apb_chan.put(tr);
```

```
         n++;       //User must update the number of
transactions.
       end
     join
   endtask

endclass

program P;
`include "vmm.sv"

initial begin
   alu_trans_channel alu_chan = new("ALU_CHAN", "Chan");
   apb_trans_channel apb_chan = new("APB_CHAN", "Chan");
   vmm_ms_scenario_gen gen = new("Gen");
      //Multi-stream scenario generator
   my_scenario scn = new;

   gen.register_channel("ALU_SCN_CHAN", alu_chan);
      //register alu_chan channel to the generator
   gen.register_channel("APB_SCN_CHAN", apb_chan);
      //register apb_chan channel to the generator
   gen.register_ms_scenario("SCN", scn);
      //register multi-stream scenario to the generator
   gen.stop_after_n_scenarios = 5;
   gen.start_xactor();
   fork
     repeat(5) begin
       alu_trans tr;
       alu_chan.get(tr);
       tr.display("ALU:");
     end
     repeat(5) begin
       apb_trans tr;
       apb_chan.get(tr);
       tr.display("APB:");
     end
   join
end

endprogram
```

The example above outputs APB transactions and ALU transactions concurrently five times since the scenario gets executed five times (**`stop_after_n_scenarios`** = 5).

Hierarchical Multi-Stream Scenarios

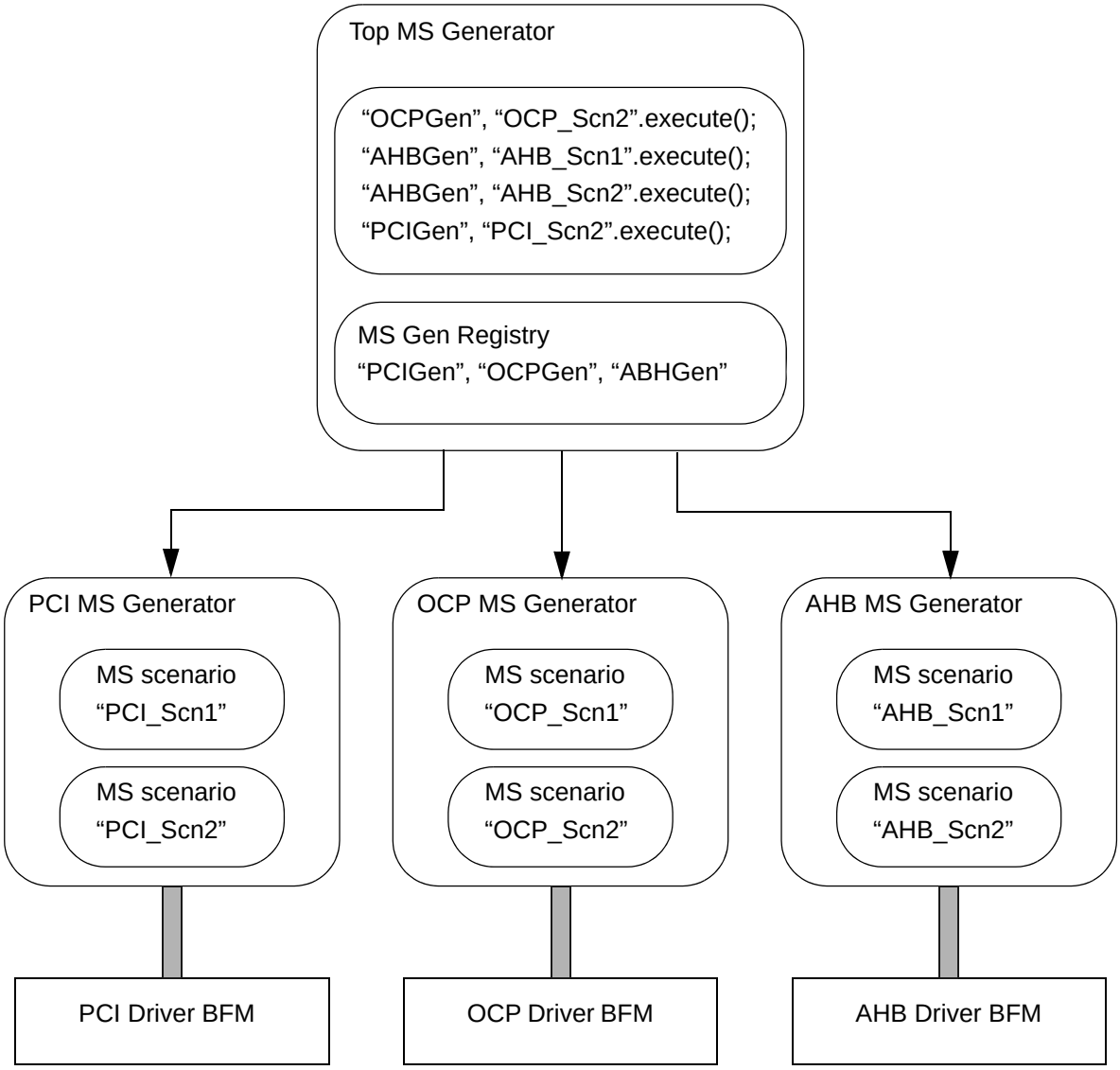
Multi-stream scenarios allow you to reuse existing single stream scenarios and other multi-stream scenarios in a hierarchical way. You can instantiate other scenarios and execute them. In case of a single stream scenario, its **`apply()`** method must be called after it is randomized, and in case of a multi-stream scenario, the **`execute()`** method must be called after randomization. Any registered multi-stream scenario handle can be obtained inside any other scenario by its name. Any level of hierarchy can be achieved using this approach.

```
class my_subsystem_scenario extends vmm_ms_scenario;

   rand cpu_ms_write_scenario  cpu_wr_scn;
   rand dma_single_stream_scenario dma_scn;

   int SCN_KIND = define_scenario(“MSCN”, 0);

   function new(vmm_ms_scenario parent = null);
     super.new(parent);
     cpu_wr_scn = new();
     dma_scn = new();
   endfunction

   task execute(ref int n);
        vmm_ms_scenario cpu_rd_scn =
get_ms_scenario(“CpuReadScn”);
        fork
           begin
               cpu_wr_scn.execute(n);
               void’(cpu_rd_scn.randomize());
               cpu_rd_scn.execute(n);
           end
```

```
            begin
                dma_scn.apply(n);
            end
        join
  endtask
endclass
```

Coordinating Multi-Stream Scenario Generators

Verification at subsystem/system level needs co-ordination between various generators. VMM allows a generator to get registered to another generator and execute the scenarios of registered generators. This enables a top level generator control and synchronize the execution of scenarios of other generators. You can get a handle of scenario of any registered generator by specifying the name of the required scenario and its generator in the **`get_ms_scenario()`** method. Note that the registered generators should not be started, since the top generator is active.




```
class top_scenario extends vmm_ms_scenario;
   int SCN_KIND = define_scenario(“TOP_SCN”, 0);

   task execute(ref int n);
        vmm_ms_scenario ocp_scn2  =
get_ms_scenario(“OCP_SCN2”,  “OCPGen”);
        vmm_ms_scenario ahb_scn1 =
```

```
get_ms_scenario(“AHB_SCN1”, “AHBGen”);
        vmm_ms_scenario ahb_scn2 =
get_ms_scenario(“AHB_SCN2”, “AHBGen”);
        vmm_ms_scenario pci_scn1  =
get_ms_scenario(“PCI_SCN1”, “PCIGen”);

        void`(ocp_scn2.randomize()); ocp_scn2.execute();
        void`(ahb_scn1.randomize()); ahb_scn1.execute();
        void`(ahb_scn2.randomize()); ahb_scn2.execute();
        void`(pci_scn1.randomize());   pci_scn1.execute();

endclass

   ocp_scenario1 ocp_scn1 = new;
   ocp_scenario2 ocp_scn2 = new;
   vmm_ms_scenario_gen ocp_gen = new(“OcpGen”);
   ocp_gen.register_ms_scenario(“OCP_SCN1”, ocp_scn1);
   ocp_gen.register_ms_scenario(“OCP_SCN2”, ocp_scn2);

   ahb_scenario1 ahb_scn1 = new;
   ahb_scenario2 ahb_scn2 = new;
   vmm_ms_scenario_gen ahb_gen = new(“AhbGen”);
   ahb_gen.register_ms_scenario(“AHB_SCN1”, ahb_scn1);
   ahb_gen.register_ms_scenario(“AHB_SCN2”, ahb_scn2);

   pci_scenario1 pci_scn1 = new;
   pci_scenario2 pci_scn2 = new;
   vmm_ms_scenario_gen pci_gen = new(“PciGen”);
   pci_gen.register_ms_scenario(“PCI_SCN1”, pci_scn1);
   pci_gen.register_ms_scenario(“PCI_SCN2”, pci_scn2);

   vmm_ms_scenario_gen, top_gen = new(“TopGen”);
   top_gen.register_ms_scenario_gen(“OCPGen”, ocp_gen);
   top_gen.register_ms_scenario_gen(“AHBGen”, ahb_gen);
   top_gen.register_ms_scenario_gen(“PCIGen”, pci_gen);


   top_gen.start_xactor();
```

A scenario can reserve a channel for exclusive use. The **`vmm_channel::grab()`** method grabs a channel for the exclusive use of a scenario and its sub-scenarios. If the channel is currently grabbed by another scenario, the task will block until the channel can be grabbed by the specified scenario descriptor. The channel will remained grabbed until it is released by calling **`vmm_channel::ungrab().`**

```
task my_ms_scenario::execute(ref int n);
   vmm_channel to_eth0 = get_channel(“ETH0”);
   vmm_channel to_eth1 = get_channel(“ETH1”);

   fork
      begin
         to_eth0.grab(this);
         repeat (10) to_eth0.put(short_fr,.grabber(this));
         to.eth0.ungrab(this);
      end
      begin
         to_eth1.grab(this);
         repeat ( 3) to_eth1.put(long_fr, .grabber(this));
         to_eth1.ungrab(this);
      end
   join
   n += 13;
endtask
```

If a channel has been grabbed by a scenario that is a parent of the specified scenario, then the channel is immediately grabbed by the scenario.

```
task your_ms_scenario::execute(ref int n);
   my_ms_scenario mine = new(this);
   vmm_channel to_eth = get_channel(“ETH0”);

   to_eth.grab(this);
   to_eth.put(bad, .grabber(this));
   mine.execute(n);
   to_eth.put(bad, .grabber(this));
   to_eth.ungrab(this);
   n += 2;
```

```
endtask
```

When a channel is grabbed, the **`vmm_channel::GRABBED`** notification is indicated.

It is important to note that grabbing multiple channels creates a possible deadlock situation. For example, two multi-stream scenarios may attempt to concurrently grab the same multiple channels, but in a different order. This might result in some of the channels to be grabbed by one of the scenario and some of the channels to be grabbed by the other. This would create a deadlock situation because neither scenario would eventually grab the remaining required channels. To avoid this situation, the **`vmm_ms_scenario::grab_channels()`** method should be used to grab multiple channels.

Configuring Scenario Generators

Scenario generators are configured by using many concurrent mechanisms. Any one of these mechanisms can be used by itself or in combination with others to achieve the desired results.

Stopping a Generator

The total number of scenarios to generate is specified by their *`stop_after_n_scenarios`* class property. By default, it is set to zero or an infinite number of scenarios. Example 4-16 shows how a specific scenario generator instance may be configured to automatically stop after generating one scenario.

Example 4-16 Configuring the number of scenarios to generate

```
initial
begin
   env.build();
   env.gen.stop_after_n_scenarios = 1;
   env.run();
end
```

The minimum total number of transactions to generate is specified by their *`stop_after_n_insts`* class property. By default, it is set to zero—or an infinite number of transactions. Example 4-17 shows how all instances of a scenario generator type may be configured to automatically stop after generating at least one hundred transactions.

Example 4-17 Configuring the number of transactions to generate

```
initial
begin
   env.build();
   begin
      ‘foreach_vmm_xactor(ahb_scenario_gen, "/./", "/./")
        xact.stop_after_n_insts = 100;
   end
   env.run();
end
```

Available Scenarios

The scenarios that are available to be generated by the generator must be registered with a generator. By default, single-stream scenario generators only know about the “atomic” scenario, and multi-stream scenario generators do not know about any scenarios. Example 4-18 shows how the default atomic scenario was removed from a single-stream scenario generator instance. Example 4-19 shows how a user-defined scenario can be registered with all instances of a specific scenario generator class.

Example 4-18 Removing scenarios

```
initial
begin
   env.build();
   env.gen.unregister_scenario_by_name("Atomic");
   env.run();
end
```

Example 4-19 Registering scenarios

```
class a_scenario extends
   ahb_scenario;
   ...
endclass

initial
begin
   env.build();
   begin
      ‘foreach_vmm_xactor(ahb_scenario_gen,
                          "/./", "/./") begin
         a_scenario a = new;
         xact.register_scenario("A", a);
      end
   end
   env.run();
end
```

It is also possible to design a verification environment where scenarios can be automatically registered with all instances of the relevant scenario generators. Example 4-20 shows how to implement a type-specific global scenario registry and automatic scenario registration in an environment.

Example 4-20 Automatic scenario registration

```
class auto_ahb_scenario extends ahb_scenario;
   static string names[$];
   static ahb_scenario registry[$];
   static function bit auto_register(string name,
                                     ahb_scenario sc);
      this.names.push_back(name);
      this.registry.push_back(sc);
```

```
   endfunction
endclass

class a_scenario extends auto_ahb_scenario;
   ...
   static local a_scenario _sc = new();
   static local bit _dummy = auto_register("A", this._sc);
endclass

class b_scenario extends auto_ahb_scenario;
   ...
   static local b_scenario _sc = new();
   static local bit _dummy = auto_register("B", this._sc);
endclass

class tb_env extends vmm_env;
   ...
   virtual task build();
      ...
      foreach (auto_ahb_scenario::names[i]) begin
         `foreach_vmm_xactor(ahb_scenario_gen,
                             "/./", "/./")
            xact.register_scenario(
            auto_ahb_scenario::names[i],
            auto_ahb_scenario::registry[i]);
      end
   endtask
   ...
endclass
```

Scenario Generation Order

The next scenario to generate is defined by randomizing their respective *select_scenario* class property. By default, scenarios are selected in a round-robin fashion. The scenario selection can be modified by changing the constraints on the *select* subclass property. Example 4-2 shows how the scenario selection process for a specified generator instance can be configured to randomly select the next scenario. Example 4-21 shows how the scenario selection process for all multi-stream

scenario generator instances can be configured to select one specific scenario, then another specific scenario, then randomly make a selection from the remaining scenarios.

Example 4-21 Configuring the scenario selection process

```
class a_then_b_then_random extends
   vmm_ms_scenario_election;
   constraint round_robin {
      if (scenario_id == 0) select == 0;
      if (scenario_id == 1) select == 1;
      if (scenario_id > 1) select > 1;
   }
endclass

initial
begin
   env.build();
   begin
      a_then_b_then_random sel = new;
      `foreach_vmm_xactor(vmm_ms_scenario_gen,
                          "/./", "/./") begin
         a_scenario a = new;
         b_scenario b = new;
         xact.scenario_set.push_front(b);
         xact.scenario_set.push_front(a);
         xact.select_scenario = sel;
      end
   end
   env.run();
end
```

A “directed” testcase may be implemented by running one “top-level” scenario. This can be accomplished by making sure this top-level scenario will be the first one selected by pushing it at the front of the *`scenario_set`* array and configuring the generator to execute only one scenario. Example 4-22 assumes the existence of a top-level multi-stream scenario generator to execute such a directed testcase.

Example 4-22 Running only one top-level scenario

```
class directed_test extends
```

```
   vmm_ms_scenario;
   ...
endclass

initial
begin
   env.build();
   begin
      directed_test test = new;
      env.top_gen.push_front(test);
      env.top_gen.stop_after_n_scenarios = 1;
   end
   env.run();
end
```

Constraining Transactions

The *items* class property, in single-stream scenario descriptors, implements a two-stage factory for generating random transactions. First, the array is filled with copies of the *using* class property, if it is not null. Once filled, the array is repeatedly randomized and its result content is then copied onto the output channel.

To modify the constraints on all the transactions in a single-stream scenario descriptor, a factory instance should be assigned to the *using* class property, as shown in Example 4-23. Note that it is important that the `copy()` method be properly overloaded in the transaction class extension. This will ensure that the *items* array will be filled with instances of the *using* class property.

Example 4-23 Modifying constraints in all transactions

```
class my_ahb_tr extends ahb_tr;
   constraint my_constraints {
      ...
   }
   `vmm_data_member_begin(my_ahb_tr)
   `vmm_data_member_begin(my_ahb_tr)
endclass
```

```
initial
begin
   env.build();
   begin
      my_ahb_tr tr = new;
      foreach (env.gen.scenario_set[i]) begin
         env.gen.scenario_set[i].using = tr;
      end
   end
   env.run();
end
```

To modify the constraints on a specific transaction in a single-stream scenario descriptor, a factory instance should be assigned to the required *`items`* element, as shown in Example 4-24. The remaining array elements will be filled in with default factory instances.

Example 4-24 Modifying constraints in a specific transaction

```
class my_ahb_tr extends ahb_tr;
   constraint my_constraints {
      ...
   }
   `vmm_data_member_begin(my_ahb_tr)
   `vmm_data_member_begin(my_ahb_tr)
endclass

initial
begin
   env.build();
   begin
      ahb_tr_scenario sc;
      my_ahb_tr tr = new;
      sc = env.gen.get_scenario("Aa");
      sc.items.fill_scenario();
      sc.items[0] = tr;
   end
   env.run();
end
```

Example 4-25 To modify the constraints on the components of a multi-stream scenario descriptor or a hierarchical single-stream scenario descriptor, the randomized class properties should be assigned required instances, as shown in Example 4-25.*Modifying constraints in other scenario descriptors*

```
class my_ahb_tr extends ahb_tr;
   constraint my_constraints {
      ...
   }
   `vmm_data_member_begin(my_ahb_tr)
   `vmm_data_member_begin(my_ahb_tr)
endclass

initial
begin
   env.build();
   begin
      some_scenario sc;
      my_ahb_tr tr = new;
      sc = env.gen.get_scenario("Aa");
      sc.ahb = tr;
   end
   env.run();
end
```

5

VMM Standard Library Customization

The components of the VMM Standard Library are designed to meet the needs of the vast majority of users without additional customization. However, large organizations may wish to customize the components of the VMM Standard Library to offer organization-specific features and capabilities not readily available in the standard version.

You should us the Standard Library customization mechanisms described in this chapter, and in Appendix C, as a last resort. We recommend using the user-defined extension mechanisms provided by the various base and utility classes, such as virtual and callback methods.

Adding to the Standard Library

You can extend the VMM Standard Library by automatically including up to two user-specified files in the `vmm.sv` file. All user-defined Customization are then embedded in the same package as the VMM Standard Library and become automatically visible without further modifications to user code.

If the symbol `'VMM_PRE_INCLUDE` is defined, the file specified by the definition is included at the beginning of the `vmm.sv` file, at the file level, before the VMM standard library package. You can use this symbol to import the pre-processor declarations needed to customize the VMM Standard Library and to define the global customization symbols.

If the symbol `'VMM_POST_INCLUDE` is defined, the file specified by the definition is included at the top of the VMM standard library package, but only after all of the known class names have been defined. You can use this symbol to import declarations and type definitions needed by a customized VMM Standard Library and the implementation of the VMM Standard Library customizations that are built on the predefined classes.

Example 5-1 Inclusion points in the vmm.sv file

```
‘include ‘VMM_PRE_INCLUDE
...
package _vcs_vmm;
   typedef class vmm_xactor;
   ‘ifdef VMM_XACTOR_BASE
      typedef class ‘VMM_XACTOR_BASE
   ‘endif
   ...
   ‘include ‘VMM_POST_INCLUDE
   ...
   class vmm_broadcast extends ‘VMM_XACTOR;
   ...
endpackage
```

NOTE: The symbol definition must include the double quotes surrounding the filename.

Example 5-2 Adding to the VMM Standard Library

```
vcs -sverilog -ntb_opts rvm \
   +define+VMM_PRE_INCLUDE=\"vmm_defines.svh\" \
   +define+VMM_POST_INCLUDE=\"acme_stdlib.sv\" ...
```

Customizing Base Classes

The `vmm_data`, `vmm_channel`, `vmm_xactor` and `vmm_env` base classes are designed to be specialized into different protocol-specific transaction descriptors, transactors and verification environments. A set of organization-specific base classes can be created to introduce organization-specific generic functionality to all VMM components created by that organization, as illustrated in Example 5-3 and Example 5-4.

Example 5-3 Organization-specific transactor base class

```
class acme_xactor extends vmm_xactor;
   ...
endclass: acme_xactor
```

Example 5-4 Transactor based on organization-specific base class

```
class ahb_master extends acme_xactor;
   ...
endclass: ahb_master
```

A problem exists that any VMM component not written by the organization, such as the one shown in Example 5-5, will not be based on that organization's base class. This makes several kinds of features (such as automatically starting all transactor instances when `acme_env::start()` is executed) impossible to create.

Example 5-5 Transactor based on standard base class

```
class ocp_master extends vmm_xactor;
   ...
endclass: ocp_master
```

You can use the following techniques to customize the VMM base classes. Although the techniques are described using the `vmm_xactor` base class, you can be apply them to the `vmm_data` and `vmm_env` base classes as well. The only difference is that their respective symbols would start with `"VMM_DATA"` and `"VMM_ENV"` respectively, instead of `"VMM_XACTOR"`.

Appendix C details the customization macros available with all predefined components in the VMM standard library.

Symbolic Base Class

All VMM-compliant components should be based on the symbolic base class specified by the `‘VMM_XACTOR` symbol, as shown in Example 5-6. Upon compilation, you can redefine the symbol (defined by default to be `"vmm_xactor"`) to cause the transactor to be based on an alternate (but homomorphic) base class, as shown in Example 5-7. This alternate base class should ultimately be based on `vmm_xactor`.

Example 5-6 Transactors based on symbolic base class

```
class ahb_master extends ‘VMM_XACTOR;
   ...
endclass: ahb_master

class ocp_master extends ‘VMM_XACTOR;
   ...
endclass: ocp_master
```

Example 5-7 Redefining the symbolic vmm_xactor base class

```
‘define VMM_XACTOR acme_xactor
```

In the above example, the simple mechanism works if the constructor of the alternate base class has the exact same arguments as the `vmm_xactor` base class. Additional macros are provided to support non-homomorphic constructors.

You should write transactors using the following (see Example 5-8):

- `VMM_XACTOR_NEW_ARGS`
- `VMM_XACTOR_NEW_CALL`

Notice how **a comma does not preceed** each macro. The purpose of this is to handle any instance where the symbols are not defined. It also implies that, whenever these symbols are defined, their definition must start with a comma.

Example 5-8 Transactor supporting non-homomorphic base constructor

```
class ocp_master extends ‘VMM_XACTOR;
   ...
   extern function new(string inst,
                       int    stream_id = -1
                       ‘VMM_XACTOR_NEW_ARGS);
   ...
endclass: ocp_master

function ocp_master::new(string inst,
                         int    stream_id
                         ‘VMM_XACTOR_NEW_ARGS);
   super.new("OCP Master", inst, stream_id
‘VMM_XACTOR_NEW_CALL);
   ...
endfunction: new
```

You can then use an alternate transactor base class by defining the symbolic constructor argument macros appropriately. For example Example 5-9 shows how to use the alternate base class shown in Example 5-10.

Example 5-9 Using a non-homomorphic transactor base class

```
‘define VMM_XACTOR            acme_xactor
‘define VMM_XACTOR_NEW_ARGS  , acme_xactor parent = null, \
                               int key = -1
‘define VMM_XACTOR_NEW_CALL , parent, key
```

In order to be backward compatible with existing VMM-compliant transactors, the first arguments of the alternate base class must match the arguments of the standard `vmm_xactor` base class and provide default argument values for any subsequent arguments, as shown in Example 5-10.

Example 5-10 Backward-compatible alternate base class

```
class acme_xactor extends vmm_xactor;
   ...
   function new(string      name,
                string      inst,
                int         stream_id = -1,
                acme_xactor parent    = null
                int         key       = -1);
      super.new(name, inst, stream_id);
      ...
   endfunction: new
endclass: acme_xactor
```

All predefined transactions, transactors and verification environments in the VMM library (`vmm_broadcast`, `vmm_scheduler`, `vmm_atomic_gen` and `vmm_scenario_gen`) and application packages (`vmm_rw_access`, `vmm_rw_xactor`, `vmm_ral_env`) are written using symbolic base classes and additional constructor arguments. By default, they are based on the standard VMM base classes.

Appendix C details the customization macros available with all predefined components in the VMM standard library. Refer to the User's Guide which corresponds to the VMM application package for the available customization macros.

It is important to note that the implementation of virtual methods are sometimes required to invoke the base class implementation (for example, `vmm_xactor::start_xactor()`) and sometimes may not (for example, `vmm_data::compare()`). When using an alternate `vmm_data` base class, it is important to understand that, except for `vmm_data:copy_data()`, none of the virtual methods in the base class are called by their respective extensions.

Customizing Utility Classes

The vmm_log, vmm_notify and vmm_consensus utility classes are designed to be used as-is when creating verification components, verification environments and test cases. You can create a set of organization-specific utility classes to introduce organization-specific generic functionality to all VMM components, environments and test cases created by that organization, as illustrated in Example 5-11.

Example 5-11 Organization-specific message interface

```
class acme_log extends vmm_log;
   ...
endclass: acme_log
```

A problem exists that any VMM component not written by the organization, such as the standard library component shown in Example 5-12, will not use that organization's utility class. This makes several kind of features impossible to create.

Example 5-12 VMM base class using standard utility class

```
class vmm_xactor;
   vmm_log log;
   ...
endclass: vmm_xactor
```

You can use the following techniques to customize the VMM utility classes. Although the techniques are described using the vmm_log utility class, you can apply them to the vmm_notify and vmm_consensus utility classes as well. The only difference is that their respective symbols would start with "VMM_NOTIFY" and "VMM_CONSENSUS" respectively instead of "VMM_LOG".

Appendix C details the customization macros available with all predefined components in the VMM standard library.

Symbolic Utility Class

All VMM-compliant components should use the symbolic base class specified by the `‘VMM_LOG` symbol, as shown in Example 5-13 and Example 5-14. You can redefine the symbol (defined by default to be `"vmm_log"`) at compile-time to cause the base classes and components to use an alternate (but homomorphic) utility class, as shown in Example 5-15. This alternate utility class should ultimately be based on `vmm_log`.

Example 5-13 VMM Base class using symbolic utility class

```
class vmm_xactor;
   ‘VMM_LOG log;
   ...
endclass: vmm_xactor
```

Example 5-14 Scoreboard using symbolic utility class

```
class scoreboard;
   ‘VMM_LOG log;
   ...
endclass: scoreboard
```

Example 5-15 Redefining the symbolic vmm_log utility class

```
‘define VMM_log acme_log
```

The simple mechanism shown above works if the constructor of the alternate utility class has the exact same arguments as the `vmm_log` utility class.

All predefined elements in the VMM library and application packages are written using symbolic utility classes. By default, they use the standard VMM utility classes.

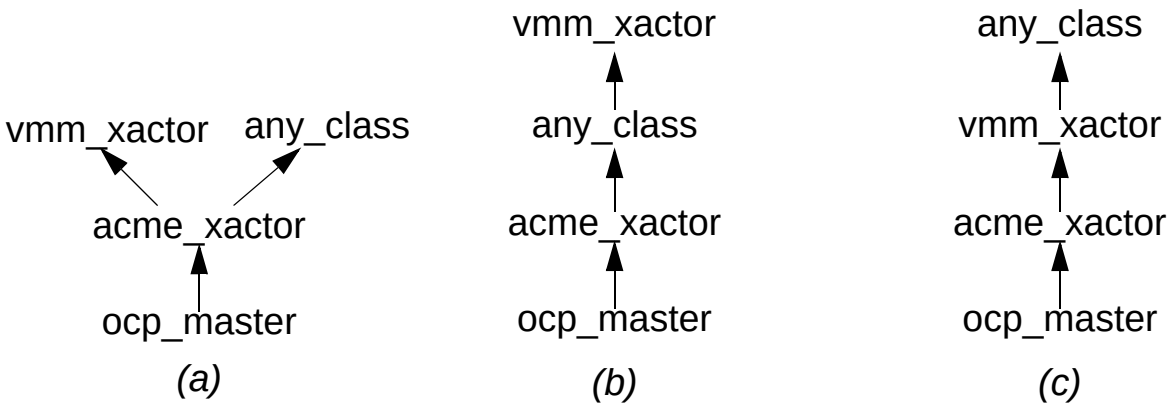
Appendix C details the customization macros available with all predefined components in the VMM standard library. Refer to the User's Guide which corresponds to the appropriate VMM application package for the available customization macros.

Underpinning a Classes

SystemVerilog does not support multiple inheritance. Class inheritance is limited to a single lineage. It may be desirable to have all transactors be derived from more than one base class.

For example, it may be useful to have all transactors derived from the organization-specific transactor base class and the organization-specific `"any class"` base class. Figure 5-1(a) displays how to accomplish this in a language supporting multiple inheritance, such as C++. Figure 5-1(b) and Figure 5-1(c) show two alternative implementations in a single-inheritance language, such as SystemVerilog.

Figure 5-1 Transactor inheriting from more than one class



You can implement the inheritance shown in Figure 5-1(b) by using the `VMM_XACTOR` symbolic base class macros shown in "Customizing Base Classes" on page 5-3. However, this can only be done if the ultimate base class can, in turn, be based on the `vmm_xactor base` class—which is not always possible or sensible.

It is possible to base the VMM Standard Library base and utility classes on a suitable user-defined base class. Although the techniques are described using the `vmm_xactor` base class, they can be applied to the all other base and utility classes defined in the VMM Standard Library as well. The only difference is that their respective symbols would start, for example, with `"VMM_DATA"` and `"VMM_LOG"` respectively instead of `"VMM_XACTOR"`.

A Standard Library base or utility class can be based on a user-defined class by appropriately defining the following macros:

- `VMM_XACTOR_BASE`
- `VMM_XACTOR_BASE_NEW_ARGS`
- `VMM_XACTOR_BASE_NEW_CALL`

- VMM_XACTOR_BASE_METHODS

If you define the VMM_XACTOR_BASE macro, the vmm_xactor base class becomes implemented as shown in Example 5-16.

Example 5-16 Targetable vmm_xactor base class

```
class vmm_xactor extends ‘VMM_XACTOR_BASE;
   ...
   function new(string name,
                string inst,
                int    stream_id = -1
                ‘VMM_XACTOR_BASE_NEW_ARGS);
‘ifdef VMM_XACTOR_BASE_NEW_CALL
      super.new(‘VMM_XACTOR_BASE_NEW_CALL);
‘endif
      ...
   endfunction: new

   ‘VMM_XACTOR_BASE_METHODS

   ...
endclass: vmm_xactor
```

The VMM_XACTOR_BASE symbol is used to define the name of the base class that the vmm_xactor class should be based on.

The VMM_XACTOR_BASE_NEW_ARGS symbol is optionally used to define additional arguments required by the base class constructor if any. When defined, these symbols must include an initial comma.

The VMM_XACTOR_BASE_NEW_CALL symbol is optionally used to define the arguments when calling the base class constructor, if any.

The VMM_XACTOR_BASE_METHODS symbol is optionally used to overload any virtual methods from the base class in the vmm_xactor base class, if any.

Example 5-17 shows how the vmm_xactor base class can be targeted to the base class shown in Example 5-18.

Example 5-17 Underpinning vmm_xactor base class

```
‘define VMM_XACTOR_BASE                    any_class
‘define VMM_XACTOR_BASE_METHODS \
    virtual function string whoami(); \
       return "vmm_xactor"; \
    endfunction: whoami
```

Example 5-18 Ultimate base class

```
virtual class any_class;
   virtual function string whoami();
endclass: any_class
```

If you choose to expose the arguments of the new base class underpinning the vmm_xactor base class to the transactors, you must add the content of the following symbols:

- VMM_XACTOR_BASE_NEW_ARGS
- VMM_XACTOR_BASE_NEW_CALL

...to the following symbols, respectively:

- VMM_XACTOR_NEW_ARGS
- VMM_XACTOR_NEW_CALL

vmm_object

The “vmm_object” base class is an optional VMM Standard Library customization extension that provides a common base underpinning the vmm_data, vmm_scenario, vmm_ms_scenario, vmm_channel, vmm_notify, vmm_xactor, vmm_subenv, vmm_env, vmm_consensus and vmm_test classes.

This optional customization is enabled by including the `vmm_object.svh` and `vmm_object.sv` files at the `‘VMM_PRE_INCLUDE` and `‘VMM_POST_INCLUDE` points respectively:

```
% vcs ... \
   +define+VMM_PRE_INCLUDE=$VMM_HOME/sv/std_lib/opt/vmm_object.svh \
   +define_VMM_POST_INCLUDE=$VMM_HOME/sv/std_lib/opt/vmm_object.sv \
   ...
% vcs ... \
  +define+VMM_PRE_INCLUDE=$VCS_HOME/etc/rvm/sv/std_lib/opt/vmm_object.svh \
  +define_VMM_POST_INCLUDE=$VCS_HOME/etc/rvm/sv/std_lib/opt/vmm_object.sv \
   ...
```

See “vmm_object” on page A-440 for more details on the functionality provided by this optional customization.

Base Classes as IP

The base class underpinning mechanism shown above can be applied recursively to any class hierarchy. This allows the creation of base class IP that can be positioned between two appropriately-written classes.

For example, Example 5-19 shows a VMM-compliant transactor base class provided by company XYZ. Any organization, whose transactor base class has a structure similar to the one shown in Example 5-9, can then leverage that base class by inserting it into their transactor class hierarchy.

By default, this third-party base class should be based on the vmm_xactor base class and can thus be easily inserted between the organization’s transactor base class and the vmm_xactor base class as shown in Example 5-20. But it can also be inserted above the organization’s own transactor base class as shown in Example 5-21.

Example 5-19 Transactor base class IP

```
‘include "vmm.sv"
‘ifndef XYZ_XACTOR_BASE
  ‘define XYZ_XACTOR_BASE          ‘VMM_XACTOR
‘endif
‘ifndef XYZ_XACTOR_BASE_NEW_ARGS
  ‘define XYZ_XACTOR_BASE_NEW_ARGS ‘VMM_XACTOR_NEW_ARGS
  ‘define XYZ_XACTOR_BASE_NEW_CALL ‘VMM_XACTOR_NEW_CALL
‘endif

class xyz_xactor extends XYZ_XACTOR_BASE;
   ...
   function new(string      name,
                string      inst,
                int         stream_id = -1,
                bit         foo       = 0
                ‘XYZ_XACTOR_BASE_NEW_ARGS);
      super.new(name, inst, stream_id
‘XYZ_XACTOR_BASE_NEW_CALL);
      ...
   endfunction: new
   ...
endclass: xyz_xactor
```

Example 5-20 Using base class IP below organization base class

```
‘define ACME_XACTOR_BASE            xyz_xactor
‘define ACME_XACTOR_BASE_NEW_ARGS  , bit foo = 0 \
                                  ‘XYZ_XACTOR_BASE_NEW_ARGS
‘define ACME_XACTOR_BASE_NEW_CALL
```

Example 5-21 Using base class IP above organization base class

```
‘define XYZ_XACTOR_BASE              acme_base
‘define XYZ_XACTOR_BASE_NEW_ARGS    , acme_xactor parent =
null, \
                                      int key = -1
‘define XYZ_XACTOR_BASE_NEW_CALL    , parent, key

‘define VMM_XACTOR                   xyz_xactor
‘define VMM_XACTOR_NEW_ARGS        , bit foo = 0 \
                                  ‘XYZ_XACTOR_BASE_NEW_ARGS
‘define VMM_XACTOR_NEW_CALL       ‘XYZ_XACTOR_BASE_NEW_CALL
```

Customization Macros vs. Parameterized Classes

The following section describes why you would use all of these macros instead of class parameters. For example, instead of using the following:

```
class vmm_xactor
‘ifdef VMM_XACTOR_BASE
   extends ‘VMM_XACTOR_BASE
‘endif
;
```

You could use:

```
class #(type base = vmm_xactor_base) vmm_xactor
   extends base;
```

Parameterized classes are a powerful concept but they are not always the solution. There are several reasons to use macros for base class customization:

- A class parameter is not optional

  If the parameterized form of the `"vmm_xactor"` base class were used, it would always need to be based on another class. It would not be possible to have the `"vmm_xactor"` base class be a primary base class by default, as is the case in the VMM library.

- The constructor of a parameterized class cannot be parameterized

  If the custom base class requires additional constructor arguments, they cannot be added to the `"vmm_xactor"` base class constructor through a parameter. They must be added using macros.

- Virtual method implementations cannot be added through a class parameter

  If the custom base class requires additional virtual method implementations, they cannot be added to the `"vmm_xactor"` base class through a parameter. They must be added using macros.

- A parameterized class must be specialized every time it is customized

  Whenever a parameterized needs to be customized, the parameter values must be specified whenever the class is used. This would make it impossible to automatically have all existing `"vmm_xactor"` extensions be customized on a user-defined customization base class without having to modify every reference to the `"vmm_xactor"` class to specialize it.

A VIP coded, as shown in the following example, can be customized afterwards using macros without requiring any modifications:

```
class ahb_master extends vmm_xactor;
   ...
endclass
```

However, any customization of the "vmm_xactor" class done through class parameters would have required it to be modified, as shown in the following example:

```
class ahb_master extends vmm_xactor#(my_xactor_base);
   ...
endclass
```

For these reasons, the VMM library is customized using macros.

A

Standard Library Classes

This appendix provides detailed information about the classes that compose the VMM Standard Library.

This appendix documents the functionality and features of both OpenVera and SystemVerilog classes, which is identical, except for the following difference:

- OpenVera methods have a prefix of `rvm`
- SystemVerilog methods have a prefix of `vmm`

It is important to note that this appendix uses the SystemVerilog name in the heading to introduce each method.

Additionally, there are a few instances 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 the use of the other language, as they would be almost identical. Rather than use examples that are almost identical, this appendix provides very different examples for 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.

VMM Standard Library Class Summary

vmm_atomic_gen

A macro is used to define a class named *class-name*_**atomic_gen** for any user-specified class derived from **vmm_data**[1], using a process similar to the **‘vmm_channel** macro.

The atomic generator class is an extension of the **vmm_xactor** class and as such, inherits all of the public interface elements provided in the base class.

Summary

1. With a constructor callable without any arguments.

‘vmm_atomic_gen()

Define an atomic generator class.

SystemVerilog

```
‘vmm_atomic_gen(class-name, "Class Description")
```

OpenVera

Not supported.

Description

Defines an atomic generator class named *class-name*_**atomic_gen** to generate instances of the specified class. The generated class must be derived from the **vmm_data** class and the *class-name*_**channel** class must exist.

‘vmm_atomic_gen_using()

Define an atomic generator class.

SystemVerilog

```
‘vmm_atomic_gen_using(class-name, channel-type "Class
Description")
```

OpenVera

Not supported.

Description

Defines an atomic generator class named *class-name*_**atomic_gen** to generate instances of the specified class, with the specified output channel type. The generated class must be compatible with the specified channel type and both must exist.

This macro should be used only when generating instances of a derived class that must be applied to a channel of the base class.

vmm_atomic_gen::new()

Create a new instance of the *class-name*_**atomic_gen** class

SystemVerilog

```
function new(string instance, int stream_id = -1,
    class-name_channel out_chan = null);
```

OpenVera

Not supported.

Description

Creates a new instance of the *class-name*_**atomic_gen** class with the specified instance name and optional stream identifier. The generator can be optionally connected to the specified output channel. If no output channel instance is specified, one will be created internally in the *class-name*_**atomic_gen::out_chan** property.

The name of the transactor is defined as the user-defined class description string specified in the class implementation macro appended with “Atomic Generator”.

Example

Example A-1

```
program t( );
`vmm_atomic_gen(atm_cell, "ATM Cell")
   atm_cell_atomic_gen gen = new("Singleton");
   . . .
endprogram
```

vmm_atomic_gen::class-name_channel out_chan

Reference the output channel for the instances generated by this transactor.

SystemVerilog

```
class-name_channel out_chan;
```

OpenVera

Not supported.

Description

The output channel may have been specified via the constructor. If no output channel instances were specified, a new instance is automatically created. This reference in this property may be dynamically replaced but the generator should be stopped during the replacement.

Example

Example A-2

```
program t( );
`vmm_atomic_gen(atm_cell, "ATM Cell")

   atm_cell_atomic_gen gen = new("Singleton");
   atm_cell cell;
   . . .
   gen.out_chan.get(cell);
   . . .
endprogram
```

vmm_atomic_gen::stop_after_n_insts

Stop after the specified number of object instances has been generated.

SystemVerilog

```
int unsigned stop_after_n_insts;
```

OpenVera

Not supported.

Description

The generator will stop after the specified number of object instances has been generated and consumed by the output channel. The generator must be reset before it can be restarted. If the value of this property is 0, the generator will not stop on its own.

The default value of this property is 0.

Example

Example A-3

```
program t( );
`vmm_atomic_gen(atm_cell, "ATM Cell")

   atm_cell_atomic_gen gen = new("Singleton");
   gen.stop_after_n_insts = 10;
   . . .
endprogram
```

vmm_atomic_gen::randomized_obj

Randomize to create the random content of the output descriptor stream.

SystemVerilog

```
class-name randomized_obj;
```

OpenVera

Not supported.

Description

Transaction or data descriptor instance that is repeatedly randomized to create the random content of the output descriptor stream. The individual instances of the output stream are copied from this instance, after randomization, using the **`vmm_data::copy()`** method.

The atomic generator uses a class factory pattern to generate the output stream instances. The generated stream can be constrained using various techniques on this property.

The **`vmm_data::stream_id`** property of this instance is set to the generator's stream identifier before each randomization. The **`vmm_data::data_id`** property of this instance is also set before each randomization. It will be reset to 0 when the generator is reset and after the specified maximum number of instances has been generated.

Example

Example A-4

```
program test_...;
...
class long_eth_frame extends eth_frame;
  constraint long_frames {
    data.size() == max_len;
  }
endclass: long_eth_frame
...
initial begin
  env.build();
  begin
    long_eth_frame fr = new;
    env.host_src.randomized_obj = fr;
  end
  ...
  top.env.run();
end
endprogram
```

vmm_atomic_gen::enum {GENERATED}

Notification identifier for the notification service interface.

SystemVerilog

```
enum {GENERATED};
```

OpenVera

Not supported.

Description

Notification identifier for the notification service interface in the **`vmm_xactor::notify`** property provided by the **`vmm_xactor`** base class. It is configured as a **`vmm_xactor::ONE_SHOT`** notification and is indicated immediately before an instance is added to the output channel. The generated instance is specified as the status of the notification.

Example

Example A-5

```
gen.notify.wait_for(atm_cell_atomic_gen::GENERATED);
```

vmm_atomic_gen::enum {DONE}

Notification identifier for the notification service interface.

SystemVerilog

```
enum {DONE};
```

OpenVera

Not supported.

Description

Notification identifier for the notification service interface in the **`vmm_xactor::notify`** property provided by the **`vmm_xactor`** base class. It is configured as a **`vmm_xactor::ON_OFF`** notification and is indicated when the generator stops because the specified number of instances has been generated. No status information is specified.

Example

Example A-6

```
gen.notify.wait_for(atm_cell_atomic_gen::DONE);
```

vmm_atomic_gen::inject()

Inject the specified transaction or data descriptor in the output stream.

SystemVerilog

```
virtual task inject(class-name data, ref bit dropped);
```

OpenVera

Not supported.

Description

Unlike injecting the descriptor directly in the output channel, it counts toward the number of instances generated by this generator and will be subjected to the callback methods. The method returns once the instance has been consumed by the output channel or it has been dropped by the callback methods.

This method can be used to inject directed stimulus while the generator is running (with unpredictable timing) or when the generated is stopped.

Example

Example A-7

```
task directed_stimulus;
  eth_frame to_phy, to_mac;
  ...
  to_phy = new;
  to_phy.randomize();
  ...
```

```
    fork
      env.host_src.inject(to_phy, dropped);
      begin
        // Force the earliest possible collision
        @ (posedge tb_top.mii.tx_en);
        env.phy_src.inject(to_mac, dropped);
      end
    join
    ...
    -> env.end_test;
  endtask: directed_stimulus
```

vmm_atomic_gen::post_inst_gen()

Callback invoked after a new transaction or data descriptor has been created.

SystemVerilog

```
virtual task post_inst_gen(class-name_atomic_gen gen,
    class-name data, ref bit drop);
```

OpenVera

Not supported.

Description

Callback method invoked by the generator after a new transaction or data descriptor has been created and randomized but before it is added to the output channel.

The **gen** argument refers to the generator instance that is invoking the callback method (in case the same callback extension instance is registered with more than one transactor instance). The **data** argument refers to the newly generated descriptor— which can be modified. If the value of the **drop** argument is set to non-zero, the generated descriptor will not be forwarded to the output channel, but the remaining registered callbacks will still be invoked.

*class-name*_atomic_gen_callbacks

This class implements a façade for atomic generator, transactor, callback methods. This class is automatically declared and implemented for any user-specified class by the atomic generator macro.

vmm_broadcast

Channels are point-to-point data transfer mechanisms. If multiple consumers are extracting transaction descriptors from a channel, the transaction descriptors are distributed among the various consumers and each of the *N* consumers sees 1/*N* descriptors. If a point-to-multi-point mechanism is required, where all consumers must see all of the transaction descriptors in the stream, a **`vmm_broadcast`** component can be used to replicate the stream of transaction descriptors from a source channel to an arbitrary and dynamic number of output channels. If only two output channels are required, the **`vmm_channel::tee()`** method of the source channel may also be used.

Individual output channels can be configured to receive a copy of the reference to the source transaction descriptor (most efficient but the same descriptor instance is shared by the source and all like-configured output channels) or to use a new descriptor instance copied from the source object (least efficient but uses a separate instance that can be modified without affecting other channels or the original descriptor). A **`vmm_broadcast`** component can be configured to use references or copies in output channels by default.

In the *As Fast As Possible (AFAP)* mode, the full level of the output channels is ignored. Only the full level of the source channel will control the flow of data through the broadcaster. Output channels are kept non-empty as much as possible. As soon as an active output channel becomes empty, the next descriptor is removed from the source channel (if available) and added to all output channels, even if they are already full.

In the *As Late As Possible (ALAP)* mode, the slowest of the output or input channels controls the flow of data through the broadcaster. Only once *all* active output channels are empty, the next descriptor is removed from the source channel (if available) and added to all output channels.

If there are no active output channels, the input channel is continuously drained as transaction descriptors are added to it to avoid data accumulation.

This class is based on the **`vmm_xactor`** class.

Summary

vmm_broadcast::log

Message service interface for this broadcaster.

SystemVerilog

```
vmm_log log;
```

OpenVera

Not supported.

Description

Set by the constructor and uses the name and instance name specified in the constructor.

vmm_broadcast::new()

Create a new instance of a channel broadcaster object.

SystemVerilog

```
function new(string name,
     string instance,
     vmm_channel source,
     bit use_references = 1,
     bcast_mode_typ mode = AFAP);
```

OpenVera

Not supported.

Description

Creates a new instance of a channel broadcaster object with the specified name, instance name, source channel and broadcasting mode. If **`use_references`** is TRUE (that is, non-zero), references to the original source transaction descriptors are assigned to output channels by default (unless individual output channels are configured otherwise).

See the documentation for the **`broadcast_mode()`** method on page A-25 for a description of the available modes.

Example

Example A-8

```
vmm_broadcast bcast = new("Bcast", "", in_chan, 1);
```

vmm_broadcast::start_xactor()

Start this **`vmm_broadcast`** instance.

SystemVerilog

```
virtual function void start_xactor();
```

OpenVera

Not supported.

Description

The broadcaster can be stopped. Any extension of this method must call **`super.start_xactor`***`()`*.

Example

Example A-9

```
vmm_broadcast bcast = new("Bcast", "", in_chan, 1);
bcast.start_xactor();
```

vmm_broadcast::stop_xactor()

Suspend this **`vmm_broadcast`** instance.

SystemVerilog

```
virtual function void stop_xactor();
```

OpenVera

Not supported.

Description

The broadcaster can be restarted. Any extension of this method must call **`super.stop_xactor()`**.

Example

Example A-10

```
program test_directed;
...
initial begin
   ...
   env.start();
   env.host_src.stop_xactor();
   env.phy_src.stop_xactor();
   fork
      directed_stimulus;
   join_none
   env.run();
end

task directed_stimulus;
   ...
endtask: directed_stimulus
```

```
endprogram: test
```

vmm_broadcast::reset_xactor()

Reset this **`vmm_broadcast`** instance.

SystemVerilog

```
virtual function void
           reset_xactor(reset_e rst_type = SOFT_RST);
```

OpenVera

Not supported.

Description

The broadcaster can be restarted. The input channel and all output channels are flushed.

vmm_broadcast::broadcast_mode()

Change the broadcasting mode to the specified mode.

SystemVerilog

```
virtual function void broadcast_mode(bcast_mode_e mode);
```

OpenVera

Not supported.

Description

The new mode takes effect immediately. The available modes are specified by using one of the class-level enumerated symbolic values shown in Table A-1.

Table A-1 Broadcasting Mode Enumerated Values

Enumerated Value	Broadcasting Operation
vmm_broadcast::ALAP	***As Late As Possible***. Data is broadcast *only* when all active output channels are empty. This delay ensures that data is not broadcast any faster than the slowest of all consumers can digest it.
vmm_broadcast::AFAP	***As Fast As Possible.*** Active output channels are kept non-empty as much as possible. As soon as an active output channel becomes empty, the next descriptor from the input channel (if available) is immediately broadcast to all active output channels, regardless of their fill level This mode *must not* be used if the data source can produce data at a higher rate than the slowest data consumer and if broadcast data in all output channels are not consumed at the same average rate.

vmm_broadcast::new_output()

Add the specified channel instance as a new output channel.

SystemVerilog

```
virtual function int new_output(vmm_channel channel,
     logic use_references = 1'bx);
```

OpenVera

Not supported.

Description

Adds the specified channel instance as a new output channel to the broadcaster. If **`use_references`** is TRUE (that is, non-zero), references to the original source transaction descriptor is added to the output channel. If FALSE (that is, zero), a new instance copied from the original source descriptor is added to the output channel. If unknown (that is, 1'bx), the default broadcaster configuration is used.

If there are no output channels, the data from the input channel is continuously drained to avoid data accumulation.

This method returns a unique identifier for the output channel that must be used to modify the configuration of the output channel.

Any user extension of this method must call **`super.new_output()`**.

vmm_broadcast::bcast_on()

Turn broadcasting to the specified output channel on.

SystemVerilog

```
virtual function void bcast_on(int unsigned output-id);
```

OpenVera

Not supported.

Description

By default, broadcasting to an output channel is on. When broadcasting is turned off, the output channel is flushed and the addition of new transaction descriptors from the source channel is inhibited. The addition of descriptors from the source channel is resumed as soon as broadcasting is turned on.

If all output channels are off, the input channel is continuously drained to avoid data accumulation.

Any user extension of these methods should call **`super.bcast_on()`**.

vmm_broadcast::bcast_off()

Turns broadcasting to the specified output channel off.

SystemVerilog

```
virtual function void bcast_off(int unsigned output_id);
```

OpenVera

Not supported.

Description

By default, broadcasting to an output channel is on. When broadcasting is turned off, the output channel is flushed and the addition of new transaction descriptors from the source channel is inhibited. The addition of descriptors from the source channel is resumed as soon as broadcasting is turned on.

If all output channels are off, the input channel is continuously drained to avoid data accumulation.

Any user extension of this method should call **`super.bcast_off()`**.

vmm_broadcast::add_to_output()

Overload to create broadcaster components with different broadcasting rules.

SystemVerilog

```
virtual protected function bit
         add_to_output(int unsigned decision_id,
    int unsigned output_id,
    vmm_channel channel,
    vmm_data obj);
```

OpenVera

Not supported.

Description

Overloading this method allows the creation of broadcaster components with different broadcasting rules. If this function returns TRUE (that is, non-zero), the transaction descriptor will be added to the specified output channel. If this function returns FALSE (that is, zero), the descriptor is not added to the channel. If the output channel is configured to use new descriptor instances, the **`obj`** parameter is a reference to that new instance.

This method is not necessarily invoked in increasing order of output identifiers. It is only called for output channels currently configured as ON. If this method returns FALSE for all output channels for a given broadcasting cycle, lock-up may occur. The **`decision_id`** argument is reset to 0 at the start of every broadcasting cycle and is incremented after each call to this method in the same cycle. It can be used to identify the start of broadcasting cycles.

If transaction descriptors are manually added to output channels, it is important that the **`vmm_channel::sneak()`** method be used to prevent the execution thread from blocking. It is also important that FALSE be returned to prevent that descriptor from being added to that output channel by the default broadcast operations and thus from being duplicated into the output channel.

The default implementation of this method always returns TRUE.

vmm_channel

This class implements a generic transaction-level interface mechanism.

Offset values, either accepted as arguments or returned values, are always interpreted the same way. A value of 0 indicates the head of the channel (first transaction descriptor added). A value of –1 indicates the tail of the channel (last transaction descriptor added). Positive offsets are interpreted from the head of the channel. Negative offsets are interpreted from the tail of the channel. For example, an offset value of –2 indicates the transaction descriptor just before the last transaction descriptor in the channel. It is illegal to specify a non-zero offset that does not correspond to a transaction descriptor already in the channel.

The channel includes an active slot that can be used to create more complex transactor interfaces. The active slot counts toward the number of transaction descriptors currently in the channel for control-flow purposes but cannot be accessed nor specified via an offset specification.

The implementation uses a macro to define a class named *class-name*_**channel** derived from the class named **vmm_channel** for any user-specified class named *class-name*.

Summary

VMM Channel Relationships

VMM extends its VMM channels so that transactors acting as producer or consumer for this channel can be registered.

Hence, it is possible to verify that one unique producer/consumer pair has been attached to a given channel. This insures that no collisions may occur even if user is trying to register new producer or consumer. In addition, while registering channel producer/consumer, corresponding transactors are updated with input/output channels.

Using this class, user can take benefits from built-in transactor uniqueness check and easily traverse transactor channels.

`vmm_channel::set_producer()` identifies the specified transactor as the current producer for the channel instance. This channel will be added to the list of output channels for the transactor. If a producer had been previously identified, the channel instance is removed from the previous producer's list of output channels. Specifying a NULL transactor indicates that the channel has no producer.

Although a channel can have multiple producers—albeit with unpredictable ordering of each producer's contribution to the channel, only one transactor can be identified as a channel's producer as they are primarily a point-to-point transaction-level connection mechanism.

`vmm_channel::set_consumer()` identifies the specified transactor as the current consumer for the channel instance. This channel will be added to the list of input channels for the transactor. If a consumer had been previously identified, the channel instance is removed from the previous consumer's list of input channels. Specifying a NULL transactor indicates that the channel has no consumer.

Although a channel can have multiple consumers—albeit with unpredictable distribution of each consumer's input from the channel, only one transactor can be identified as a channel's

consumer as they are primarily a point-to-point transaction-level connection mechanism. The producer/consumer relationships are set from within transactors.

```
function xact::new(string inst,
                   tr_channel  in_chan = null,
                   obj_channel out_chan = null);
super.new(“Xactor”, inst);
   if (in_chan == null) in_chan = new(…);
   this.in_chan = in_chan;
   this.in_chan.set_consumer(this);
   if (out_chan == null) out_chan = new(…);
   this.out_chan = in_chan;
   this.out_chan.set_producer(this);
endfunction
```

`vmm_channel::get_producer()` returns the transactor that has been specified as the current producer for the channel instance. Returns NULL if no producer has been identified.

`vmm_channel::get_consumer()` return the transactor that has been specified as the current consumer for the channel instance. Returns NULL if no consumer has been identified.

VMM Channel Record/Replay

VMM extends its VMM channels so that incoming transactions can be stored to a file and be replayed from this file later on.

It is possible to replay transactions either on-demand (for example, each time the channel is not blocking), or in a time-accurate way. With the latter option, record/replay can replicate the original channel insertions scheme.

```
virtual task tb_env::start();
   ...
```

```
   if (vmm_opts::get_bit(“record”, “Record generator
output”)) begin
      this.gen.out_chan.record(“gen.dat”);
   end
   if (vmm_opts::get_bit(“play”, “Playback recorded
output”)) begin
      xaction tr = new;
      this.gen.out_chan.playback(ok, “gen.dat”, tr);
   end
   else this.gen.start_xactor();
endtask
```

This feature is very useful to speed up time to debug by shutting down scenario generators. It can also be used to insure the same data stream is always injected to channels.

```
class recorded_scenario extends vmm_ms_scenario;
   virtual task execute(ref int n);
      vmm_channel to_ahb = get_channel(“AHB”);
      ahb_cycle tr = new;
      to_ahb.grab(this);
      fork
         forever begin: count
            to_ahb.notify.wait_for(vmm_channel::PUT);
            n++;
         end
      join_none
      to_ahb.playback(ok, “ahb.dat”, tr, .grabber(this));
      to_ahb.release(this);
      disable count;
   endtask
endclass
```

‘vmm_channel()

Define a channel class to transport instances of the specified class.

SystemVerilog

`‘vmm_channel(`*`class-name`*`)`

OpenVera

Not supported.

Description

The transported class must be derived from the *`vmm_data`* class. This macro is typically invoked in the same file where the specified class is defined and implemented.

This macro creates an external class declaration and no implementation. It is typically invoked when the channel class must be visible to the compiler but the actual channel class declaration is not yet available.

vmm_channel::new()

Create a new instance of a channel with the specified name, instance name and full and empty levels.

SystemVerilog

```
function new(string name,
             string instance,
             int unsigned full = 1,
             int unsigned empty = 0,
             bit fill_as_bytes = 0);
```

OpenVera

Not supported.

Description

If the **`fill_as_bytes`** argument is TRUE (that is, non-zero) the full and empty levels and the fill level of the channel are interpreted as the number of bytes in the channel as computed by the sum of **`vmm_data::byte_size()`** of all transaction descriptors in the channel, not the number of objects in the channel.

If the value is FALSE (that is, zero), the full and empty levels and the fill level of the channel are interpreted as the number of transaction descriptors in the channel.

It is illegal to configure a channel with a full level lower than the empty level.

vmm_channel::log

Message service interface for messages issued from within the channel instance.

SystemVerilog

```
vmm_log log;
```

OpenVera

Not supported.

vmm_channel::reconfigure()

Reconfigure the full or empty levels of the channel.

SystemVerilog

```
function void reconfigure(int full = -1,
    int empty = -1,
    logic fill_as_bytes = 1'bx);
```

OpenVera

Not supported.

Description

If not negative, reconfigure the full or empty levels of the channel to the specified levels . Reconfiguration may cause threads currently blocked on a **`vmm_channel::put()`** call to unblock. If the **`fill_as_bytes`** argument is specified as 1’b1 or 1’b0, the interpretation of the fill level of the channel is modified accordingly. Any other value leaves the interpretation of the fill level unchanged.

Example

Example A-11

```
class consumer extends vmm_xactor;
   transaction_channel in_chan;
   ...
   function new(transaction_channel in_chan = null);
      ...
      if (in_chan == null) in_chan = new(...);
      in_chan.reconfigure(1);
      this.in_chan = in_chan;
   endfunction: new
   ...
```

```
endclass: consumer
```

vmm_channel::full_level()

Return the currently configured full level.

SystemVerilog

```
function int unsigned full_level();
```

OpenVera

Not supported.

vmm_channel::empty_level()

Return the currently configured empty level.

SystemVerilog

```
function int unsigned empty_level();
```

OpenVera

Not supported.

vmm_channel::level()

Return the current fill level of the channel.

SystemVerilog

```
function int unsigned level();
```

OpenVera

Not supported.

Description

The interpretation of the fill level depends on the configuration of the channel instance.

vmm_channel::size()

Return the number of transaction descriptors currently in the channel.

SystemVerilog

```
function int unsigned size();
```

OpenVera

Not supported.

Description

Returns the number of transaction descriptors currently in the channel, including the active slot, regardless of the interpretation of the fill level.

vmm_channel::is_full()

Return an indication of whether the channel is full or not.

SystemVerilog

```
function bit is_full();
```

OpenVera

Not supported.

Description

Returns TRUE (that is, non-zero) if the fill level is greater than or equal to the currently configured full level. Returns FALSE otherwise.

vmm_channel::notify

Indicate the occurrence of events in the channel.

SystemVerilog

```
vmm_notify notify
```

OpenVera

Not supported.

Description

An event notification interface used to indicate the occurrence of significant events within the channel. The notifications shown in Table A-2 are pre-configured

Table A-2 Pre-Configured Notifications in vmm_channel Notifier Interface

Symbolic Property	Corresponding Significant Event
vmm_channel::FULL	Channel has reached or surpassed its configured full level. This notification is configured ON/OFF. No status is returned.
vmm_channel::EMPTY	Channel has reached or underflowed the configured empty level. This event is configured ON/OFF. No status is returned.
vmm_channel::PUT	A new transaction descriptor has been added to the channel. This event is configured ONE_SHOT. The newly added transaction descriptor is available as status.
vmm_channel::GOT	A transaction descriptor has been removed from the channel. This event is configured ONE_SHOT. The newly removed transaction descriptor is available as status.
vmm_channel::PEEKED	A transaction descriptor has been peeked from the channel. This event is configured ONE_SHOT. The newly peeked transaction descriptor is available as status.

Symbolic Property	Corresponding Significant Event
vmm_channel:: ACTIVATED	A transaction descriptor has been transferred to the active slot. This notification also implies a *PEEKED* notification. This event is configured ONE_SHOT. The newly activated transaction descriptor is available as status.
vmm_channel:: ACT_STARTED	The state of a transaction descriptor in the active slot has been updated to *STARTED*. This event is triggered ONE_SHOT. The currently active transaction descriptor is available as status.
vmm_channel:: ACT_COMPLETED	The state of a transaction descriptor in the active slot has been updated to *COMPLETED*. This event is configured ONE_SHOT. The currently active transaction descriptor is available as status.
vmm_channel:: ACT_REMOVED	A transaction descriptor has been removed from the active slot. This notification also implies a *GOT* notification. This event is configured ONE_SHOT. The newly removed transaction descriptor is available as status.
vmm_channel::LOCKED	A side of the channel has been locked. This event is configured ONE_SHOT.
vmm_channel:: UNLOCKED	A side of the channel has been unlocked. This event is configured ONE_SHOT.

vmm_channel::flush()

Flush the content of the channel.

SystemVerilog

```
function void flush();
```

OpenVera

Not supported.

Description

Flushing unblocks any thread currently blocked in the **`vmm_channel::put()`** method. This method will cause the FULL notification to be reset or the EMPTY notification to be indicated. Flushing a channel unlocks all sources and consumers.

vmm_channel::sink()

Flush the content of the channel and sink any further objects put into it.

SystemVerilog

```
function void sink();
```

OpenVera

Not supported.

Description

No transaction descriptors will accumulate in the channel while it is sunk. Any thread attempting to obtain a transaction descriptor from the channel will be blocked until the flow through the channel is restored using the **`vmm_channel::flow()`** method. This method will cause the FULL notification to be reset or the EMPTY notification to be indicated.

vmm_channel::flow()

Restore the normal flow of transaction descriptors through the channel.

SystemVerilog

```
function void flow();
```

OpenVera

Not supported.

vmm_channel::lock()

Block any source (consumer) as if the channel was full (empty) until explicitly unlocked.

SystemVerilog

```
function void lock(bit [1:0] who);
```

OpenVera

Not supported.

Description

The side that is to be locked or unlocked is specified using the sum of the symbolic values shown in Table A-3.

Locking a source does not indicate the **`FULL`** notification, nor does locking the sink indicate the **`EMPTY`** notification, although they have the same control-flow effect.

Table A-3 Channel Endpoint Identifiers

Symbolic Property	Channel Endpoint
vmm_channel::SOURCE	The producer side, i.e., any thread calling the **`vmm_channel::put()`** method
vmm_channel::SINK	The consumer side, i.e., any thread calling the **`vmm_channel::get()`** method

vmm_channel::unlock()

Block any source (consumer) as if the channel was full (empty) until explicitly unlocked.

SystemVerilog

```
function void unlock(bit [1:0] who);
```

OpenVera

Not supported.

Description

The side that is to be locked or unlocked is specified using the sum of the symbolic values shown in Table A-3.

Locking a source does not indicate the **`FULL`** notification, nor does locking the sink indicate the **`EMPTY`** notification, although they have the same control-flow effect.

vmm_channel::is_locked()

Return TRUE (non-zero) if any of the specified sides is locked.

SystemVerilog

```
function bit is_locked(bit [1:0] who);
```

OpenVera

Not supported.

Description

Returns TRUE (that is, non-zero) if any of the specified sides is locked. If both sides are specified, returns TRUE if any side is locked.

Example

Example A-12

```
while (chan.is_locked(vmm_channel::SOURCE +
      vmm_channel::SINK))
   begin
      chan.notify.wait_for(vmm_channel::UNLOCKED);
   end
```

vmm_channel::put()

Put a transaction descriptor in the channel.

SystemVerilog

```
task put(vmm_data obj,
     int offset = -1,
     vmm_scenario grabber = null);
```

OpenVera

```
task put_t(rvm_data obj,
     integer offset = -1);
```

Description

Add the specified transaction descriptor to the channel. If the channel is already full, or becomes full after adding the transaction descriptor, the task will block until the channel becomes empty.

If an offset is specified, the transaction descriptor is inserted in the channel at the specified offset. An offset of 0 specifies at the head of the channel (i.e. LIFO order). An offset of -1 indicate the end of the channel (i.e. FIFO order).

If the channel is currently grabbed by a scenario other than the one specified, this method will block and not insert the specified transaction descriptor in the channel until the channel is ungrabbed or grabbed by the specified scenario.

Example

Example A-13

```
class my_data extends vmm_data;
```

```
   . . .
endclass
`vmm_channel(my_data)

class my_scenario extends vmm_ms_scenario;
   . . .
endclass

program test_grab

   my_data_channel chan = new("Channel", "Grab", 10, 10);
   my_data md1 = new;
   my_scenario scenario_1 = new;

   initial begin
      . . .
      chan.grab(scenario_1);
      chan.put.(md1,scenario_1);
      . . .
   end

endprogram
```

vmm_channel::sneak()

Sneak a transaction descriptor in the channel.

SystemVerilog

```
function void sneak(vmm_data obj,
     int offset = -1,
     vmm_scenario grabber = null);
```

OpenVera

```
task sneak(rvm_data obj,
     integer offset = -1);
```

Description

Add the specified transaction descriptor to the channel. This method will never block, even if the channel is full. An execution thread calling this method must have some other throttling mechanism to prevent an infinite loop from occurring.

This method is designed to be used in circumstances where potentially blocking the execution thread could yield invalid results. For example, monitors must use this method to avoid missing observations.

If an offset is specified, the transaction descriptor is inserted in the channel at the specified offset. An offset of 0 specifies at the head of the channel (for example, LIFO order). An offset of -1 indicate the end of the channel (for example, FIFO order).

If the channel is currently grabbed by a scenario other than the one specified, the transaction descriptor will not be inserted in the channel.

Example

Example A-14

```
class my_data extends vmm_data;
   . . .
endclass
`vmm_channel(my_data)

class my_scenario extends vmm_ms_scenario;
   . . .
endclass

program test_grab

   my_data_channel chan = new("Channel", "Grab", 10, 10);
   my_data md1 = new;
   my_scenario scenario_1 = new;

   initial begin
      . . .
      chan.grab(scenario_1);
      chan.sneak.(md1,scenario_1);
      . . .
   end

endprogram
```

vmm_channel::unput()

Remove the specified transaction descriptor from the channel.

SystemVerilog

```
function class-name unput(int offset = -1);
```

OpenVera

Not supported.

Description

It is an error to specify an offset to a transaction descriptor that does not exist.

This method may cause the **`EMPTY`** notification to be indicated and will cause the **`FULL`** notification to be reset.

vmm_channel::get()

Retrieve the next transaction descriptor in the channel at the specified offset.

SystemVerilog

```
task get(output class-name obj, input int offset = 0);
```

OpenVera

Not supported.

Description

If the channel is empty, the function will block until a transaction descriptor is available to be retrieved. This method may cause the **`EMPTY`** notification to be indicated or the **`FULL`** notification to be reset.

It is an error to invoke this method with an offset value greater than the number of transaction descriptors currently in the channel or with a non-empty active slot.

Example

Example A-15

```
virtual function void build();
   ...
   fork
      forever begin
         eth_frame fr;
         this.mac.rx_chan.get(fr);
         this.sb.received_by_phy_side(fr);
       end
```

```
      join_none
      ...
    endfunction: build
```

vmm_channel::peek()

Get a reference to the next transaction descriptor that will be retrieved from the channel at the specified offset.

SystemVerilog

```
task peek(output class-name obj, input int offset = 0);
```

OpenVera

Not supported.

Description

Gets a reference to the next transaction descriptor that will be retrieved from the channel at the specified offset without actually retrieving it. If the channel is empty, the function will block until a transaction descriptor is available to be retrieved.

It is an error to invoke this method with an offset value greater than the number of transaction descriptors currently in the channel or with a non-empty active slot.

Example

Example A-16

```
class consumer extends vmm_xactor;
   ...
   virtual task main();
      ...
      forever begin
         transaction tr;
         this.in_chan.peek(tr);
         ...
         this.in_chan.get(tr);
```

```
      end
   endtask: main
   ...
endclass: consumer
```

vmm_channel::activate()

Removes the transaction descriptor currently in the active slot.

SystemVerilog

```
task activate(output class-name obj, input int offset = 0);
```

OpenVera

Not supported.

Description

If the active slot is not empty, first removes the transaction descriptor currently in the active slot.

Move the transaction descriptor at the specified offset in the channel to the active slot and update the status of the active slot to **`vmm_channel::PENDING`**. If the channel is empty, this method will wait until a transaction descriptor becomes available. The transaction descriptor is still considered as being in the channel.

It is an error to invoke this method with an offset value greater than the number of transaction descriptors currently in the channel or to use this method with multiple concurrent consumer threads.

Example

Example A-17

```
class consumer extends vmm_xactor;
   ...
   virtual task main();
      ...
      forever begin
```

```
            transaction tr;
            ...
            this.in_chan.activate(tr);
            this.in_chan.start();
            ...
            this.in_chan.complete();
            this.in_chan.remove();
         end
      endtask: main
      ...
endclass: consumer
```

vmm_channel::active_slot()

Return the transaction descriptor currently in the active slot.

SystemVerilog

```
function class-name active_slot();
```

OpenVera

Not supported.

Description

Returns the transaction descriptor currently in the active slot..
Returns *null* if the active slot is empty.

vmm_channel::start()

Update the status of the active slot to **`vmm_channel::STARTED`**.

SystemVerilog

```
function class-name start();
```

OpenVera

Not supported.

Description

The transaction descriptor remains in the active slot. It is an error to call this method if the active slot is empty. The **`vmm_data::STARTED`** notification of the transaction descriptor in the active slot is indicated.

Example

Example A-18

```
class consumer extends vmm_xactor;
   ...
   virtual task main();
      ...
      forever begin
         transaction tr;
         ...
         this.in_chan.activate(tr);
         this.in_chan.start();
         ...
         this.in_chan.complete();
         this.in_chan.remove();
      end
   endtask: main
   ...
```

```
endclass: consumer
```

vmm_channel::complete()

Update the status of the active slot to **`vmm_channel::COMPLETED`**.

SystemVerilog

```
function class-name complete(vmm_data status = null);
```

OpenVera

Not supported.

Description

The transaction descriptor remains in the active slot and may be restarted. It is an error to call this method if the active slot is empty. The **`vmm_data::ENDED`** notification of the transaction descriptor in the active slot is indicated with the optionally specified completion status descriptor.

Example

Example A-19

```
class consumer extends vmm_xactor;
   ...
   virtual task main();
      ...
      forever begin
         transaction tr;
         ...
         this.in_chan.activate(tr);
         this.in_chan.start();
         ...
         this.in_chan.complete();
         this.in_chan.remove();
      end
   endtask: main
```

```
   ...
endclass: consumer
```

vmm_channel::remove()

Update the status of the active slot to **`vmm_channel::INACTIVE`**.

SystemVerilog

```
function class-name remove();
```

OpenVera

Not supported.

Description

Update the status of the active slot to **`vmm_channel::INACTIVE`** and removes the transaction descriptor from the active slot from the channel. This method may cause the **`EMPTY`** notification to be indicated or the **`FULL`** notification to be reset. It an error to call this method with an active slot in the **`vmm_channel::STARTED`** state. The **`vmm_data::ENDED`** notification of the transaction descriptor in the active slot is indicated.

Example

Example A-20

```
class consumer extends vmm_xactor;
   ...
   virtual task main();
      ...
      forever begin
         transaction tr;
         ...
         this.in_chan.activate(tr);
         this.in_chan.start();
         ...
         this.in_chan.complete();
```

```
            this.in_chan.remove();
         end
      endtask: main
      ...
endclass: consumer
```

vmm_channel::status()

Return an enumerated value indicating the status of the transaction descriptor in the active slot.

SystemVerilog

```
function active_status_e status();
```

OpenVera

Not supported.

Description

Returns one of the enumerated values in Table A-4, indicating the status of the transaction descriptor in the active slot.

Table A-4 Pre-Configured Notifications in `vmm_channel` Notifier Interface

Symbolic Property	Corresponding Significant Event
`vmm_channel::INACTIVE`	No transaction descriptor is present in the active slot.
`vmm_channel::PENDING`	A transaction descriptor is present in the active slot but it has not been started yet.
`vmm_channel::STARTED`	A transaction descriptor is present in the active slot and it has been started, but it is not completed yet. The transaction is being processed by the downstream transactor
`vmm_channel::COMPLETED`	A transaction descriptor is present in the active slot and it has been processed by the downstream transactor, but it has not yet been removed from the active slot.

vmm_channel::tee()

Retrieve a copy of the transaction descriptor references that have been retrieved by the **`get()`** or **`activate()`** methods.

SystemVerilog

```
task tee(output class-name obj);
```

OpenVera

Not supported.

Description

When the tee mode is ON, retrieve a copy of the transaction descriptor references that have been retrieved by the **`get()`** or **`activate()`** methods. The task will block until one of the **`get()`** or **`activate()`** methods successfully completes.

This method can be used to fork off a second stream of references to the transaction descriptor stream. Note that the transaction descriptors themselves are not copied. The references returned by this method are referring to the same transaction descriptor instances obtained by the **`get()`** and **`activate()`** methods.

vmm_channel::tee_mode()

Turn the tee mode ON or OFF for this channel.

SystemVerilog

```
function bit tee_mode(bit is_on);
```

OpenVera

Not supported.

Description

Returns TRUE if the tee mode was previously ON. A threads blocked on a call to the **`vmm_channel::tee()`** method will not unblock execution if the tee mode is turned OFF. If the stream of references is not drained via the **`vmm_channel::tee()`** method, data will accumulate in the secondary channel when the tee mode is ON.

vmm_channel::connect()

Connect the output of this channel instance to the input of the specified channel instance.

SystemVerilog

```
function void connect(vmm-channel downstream);
```

OpenVera

Not supported.

Description

The connection is performed with a blocking model to communicate the status of the downstream channel to the producer interface of the upstream channel. Flushing this channel will cause the downstream connected channel to be flushed as well. However, flushing the downstream channel will not flush this channel.

The effective full and empty levels of the combined channels is equal to the sum of their respective levels minus one. However, the detailed blocking behavior of the various interface methods will differ from using a single channel with an equivalent configuration. Additional zero-delay simulation cycles may be required while transaction descriptors are transferred from the upstream channel to the downstream channel.

Connected channels need not be of the same type but must carry compatible polymorphic data.

The connection of a channel into another one can be dynamically modified and broken by connection to a *null* reference. However, modifying the connection while there is data flowing through the channels may yield unpredictable behavior.

vmm_channel::for_each()

Iterate over all of the transaction descriptors currently in the channel.

SystemVerilog

```
function class-name for_each(bit reset = 0);
```

OpenVera

Not supported.

Description

The content of the active slot, if non-empty, is not included in the iteration. If the reset argument is TRUE, a reference to the first transaction descriptor in the channel is returned. Otherwise, a reference to the next transaction descriptor in the channel is returned. Returns *null* when the last transaction descriptor in the channel has been returned. It will keep returning *null* unless reset.

Modifying the content of the channel in the middle of an iteration will yield unexpected results.

vmm_channel::for_each_offset()

Return the offset of the last transaction descriptor returned by the **`vmm_channel::for_each()`** method.

SystemVerilog

```
function int unsigned for_each_offset();
```

OpenVera

Not supported.

Description

Returns the offset of the last transaction descriptor returned by the **`vmm_channel::for_each()`** method. An offset of 0 indicates the first transaction descriptor in the channel.

vmm_channel::record()

Start recording the flow of transaction descriptors.

SystemVerilog

```
function bit record(string filename);
```

OpenVera

Not supported.

Description

Starts recording the flow of transaction descriptors added through the channel instance in the specified file. The **`vmm_data::save()`** method must be implemented for that transaction descriptor and defines the file format. A transaction descriptor is recorded when added to the channel by the **`vmm_channel::put()`** method.

A *`null`* filename stops the recording process. Returns TRUE if the specified file was successfully opened.

vmm_channel::playback()

Playback a recorded transaction stream.

SystemVerilog

```
task playback(output bit success,
     input  string      filename,
     input  vmm_data    factory,
     input  bit         metered = 0,
     input  vmm_scenario grabber = null);
```

OpenVera

```
task playback_t(var bit success,
     string      filename,
     rvm_data    factory,
     bit          metered = 0);
```

Description

Inject the recorded transaction descriptors into the channel in the same sequence in which they were recorded. The transaction descriptors are played back one by one in the order found in the file. The recorded transaction stream replaces the producer for the channel. Playback does not have to happen in the same simulation run as recording: it can be executed in a different simulation run.

You must provide a non-null factory argument, of the same transaction descriptor type as that with which recording was done. The `vmm_data::byte_unpack()` or `vmm_data::load()` method must be implemented for the transaction descriptor passed in to the `factory` argument.

If the `metered` argument is `TRUE`, the transaction descriptors are played back (that is, sneak/put/unput-ed) to the channel in the same relative simulation time interval as the one in which they were originally recorded.

While playing back a recorded transaction descriptor stream on a channel, all other sources of the channel are blocked (for example, `vmm_channel::put()` from any other source be blocked). Transactions added using `vmm_channel::sneak()` would still be allowed from other sources, but a warning will be printed on any such attempt.

The `success` argument is set to `TRUE` if the playback was successful. If the playback process encounters an error condition such as a `NULL` (empty string) filename, a corrupt file or an empty file, then `success` is set to FALSE.

When playback is completed, the `PLAYBACK_DONE` notification is indicated by `vmm_channel::notify`.

If the channel is currently grabbed by a scenario other than the one specified, the playback operation will be blocked until the channel is ungrabbed.

Example

Example A-21

```
class packet_env extends vmm_env;
   ...
   task start();
      ...
      `ifndef PLAY_DATA
         this.gen.start_xactor();
      `else
         fork
            begin
```

```
                bit success;
                data_packet factory = new;
                this.gen.out_chan.playback(success,
                                           "stimulus.dat",
                                           factory, 1);
                if (!this.success) begin
                   `vmm_error(this.log,
                              "Error during playback");
                end
             end
          join_none
       `endif
    endtask
    ...
endclass::packet_env
```

vmm_channel_typed#(type)

Parameterized transaction-level interface.

SystemVerilog

```
class vmm_channel_typed #(type T) extends vmm_channel;
```

OpenVera

Not supported.

Description

Parameterized class implementing a strongly typed transaction-level interface. The specified type parameter *T* must be based on the `vmm_data` base class.

This class is the underlying class corresponding to the *T*`_channel` class that is created when using the `‘vmm_channel(T)` macro. They are both interchangeable. The parameterized class may be used directly without having to declare the strongly-typed channel using the `‘vmm_channel()` macro beforehand.

The parameterized class also allows channels of parameterized classes to be defined without having to define an intermediate `typedef`.

Example

Example A-22 Equivalent definitions

```
‘vmm_channel(eth_frame)
eth_frame_channel in_chan;
```

```
vmm_channel_typed#(eth_frame) in_chan;
```

Example A-23 Equivalent definitions

```
typedef apb_tr#(32, 64) apb_32_64_tr;
‘vmm_channel(apb_32_64_tr)
apb_32_64_tr_channel in_chan;

vmm_channel_typed#(apb_tr#(32, 64)) in_chan;
```

vmm_channel::set_producer()

Specify the current producer for a channel.

SystemVerilog

```
function void set_producer(vmm_xactor producer);
```

OpenVera

Not supported.

Description

Identify the specified transactor as the current producer for the channel instance. This channel will be added to the list of output channels for the transactor. If a producer had been previously identified, the channel instance is removed from the previous producer's list of output channels.

Specifying a `NULL` transactor indicates that the channel has no producer.

Although a channel can have multiple producers—albeit with unpredictable ordering of each producer's contribution to the channel, only one transactor can be identified as a channel's producer as they are primarily a point-to-point transaction-level connection mechanism.

Example

Example A-24

```
class tr extends vmm_data;
 . . .
```

```
endclass
`vmm_channel(tr)
`vmm_scenario_gen(tr, "tr")

program prog;

   initial begin
      tr_scenario_gen sgen = new("Scen Gen");
      tr_channel chan1 = new("tr_channel", "chan1");
      . . .
      chan1.set_producer(sgen);
      . . .
   end
endprogram
```

vmm_channel::set_consumer()

Specify the current consumer for a channel.

SystemVerilog

```
function void set_consumer(vmm_xactor consumer);
```

OpenVera

Not supported.

Description

Identify the specified transactor as the current consumer for the channel instance. This channel will be added to the list of input channels for the transactor. If a consumer had been previously identified, the channel instance is removed from the previous consumer's list of input channels.

Specifying a `NULL` transactor indicates that the channel has no consumer.

Although a channel can have multiple consumers—albeit with unpredictable distribution of each consumer's input from the channel, only one transactor can be identified as a channel's consumer as they are primarily a point-to-point transaction-level connection mechanism.

Example

Example A-25

```
class tr extends vmm_data;
 . . .
```

```
endclass
`vmm_channel(tr)

class xactor extends vmm_xactor;
   . . .
endclass

program prog;

   initial begin
      xactor xact = new("xact");
      tr_channel chan1 = new("tr_channel", "chan1");
      . . .
      chan1.set_consumer(xact);
      . . .
   end
endprogram
```

vmm_channel::get_producer()

Returns the current producer for a channel.

SystemVerilog

```
function vmm_xactor get_producer();
```

OpenVera

Not supported.

Description

Return the transactor that has been specified as the current producer for the channel instance. Returns `NULL` if no producer has been identified.

Example

Example A-26

```
class tr extends vmm_data;
 . . .
endclass
`vmm_channel(tr)

class xactor extends vmm_xactor;
   . . .
endclass

program prog;

   initial begin
      tr_atomic_gen agen = new("Atomic Gen");
      xactor xact = new("Xact", agen.out_chan);
      . . .
```

```
      if (xact.in_chan.get_producer() != agen) begin
      `vmm_error(log, "Wrong producer for xact.in_chan");
      end
      . . .
   end
endprogram
```

vmm_channel::get_consumer()

Returns the current consumer for a channel.

SystemVerilog

```
function vmm_xactor get_consumer();
```

OpenVera

Not supported.

Description

Return the transactor that has been specified as the current consumer for the channel instance. Returns `NULL` if no consumer has been identified.

Example

Example A-27

```
class tr extends vmm_data;
 . . .
endclass
`vmm_channel(tr)

class xactor extends vmm_xactor;
   . . .
endclass

program prog;

   initial begin
      tr_atomic_gen agen = new("Atomic Gen");
      xactor xact = new("Xact", agen.out_chan);
      . . .
```

```
      if (agen.out_chan.get_consumer() != xact) begin
       `vmm_error(log, "Wrong consumer for agen.out_chan");
      end
      . . .
   end
endprogram
```

vmm_channel::grab()

Grab a channel for exclusive use.

SystemVerilog

```
task grab(vmm_scenario grabber);
```

OpenVera

Not supported.

Description

Grab a channel for the exclusive use of a scenario and its sub-scenarios. If the channel is currently grabbed by another scenario, the task will block until the channel can be grabbed by the specified scenario descriptor. The channel will remained grabbed until it is released by calling `vmm_channel::ungrab()`.

If a channel has been grabbed by a scenario that is a parent of the specified scenario, then the channel is immediately grabbed by the scenario.

If exclusive access to a channel is required outside of a scenario descriptor, simply allocate a dummy scenario descriptor and use its reference.

When a channel is grabbed, the `vmm_channel::GRABBED` notification is indicated.

It is important to note that grabbing multiple channels creates a possible deadlock situation. For example, two multi-stream scenarios may attempt to concurrently grab the same multiple

channels, but in a different order. This may result in some of the channels to be grabbed by one of the scenario and some of the channels to be grabbed by the other. This would create a deadlock situation because neither scenario would eventually grab the remaining required channels.

Example

Example A-28

```
class my_data extends vmm_data;
   . . .
endclass
`vmm_channel(my_data)

class my_scenario extends vmm_ms_scenario;
   . . .
endclass

program test_grab

   my_data_channel chan = new("Channel", "Grab", 10, 10);
   my_scenario scenario_1 = new;
   my_scenario scenario_2 = new;

   initial begin
      . . .
      chan.grab(scenario_1);
      . . .
      chan.ungrab(scenario_1);
      chan.grab(scenario_2);
      . . .
   end

endprogram
```

vmm_channel::try_grab()

Try grabbing a channel for exclusive use.

SystemVerilog

```
function bit try_grab(vmm_scenario grabber);
```

OpenVera

Not supported.

Description

Try grabbing a channel for exclusive use and returns TRUE if the channel was successfully grabbed by the scenario. Returns FALSE otherwise.

See `vmm_channel::grab()` for more details on the channel grabbing rules.

Example

Example A-29

```
class my_data extends vmm_data;
   . . .
endclass
`vmm_channel(my_data)

class my_scenario extends vmm_ms_scenario;
   . . .
endclass

program test_grab
```

```
   my_data_channel chan = new("Channel", "Grab", 10, 10);
   my_scenario scenario_1 = new;
   bit grab_success;

   initial begin
      . . .
      grab_success = chan.try_grab(scenario_1);
      if(grab_success == 0)
         `vmm_error(log, "scenario_1 could not grab the
channel");
      else if(parent_grab == 1)
      `vmm_note(log, "scenario_1 has grabbed the channel ");
      . . .
   end

endprogram
```

vmm_channel::ungrab()

Release a channel from exclusive use.

SystemVerilog

```
function void ungrab(vmm_scenario grabber);
```

OpenVera

Not supported.

Description

Release a channel that had been previously grabbed for the exclusive use of a scenario using `vmm_channel::grab()`. If another scenario is waiting to grab the channel, it will be immediately grabbed.

A channel must be explicitly ungrabbed after the execution of an exclusive transaction stream is completed to avoid creating deadlocks.

When a channel is ungrabbed, the `vmm_channel::UNGRABBED` notification is indicated.

Example

Example A-30

```
class my_data extends vmm_data;
   . . .
endclass
`vmm_channel(my_data)
```

```
class my_scenario extends vmm_ms_scenario;
   . . .
endclass

program test_grab

   my_data_channel chan = new("Channel", "Grab", 10, 10);
   my_scenario scenario_1 = new;
   my_scenario scenario_2 = new;

   initial begin
      . . .
      chan.grab(scenario_1);
      . . .
      chan.ungrab(scenario_1);
      chan.grab(scenario_2);
      . . .
   end

endprogram
```

vmm_channel::is_grabbed()

Check if a channel is currently under exclusive use.

SystemVerilog

```
function bit is_grabbed();
```

OpenVera

Not supported.

Description

Returns TRUE if the channel is currently grabbed by a scenario. Returns FALSE otherwise.

Example

Example A-31

```
class my_data extends vmm_data;
   . . .
endclass
`vmm_channel(my_data)

class my_scenario extends vmm_ms_scenario;
   . . .
endclass

program test_grab

   my_data_channel chan = new("Channel", "Grab", 10, 10);
   my_scenario scenario_1 = new;
   bit chan_status;

   initial begin
```

```
        . . .
        chan_status = chan.is_grabbed();
        if(chan_status == 1)
          `vmm_note(log, "The channel is currently grabbed");
        else if(parent_grab == 0)
       `vmm_note(log, "The channel is currently not grabbed ");
        . . .
    end

endprogram
```

vmm_channel::register_vmm_sb_ds()

Refer to the *VMM Scoreboard User Guide*.

vmm_channel::unregister_vmm_sb_ds()

Refer to the *VMM Scoreboard User Guide*.

vmm_channel::kill()

Prepare a channel for deletion.

SystemVerilog

```
function void kill();
```

OpenVera

Not supported.

Description

Prepare the channel for deletion and reclamation by the garbage collector.

Remove this channel instance from the list of input and output channels of the transactors identified as its producer and consumer.

Example

Example A-32

```
program test_grab
   vmm_channel chan;

   initial begin
      chan = new("channel" ,"chan");
      . . .
      chan.kill();
      . . .
   end

endprogram
```

vmm_consensus

This class is used to determine when all of the elements of a testcase, a verification environment or a sub-environment agree that the test may be terminated.

Summary

vmm_consensus::new()

Create a consensus, usually to determine the end-of-test.

SystemVerilog

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

OpenVera

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

Description

Create a new instance of this class with the specified name and instance name. The specified name and instance names are used as the name and instance names of the log class property.

Example

Example A-33

```
program test_consensus;

   vmm_consensus vote = new("Vote", "Main");

   initial begin
      . . .
   end
endprogram
```

vmm_consensus::log

Message service interface for the consensus.

SystemVerilog

```
vmm_log log;
```

OpenVera

```
rvm_log log;
```

Description

This property is set by the constructor using the specified name and instance name. These names may be modified afterward using the `vmm_log::set_name()` or `vmm_log::set_instance()` methods.

Example

Example A-34

```
program test_consensus;

   vmm_consensus vote = new("Vote", "Main");

   initial begin
      . . .
      if (vote.is_reached()) begin
      `vmm_note(vote.log, "Consensus has reached ");
      end else begin
      `vmm_note(vote.log, "Consensus has not reached yet");
      end
      . . .
   end
```

```
endprogram
```

vmm_consensus::register_voter()

Register a new general purpose participant.

SystemVerilog

```
function vmm_voter register_voter(string name);
```

OpenVera

```
function vmm_voter register_voter(string name);
```

Description

Create a new general-purpose voter interface that can participate in this consensus. By default, a voter opposes the end of test. The voter interface may be later unregistered from the consensus using the “vmm_consensus::unregister_voter()” method.

See the “vmm_voter” class for more details on the general-purpose participant interface.

Example

Example A-35

```
program test_consensus;

   vmm_consensus vote = new("Vote", "Main");

   initial begin
      vmm_voter v1;
      . . .
      v1 = vote.register_voter("Voter #1");
      . . .
   end
```

```
endprogram
```

vmm_consensus::unregister_voter()

Unregister a general purpose participant.

SystemVerilog

```
function void unregister_voter(vmm_voter voter);
```

OpenVera

```
task unregister_voter(vmm_voter voter);
```

Description

Remove a previously registered general-purpose voter interface from this consensus. If the voter was the only participant that objected to the consensus, the consensus will subsequently be reached.

Example

Example A-36

```
program test_consensus;

   vmm_consensus vote = new("Vote", "Main");

   initial begin
      vmm_voter v1;
      . . .
      v1 = vote.register_voter("Voter #1");
      . . .
      vote.unregister_voter(v1);
      . . .
   end

endprogram
```

vmm_consensus::register_xactor()

Register a transactor as a participant.

SystemVerilog

```
function void register_xactor(vmm_xactor xact);
```

OpenVera

```
task register_xactor(rvm_xactor xact);
```

Description

Add a transactor that can participate in this consensus. A transactor opposes the end-of-test if it is currently indicating the `vmm_xactor::IS_BUSY` notification, and consents to the end of test, if it is currently indicating the `vmm_xactor::IS_IDLE` notification. The transactor may be later unregistered from the consensus using the “vmm_consensus::unregister_xactor()” method.

Example

Example A-37

```
program test_consensus;

   vmm_consensus vote = new("Vote", "Main");

   initial begin
      vmm_xactor v1 =new("Voter", "#1");
      . . .
      vote.register_xactor(v1);
      . . .
   end
```

```
endprogram
```

vmm_consensus::unregister_xactor()

Unregister a transactor participant.

SystemVerilog

```
function void unregister_xactor(vmm_xactor xact);
```

OpenVera

```
task unregister_xactor(rvm_xactor xact);
```

Description

Remove a previously registered transactor from this consensus. If the transactor was the only participant that objected to the consensus, the consensus will subsequently be reached.

Example

Example A-38

```
program test_consensus;

   vmm_consensus vote = new("Vote", "Main");

   initial begin
      vmm_xactor v1 =new("Voter", "#1");
      . . .
      vote.register_xactor(v1);
      . . .
      vote.unregister_xactor(v1);
      . . .
   end

endprogram
```

vmm_consensus::register_channel()

Register a channel as a participant.

SystemVerilog

```
function void register_channel(vmm_channel chan);
```

OpenVera

```
task register_channel(rvm_channel chan);
```

Description

Add a channel that can participate in this consensus. By default, a channel opposes the end of test if it is not empty, and consents to the end of test if it is currently empty. The channel may be later unregistered from the consensus using the “vmm_consensus::unregister_channel()” method.

Example

Example A-39

```
program test_consensus;

   vmm_consensus vote = new("Vote", "Main");

   initial begin
      vmm_channel v1 =new("Voter", "#1");
      . . .
      vote.register_channel(v1);
      . . .
   end

endprogram
```

vmm_consensus::unregister_channel()

Unregister a channel participant.

SystemVerilog

```
function void unregister_channel(vmm_channel chan);
```

OpenVera

```
task unregister_channel(rvm_channel chan);
```

Description

Remove a previously registered channel from this consensus. If the channel was the only participant that objected to the consensus, the consensus will subsequently be reached.

Example

Example A-40

```
program test_consensus;

   vmm_consensus vote = new("Vote", "Main");

   initial begin
      vmm_channel v1 =new("Voter", "#1");
      . . .
      vote.register_channel(v1);
      . . .
      vote.unregister_channel(v1);
      . . .
   end

endprogram
```

vmm_consensus::register_notification()

Register a notification as a participant.

SystemVerilog

```
function void register_notification(vmm_notify notify,
    int notification);
```

OpenVera

```
task register_notification(rvm_notify notify,
    integer notification);
```

Description

Add an ON/OFF notification that can participate in this consensus. By default, a notification opposes the end of test if it is not indicated, and consents to the end of test if it is currently indicated. The notification may be later unregistered from the consensus using the “vmm_consensus::unregister_notification()” method.

See the “vmm_consensus::register_no_notification()” method for the opposite polarity participation.

Example

Example A-41

```
program test_consensus;

   vmm_consensus vote = new("Vote", "Main");

   initial begin
      vmm_notify v1;
      vmm_log notify_log;
```

```
      notify_log = new ("Voter", "#1");
      v1 = new (notify_log);
      v1.configure(1, vmm_notify::ON_OFF);
      . . .
      vote.register_notification(v1,1);
      . . .
   end

endprogram
```

vmm_consensus::register_no_notification()

Register a notification as a participant.

SystemVerilog

```
function void register_no_notification(vmm_notify notify,
    int notification);
```

OpenVera

```
task register_no_notification(rvm_notify notify,
    integer notification);
```

Description

Add an ON/OFF notification that can participate in this consensus. By default, a notification opposes the end of test if it is indicated, and consents to the end of test if it is not currently indicated. The notification may be later unregistered from the consensus using the "vmm_consensus::unregister_notification()" method.

See the "vmm_consensus::register_notification()" method for the opposite polarity participation.

Example

Example A-42

```
program test_consensus;

   vmm_consensus vote = new("Vote", "Main");

   initial begin
      vmm_notify v1;
      vmm_log notify_log;
```

```
      notify_log = new ("Voter", "#1");
      v1 = new (notify_log);
      v1.configure(1, vmm_notify::ON_OFF);
      . . .
      vote.register_no_notification(v1,1);
      . . .
   end

endprogram
```

vmm_consensus::unregister_notification()

Unregister a notification participant.

SystemVerilog

```
function void unregister_notification(vmm_notify notify,
    int notification);
```

OpenVera

```
task unregister_notification(rvm_notify notify,
    integer notification);
```

Description

Remove a previously registered ON/OFF notification from this consensus. If the notification was the only participant that objected to the consensus, the consensus will subsequently be reached.

Example

Example A-43

```
program test_consensus;

   vmm_consensus vote = new("Vote", "Main");

   initial begin
      vmm_notify v1;
      vmm_log notify_log;
      notify_log = new ("Voter", "#1");
      v1 = new (notify_log);
      v1.configure(1, vmm_notify::ON_OFF);
      . . .
      vote.register_notification(v1,1);
      . . .
      vote.unregister_notification(v1,1);
```

```
      . . .
   end

endprogram
```

vmm_consensus::register_consensus()

Register a sub-consensus as a participant.

SystemVerilog

```
function void register_consensus(vmm_consensus vote
    bit force_through = 0);
```

OpenVera

```
task register_consensus(vmm_consensus vote
    bit force_through = 0);
```

Description

Add a sub-consensus that can participate in this consensus. By default, a sub-consensus opposes the higher-level end of test if it is has not reached its own consensus, and consents to the higher-level end of test if it has reached (or forced) its own consensus. The sub-consensus may be later unregistered from the consensus using the “vmm_consensus::unregister_consensus()” method.

By default, a sub-consensus that has reached its consensus by force will not force a higher-level consensus, only consent to it. If the `force_through` parameter is specified as non-zero, a forced sub-consensus will force a higher-level consensus.

Example

Example A-44

```
program test_consensus;

   vmm_consensus vote = new("Vote", "Main");
```

```
   initial begin
      vmm_consensus c1;
      c1 = new("SubVote", "#1");
      . . .
      vote.register_consensus(c1, 0);
      . . .
   end

endprogram
```

vmm_consensus::unregister_consensus()

Unregister a sub-consensus participant.

SystemVerilog

```
function void unregister_consensus(vmm_consensus vote);
```

OpenVera

```
task unregister_consensus(vmm_consensus vote);
```

Description

Remove a previously registered sub-consensus from this consensus. If the sub-consensus was the only participant that objected to the consensus, the consensus will subsequently be reached. If the sub-consensus was forcing the consensus despite other objections, the consensus will subsequently no longer be reached.

Example

Example A-45

```
program test_consensus;

   vmm_consensus vote = new("Vote", "Main");

   initial begin
      vmm_consensus c1;
      c1 = new("SubVote", "#1");
      . . .
      vote.register_consensus(c1, 0);
      . . .
      vote.unregister_consensus(c1);
      . . .
```

```
   end

endprogram
```

vmm_consensus::wait_for_consensus()

Wait until a consensus has been reached.

SystemVerilog

```
task wait_for_consensus();
```

OpenVera

```
task wait_for_consensus_t();
```

Description

Wait until all participants explicitly consent and none oppose. There can be no abstentions.

If a consensus has already been reached or forced by the time this task is called, this task will return immediately.

A consensus may be broken later (if the simulation is still running) by any voter opposing the end of test or a voter forcing the consensus deciding to consent normally or oppose normally.

Example

Example A-46

```
program test_consensus;

   vmm_consensus vote = new("Vote", "Main");

   initial begin
      . . .
      vote.wait_for_consensus();
      . . .
```

```
    end

endprogram
```

vmm_consensus::wait_for_no_consensus()

Wait until a consensus is no longer reached.

SystemVerilog

```
task wait_for_no_consensus();
```

OpenVera

```
task wait_for_no_consensus_t();
```

Description

Wait until a consensus is broken by no longer being forced and any one participant opposing. If a consensus has not been reached nor forced by the time this task is called, this task will return immediately.

Example

Example A-47

```
program test_consensus;

   vmm_consensus vote = new("Vote", "Main");

   initial begin
      . . .
      vote.wait_for_no_consensus();
      . . .
   end

endprogram
```

vmm_consensus::is_reached()

Check if a consensus has been reached.

SystemVerilog

```
function bit is_reached();
```

OpenVera

```
function bit is_reached();
```

Description

This method returns an indication if a consensus has been reached. If a consensus exists—whether forced or not—a non-zero value is returned. If there is no consensus and the consensus is not being forced, a zero value is returned.

Example

Example A-48

```
program test_consensus;

   vmm_consensus vote = new("Vote", "Main");

   initial begin
      . . .
      if (vote.is_reached())
      `vmm_note (vote.log, "Consensus is reached");
      else
      `vmm_error (vote.log, "Consensus has not reached");
      . . .
   end

endprogram
```

vmm_consensus::is_forced()

Check if a consensus is being forced.

SystemVerilog

```
function bit is_forced();
```

OpenVera

```
function bit is_forced();
```

Description

This method returns an indication if a participant forces a consensus. If the consensus is forced, a non-zero value is returned. If there is no consensus, or the consensus is not being forced, a zero value is returned.

Example

Example A-49

```
program test_consensus;

   vmm_consensus vote = new("Vote", "Main");

   initial begin
      . . .
      if (vote.is_forced())
      `vmm_note (vote.log, "Consensus is forced");
      end
      . . .
   end

endprogram
```

vmm_consensus::psdisplay()

Describe the status of the consensus.

SystemVerilog

```
function string psdisplay(string prefix = "");
```

OpenVera

```
function string psdisplay(string prefix = "");
```

Description

Return a human-readable description of the current status of the consensus and who is opposing or forcing the consensus and why. Each line of the description is prefixed with the specified prefix.

Example

Example A-50

```
program test_consensus;

   vmm_consensus vote = new("Vote", "Main");

   initial begin
      . . .
      $display(vote.psdisplay());
      . . .
   end

endprogram
```

vmm_consensus::yeas()

Return a description of the consenting participants.

SystemVerilog

```
function void yeas(ref string who[],
    ref string why[]);
```

OpenVera

```
task yeas(var string who[*],
    var string why[*]);
```

Description

Return a description of the testbench elements currently consenting to the end of test, and their respective reasons.

Example

Example A-51

```
program test_consensus;

   string who[];
   string why[];
   vmm_consensus vote = new("Vote", "Main");

   initial begin
      . . .
      vote.yeas(who,why);
      for(int i=0; i<who.size; i++)
         $display(" %s ------ %s",who[i],why[i]);
      . . .
   end

endprogram
```

vmm_consensus::nays()

Return a description of the opposing participants.

SystemVerilog

```
function void nays(ref string who[],
    ref string why[]);
```

OpenVera

```
task nays(var string who[*],
    var string why[*]);
```

Description

Return a description of the testbench elements currently opposing to the end of test, and their respective reasons.

Example

Example A-52

```
program test_consensus;

   string who[];
   string why[];
   vmm_consensus vote = new("Vote", "Main");

   initial begin
      . . .
      vote.nays(who,why);
      for(int i=0; i<who.size; i++)
         $display(" %s ------ %s",who[i],why[i]);
      . . .
   end

endprogram
```

vmm_consensus::forcing()

Return a description of the forcing participants.

SystemVerilog

```
function void forcing(ref string who[],
    ref string why[]);
```

OpenVera

```
task forcing(var string who[*],
    var string why[*]);
```

Description

Return a description of the testbench elements currently forcing the end of test, and their respective reasons.

Example

Example A-53

```
program test_consensus;

   string who[];
   string why[];
   vmm_consensus vote = new("Vote", "Main");

   initial begin
      . . .
      vote.forcing(who,why);
      for(int i=0; i<who.size; i++)
         $display(" %s ------ %s",who[i],why[i]);
      . . .
   end

endprogram
```

vmm_data

VMM provides the **vmm_data** class for efficiently modeling transactions. Data modeling can be done more quickly due to unified data encapsulation and by the presence of predefined methods for allocating, copying, comparing, displaying, and byte packing or unpacking of objects

This base class is to be used as the basis for all transaction descriptors and data models. It provides a standard set of methods expected to be found in all descriptors. It also creates a common class—akin to C's *void* type—that can be used to create generic components.

The **vmm_data** class comes with shorthand macros that greatly facilitate data member declaration and provide a quick way to implement the content of predefined methods. Implementing these methods provides an environment for other classes such as **vmm_channel**, **vmm_mss**, **vmm_scoreboard** and so on.

`vmm_data_member_begin() is used to start a shorthand section. The class name specified must be the name of the **vmm_data** extension class that is being implemented.

The shorthand section can only contain shorthand macros and must be terminated by a **`vmm_data_member_end()**, as shown in the following example.

```
class bus_trans extends vmm_data;
   typedef enum bit {READ=1'b0, WRITE=1'b1} kind_e;
   rand bit [31:0] addr;
   rand bit [31:0] data;
   rand kind_e     kind;
   `vmm_data_member_begin(bus_trans)
      `vmm_data_member_scalar(addr, DO_ALL)
```

```
      `vmm_data_member_scalar(data, DO_ALL)
      `vmm_data_member_enum(kind, DO_ALL)
   `vmm_data_member_end(bus_trans)
endclass
`vmm_channel(bus_trans)
`vmm_scenario_gen(bus_trans, "Gen")
```

The example above is for a simple transaction that contains no arrays. Please note that appropriate macros should be used for arrays. Add the specified scalar type, fixed array of scalars, dynamic array of scalars, scalar-indexed associative array of scalars or string-indexed associative array of scalars data member to the default implementation of the methods specified by the **do_what** argument.

```
class eth_frame extends vmm_data;
   vlan_frame vlan_fr_var[] ;
   ...
   `vmm_data_member_begin(eth_frame)
      `vmm_data_member_handle_da(vlan_fr_var, DO_ALL)
      ...
   `vmm_data_member_end(eth_frame)
   ...
endclass
```

vmm_data::do_what_e specifies which methods are to be provided by a shorthand implementation.

```
enum {DO_PRINT, DO_COPY, DO_COMPARE, DO_PACK, DO_UNPACK,
   DO_ALL} do_what_e;
```

It is used to specify which methods are to include the specified data members in their default implementation. Multiple methods can be specified by using **add** in the individual symbolic values. All methods are specified by specifying the **DO_ALL** symbol.

```
‘vmm_data_member_scalar(len, DO_PRINT + DO_COPY +
```

```
DO_COMPARE);
```

It is possible to override the default implementation of the methods created by the **vmm_data** shorthand macros. **vmm_data::do_psdisplay()** overrides the shorthand **psdisplay()** method.

```
virtual function string do_psdisplay(string prefix = "")
```

This method overrides the default implementation of the **vmm_data::psdisplay()** method created by the **vmm_data** shorthand macros. If defined, it will be used instead of the default implementation.

The default implementation of this method in the **vmm_data** base class must not be called (for example, do not call **super.do_psdisplay()**).

The following are shorthand macros and the default implementations they replace:

Shorthand Macro:	**Overrides this default:**
do_is_valid()	is_valid()
do_allocate()	allocate()
do_copy()	copy()
do_compare()	compare()
do_byte_size()	byte_size()
do_max_byte_size()	max_byte_size()
do_byte_pack()	byte_pack()
do_byte_unpack()	byte_unpack()

Summary

vmm_data_new()

Start of explicit constructor implementation.

SystemVerilog

```
‘vmm_data_new(class-name)
```

OpenVera

Not supported.

Description

Specify that an explicit user-defined constructor is used instead of the default constructor provided by the short-hand macros. Also declares a “vmm_log” instance that can be passed to the base class constructor. Use this macro when data members must be explicitly initialized in the constructor.

The class-name specified must be the name of the vmm_data extension class that is being implemented.

This macro should be followed by the constructor declaration and must precede the shorthand data member section i.e., be located before the “‘vmm_data_member_begin()” macro.

Example

Example A-54

```
class eth_frame extends vmm_data;
   ...
   ‘vmm_data_new(eth_frame)
      function new();
         super.new(this.log)
```

```
      ...
    endfunction

  `vmm_data_member_begin(eth_frame)
    ...
  `vmm_data_member_end(eth_frame)
  ...
endclass
```

‘vmm_data_member_begin()

Start of shorthand section.

SystemVerilog

```
‘vmm_data_member_begin(class-name)
```

OpenVera

Not supported.

Description

Start the shorthand section providing a default implementation for the `psdisplay()`, `is_valid()`, `allocate()`, `copy()`, `compare()`, `byte_size()`, `max_byte_size()`, `byte_pack` and `byte_unpack()` methods. A default implementation for the constructor is also provided unless the “`vmm_data_new()`” has been previously specified.

The class-name specified must be the name of the `vmm_data` extension class that is being implemented.

The shorthand section can only contain shorthand macros and must be terminated by a “`‘vmm_data_member_end()`” .

Example

Example A-55

```
class eth_frame extends vmm_data;
   ...
   ‘vmm_data_member_begin(eth_frame)
      ...
   ‘vmm_data_member_end(eth_frame)
```

```
   ...
endclass
```

‘vmm_data_member_end()

End of shorthand section.

SystemVerilog

```
‘vmm_data_member_end(class-name)
```

OpenVera

Not supported.

Description

Terminate the shorthand section providing a default implementation for the `psdisplay()`, `is_valid()`, `allocate()`, `copy()`, `compare()`, `byte_size()`, `max_byte_size()`, `byte_pack` and `byte_unpack()` methods.

The class-name specified must be the name of the `vmm_data` extension class that is being implemented.

The shorthand section must have been started by a “`‘vmm_data_member_begin()`” .

Example

Example A-56

```
class eth_frame extends vmm_data;
   ...
   ‘vmm_data_member_begin(eth_frame)
      ...
   ‘vmm_data_member_end(eth_frame)
   ...
endclass
```

‘vmm_data_member_scalar*()

The shorthand implementation for a scalar data member.

SystemVerilog

```
‘vmm_data_member_scalar(member-name,
                        vmm_data::do_what_e do_what)

‘vmm_data_member_scalar_array(member-name,
                              vmm_data::do_what_e do_what)

‘vmm_data_member_scalar_da(member-name,
                           vmm_data::do_what_e do_what)

‘vmm_data_member_scalar_aa_scalar(member-name,
                              vmm_data::do_what_e do_what)

‘vmm_data_member_scalar_aa_string(member-name,
                              vmm_data::do_what_e do_what)
```

OpenVera

Not supported.

Description

Add the specified scalar-type, fixed array of scalars, dynamic array of scalars, scalar-indexed associative array of scalars or string-indexed associative array of scalars data member to the default implementation of the methods specified by the `do_what` argument.

A scalar is an integral type, such as bit, bit vector, and packed unions.

The shorthand implementation must be located in a section started by “‘vmm_data_member_begin()” .

Example

Example A-57

```
class eth_frame extends vmm_data;
   rand bit [47:0] da;
   ...
   ‘vmm_data_member_begin(eth_frame)
      ‘vmm_data_member_scalar(da, DO_ALL);
      ...
   ‘vmm_data_member_end(eth_frame)
   ...
endclass
```

‘vmm_data_member_string*()

The shorthand implementation for a string data member.

SystemVerilog

```
‘vmm_data_member_string(member-name,
                        vmm_data::do_what_e do_what)

‘vmm_data_member_string_array(member-name,
                              vmm_data::do_what_e do_what)

‘vmm_data_member_string_da(member-name,
                           vmm_data::do_what_e do_what)

‘vmm_data_member_string_aa_scalar(member-name,
                              vmm_data::do_what_e do_what)

‘vmm_data_member_string_aa_string(member-name,
                              vmm_data::do_what_e do_what)
```

OpenVera

Not supported.

Description

Add the specified string-type, fixed array of strings, dynamic array of strings, scalar-indexed associative array of strings or string-indexed associative array of strings data member to the default implementation of the methods specified by the `do_what` argument.

The shorthand implementation must be located in a section started by “‘vmm_data_member_begin()” .

Example

Example A-58

```
class eth_frame extends vmm_data;
   string frame_name;
   ...
   `vmm_data_member_begin(eth_frame)
      `vmm_data_member_string(frame_name, DO_ALL)
      ...
   `vmm_data_member_end(eth_frame)
   ...
endclass
```

vmm_data_member_enum*()

The shorthand implementation for an enumerated data member.

SystemVerilog

```
‘vmm_data_member_enum(member-name,
                      vmm_data::do_what_e do_what)

‘vmm_data_member_enum_array(member-name,
                            vmm_data::do_what_e do_what)

‘vmm_data_member_enum_da(member-name,
                         vmm_data::do_what_e do_what)

‘vmm_data_member_enum_aa_scalar(member-name,
                                vmm_data::do_what_e do_what)

‘vmm_data_member_enum_aa_string(member-name,
                                vmm_data::do_what_e do_what)
```

OpenVera

Not supported.

Description

Add the specified enum-type, fixed array of enums, dynamic array of enums, scalar-indexed associative array of enums or string-indexed associative array of enums data member to the default implementation of the methods specified by the `do_what` argument.

The shorthand implementation must be located in a section started by “`‘vmm_data_member_begin()`” .

Example

Example A-59

```
typedef enum bit[1:0] {NORMAL, VLAN, JUMBO } packet_type;

class eth_frame extends vmm_data;
   rand packet_type packet_type_var;
   . . .
   `vmm_data_member_begin(eth_frame)
      `vmm_data_member_enum (packet_type_var, DO_ALL)
      . . .
   `vmm_data_member_end(eth_frame)
   . . .
endclass
```

‘vmm_data_member_vmm_data*()

The shorthand implementation for a vmm_data-based data member.

SystemVerilog

```
‘vmm_data_member_vmm_data(member-name,
                          vmm_data::do_what_e do_what,
                          vmm_data::do_how_e  do_how)

‘vmm_data_member_vmm_data_array(member-name,
                          vmm_data::do_what_e do_what,
                          vmm_data::do_how_e  do_how)

‘vmm_data_member_vmm_data_da(member-name,
                          vmm_data::do_what_e do_what,
                          vmm_data::do_how_e  do_how)

‘vmm_data_member_vmm_data_aa_scalar(member-name,
                          vmm_data::do_what_e do_what,
                          vmm_data::do_how_e  do_how)

‘vmm_data_member_vmm_data_aa_string(member-name,
                          vmm_data::do_what_e do_what,
                           vmm_data::do_how_e   do_how)
```

OpenVera

Not supported.

Description

Add the specified vmm_data-type, fixed array of vmm_datas, dynamic array of vmm_datas, scalar-indexed associative array of vmm_datas or string-indexed associative array of vmm_datas data member to the default implementation of the methods specified by the `do_what` argument. The `do_how` argument specifies whether the `vmm_data` values must be processed deeply or shallowly.

The shorthand implementation must be located in a section started by “‘vmm_data_member_begin()” .

Example

Example A-60

```
class vlan_frame extends vmm_data;
   . . .
endclass

class eth_frame extends vmm_data;
   vlan_frame vlan_fr_var ;
   . . .
   `vmm_data_member_begin(eth_frame)
    `vmm_data_member_vmm_data(vlan_fr_var, DO_ALL, DO_DEEP)
      . . .
   `vmm_data_member_end(eth_frame)
   . . .
endclass
```

‘vmm_data_member_handle*()

The shorthand implementation for a class handle data member.

SystemVerilog

```
‘vmm_data_member_handle(member-name,
                        vmm_data::do_what_e do_what)

‘vmm_data_member_handle_array(member-name,
                              vmm_data::do_what_e do_what)

‘vmm_data_member_handle_da(member-name,
                           vmm_data::do_what_e do_what)

‘vmm_data_member_handle_aa_scalar(member-name,
                              vmm_data::do_what_e do_what)

‘vmm_data_member_handle_aa_string(member-name,
                              vmm_data::do_what_e do_what)
```

OpenVera

Not supported.

Description

Add the specified handle-type fixed array of handles, dynamic array of handles, scalar-indexed associative array of handles or string-indexed associative array of handles data member to the default implementation of the methods specified by the `do_what` argument.

The shorthand implementation must be located in a section started by `‘vmm_data_member_begin()`.

Example

Example A-61

```
class vlan_frame;
   . . .
endclass

class eth_frame extends vmm_data;
   vlan_frame vlan_fr_var ;
   . . .
   `vmm_data_member_begin(eth_frame)
      `vmm_data_member_handle(vlan_fr_var, DO_ALL)
      . . .
   `vmm_data_member_end(eth_frame)
   . . .
endclass
```

‘vmm_data_member_user_defined()

User-defined shorthand implementation data member.

SystemVerilog

```
‘vmm_data_member_user_defined(member-name,
                              vmm_data::do_what_e do_what)
```

OpenVera

Not supported.

Description

Add the specified user-defined default implementation of the methods specified by the `do_what` argument.

Refer to the section entitled, “User-defined vmm_data Member Default Implementation” on page 2-6 for details on how to specify the shorthand implementation for a data member.

The shorthand implementation must be located in a section started by `“‘vmm_data_member_begin()”` .

Example

Example A-62

```
class eth_frame extends vmm_data;
   rand bit [47:0] da;
   . . .
   `vmm_data_member_begin(eth_frame)
      `vmm_data_member_user_defined(da, DO_ALL)
      . . .
   `vmm_data_member_end(eth_frame)
```

```
   . . .
   function bit do_da ( input vmm_data::do_what_e do_what)

      do_da = 1;  // Success, abort by returning 0

      case (do_what)
      . . .
      endcase
   endfunction
endclass
```

‘vmm_data_byte_size()

The shorthand implementation packing size methods.

SystemVerilog

```
‘vmm_data_byte_size(max-expr, size-expr)
```

OpenVera

Not supported.

Description

Provide a default implementation of the `byte_size()` and `max_byte_size()` methods. The first and second expressions specify the value returned by the `max_byte_size()` and `byte_size()` methods respectively. The expression must be a valid SystemVerilog expression in the content of the class.

The shorthand implementation must be located immediately after the “‘vmm_data_member_end()” .

Example

Example A-63

```
class eth_frame extends vmm_data;
   ...
   ‘vmm_data_member_begin(eth_frame)
      ...
   ‘vmm_data_member_end(eth_frame)
   ‘vmm_data_byte_size(1500, this.len_typ+16)
   ...
endclass
```

vmm_data::new()

Create a new instance of this data model or transaction descriptor.

SystemVerilog

```
function new(vmm_log log);
```

OpenVera

Not supported.

Description

Creates a new instance of this data model or transaction descriptor with the specified message service interface. The specified message service interface is used when constructing the **`vmm_data::notify`** property.

Example

Example A-64

Because of the potentially large number of instances of data objects, a *`class-static`* message service interface should be used to minimize memory usage and to be able to control class-generic messages:

```
class eth_frame extends vmm_data {
   static vmm_log log = new(“eth_frame”, “class”);
   function new()
      super.new(this.log);
      ...
   endfunction
endclass: eth_frame
```

vmm_data::log

Replace the message service interface for this instance of a data model or transaction descriptor.

SystemVerilog

```
function vmm_log set_log(vmm_log log);
```

OpenVera

Not supported.

Description

Replaces the message service interface for this instance of a data model or transaction descriptor with the specified message service interface and returns a reference to the previous message service interface. Can be used to associate a descriptor with the message service interface of a transactor currently processing the transaction or to set the service when it was not available during initial construction.

vmm_data::stream_id

Unique identifier for a data model or transaction descriptor instance

SystemVerilog

```
int stream_id;
```

OpenVera

Not supported.

Description

Specifies the offset of the descriptor within a sequence and the sequence offset within a stream. This property must be set by the transactor that instantiates the descriptor. It is set by the predefined generator before randomization so it can be used to specify conditional constraints to express instance-specific or stream-specific constraints.

vmm_data::scenario_id

Unique identifier for a data model or transaction descriptor instance

SystemVerilog

```
int scenario_id;
```

OpenVera

Not supported.

Description

Specifies the offset of the descriptor within a sequence and the sequence offset within a stream. This property must be set by the transactor that instantiates the descriptor. It is set by the predefined generator before randomization so it can be used to specify conditional constraints to express instance-specific or stream-specific constraints.

vmm_data::data_id

Unique identifier for a data model or transaction descriptor instance

SystemVerilog

```
int data_id;
```

OpenVera

Not supported.

Description

Specifies the offset of the descriptor within a sequence and the sequence offset within a stream. This property must be set by the transactor that instantiates the descriptor. It is set by the predefined generator before randomization so it can be used to specify conditional constraints to express instance-specific or stream-specific constraints.

vmm_data::notify

A notification service interface with three pre-configured events.

SystemVerilog

```
vmm_notify notify;
enum {EXECUTE;
     STARTED;
     ENDED};
```

OpenVera

Not supported.

Description

The **EXECUTE** notification is ON/OFF and indicated by default. It can be used to prevent the execution of a transaction or the transfer of data if reset. The **STARTED** and **ENDED** notifications are ON/OFF events and indicated by the transactor at the start and end of the transaction execution or data transfer. The meaning and timing of the notifications is specific to the transactor executing the transaction described by this instance

vmm_data::display()

Display the current value of the transaction or data.

SystemVerilog

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

OpenVera

Not supported.

Description

Displays the current value of the transaction or data described by this instance in a human-readable format on the standard output. Each line of the output will be prefixed with the specified prefix. This method prints the value returned by the **`psdisplay()`** method.

vmm_data::psdisplay()

Return an image of the current value of the transaction or data.

SystemVerilog

```
virtual function string psdisplay(string prefix = "");
```

OpenVera

Not supported.

Description

Returns an image of the current value of the transaction or data described by this instance in a human-readable format as a string. The string may contain newline characters to split the image across multiple lines. Each line of the output must be prefixed with the specified prefix.

vmm_data::is_valid()

Check the current value of the transaction or data.

SystemVerilog

```
virtual function bit is_valid(bit silent = 1,
    int kind = -1);
```

OpenVera

Not supported.

Description

Checks if the current value of the transaction or data described by this instance is valid and error free, according to the optionally specified kind or format. Returns TRUE (that is, non-zero) if the content of the object is valid. Returns FALSE otherwise. The meaning (and use) of the **`kind`** argument is descriptor-specific and defined by the user extension of this method.

If **`silent`** is TRUE (that is, non-zero), no error or warning messages are issued if the content is invalid. If **`silent`** is FALSE, warning or error messages may be issued if the content is invalid.

vmm_data::allocate()

Allocate a new instance.

SystemVerilog

```
virtual function vmm_data allocate();
```

OpenVera

Not supported.

Description

Allocates a new instance of the same type as the object instance. Returns a reference to the new instance. Useful to implement class factories to create instances of user-defined derived class in generic code written using the base class type.

vmm_data::copy()

Copy the current value of the object instance.

SystemVerilog

```
virtual function vmm_data copy(vmm_data to = null);
```

OpenVera

Not supported.

Description

Copies the current value of the object instance to the specified object instance. If no target object instance is specified, a new instance is allocated. Returns a reference to the target instance.

Example

Example A-65

The following trivial implementation will not work. Constructor copying is a shallow copy. The objects instantiated in the object (such as those referenced by the log and notify properties) are not copied and both copies will share references to the same service interfaces. Furthermore, it will not properly handle the case when the to argument is not null.

Invalid implementation of the **`vmm_data::copy()`** method:

```
function vmm_data atm_cell::copy(vmm_data to = null)
   copy = new this;
endfunction
```

The following implementation is usually preferable.

Proper implementation of the **`vmm_data::copy()`** method:

```
function vmm_data atm_cell::copy(vmm_data to = null)
   atm_cell cpy;

   if (to != null) begin
      if ($cast(cpy, to)) begin
         ‘vmm_fatal(log, "Not an atm_cell instance");
         return null;
      end
   end else cpy = new;

   this.copy_data(cpy);
   cpy.vpi = this.vpi;
   ...
   copy = cpy;
endfunction: copy
```

The base-class implementation of this method must not be called as it contains error detection code of a derived class that forgot to supply an implementation. The **`vmm_data::copy_data()`** method should be called instead.

vmm_data::copy_data()

Copy the current value of all base class data properties.

SystemVerilog

```
virtual protected function void copy_data(vmm_data to);
```

OpenVera

Not supported.

Description

Copies the current value of all base class data properties in the current data object into the specified data object instance. This method should be called by the implementation of the **`vmm_data::copy()`** method in classes immediately derived from this base class.

vmm_data::compare()

Compare the current object instance with the specified object instance.

SystemVerilog

```
virtual function bit compare(input vmm_data to,
     output string diff,
     input int kind = -1);
```

OpenVera

Not supported.

Description

Compares the current value of the object instance with the current value of the specified object instance, according to the specified kind. Returns TRUE (that is, non-zero) if the value is identical. If the value is different, FALSE is returned and a descriptive text of the first difference found is returned in the specified *`string`* variable. The **`kind`** argument may be used to implement different comparison functions (for example, full compare, comparison of *`rand`* properties only, comparison of all properties physically implemented in a protocol and so on.)

Example

Example A-66

```
function bit check(eth_frame actual)
   sb_where_to_find_frame where;
   eth_frame              q[$];
   eth_frame              expect;
```

```
   check = 0;
   if (!index_tbl[hash(actual)].exists()) return;
   where = index_tbl[hash(actual)];
   q = sb.port[where.port_no].queue[where.queue_no];
   expect = q.pop_front();
   if (actual.compare(expect)) check = 1;
endfunction: check
```

vmm_data::byte_pack()

Pack the content of the transaction or data into a dynamic array of bytes.

SystemVerilog

```
virtual function int unsigned byte_pack(
    ref logic [7:0] bytes[],
    input int unsigned offset = 0,
    input int kind = -1);
```

OpenVera

Not supported.

Description

Packs the content of the transaction or data into the specified dynamic array of bytes, starting at the specified offset in the array. The array is resized appropriately. Returns the number of bytes added to the array.

If the data can be interpreted or packed in different ways, the *`kind`* argument can be used to specify which interpretation or packing to use.

vmm_data::byte_unpack()

Unpack the specified number of bytes of data.

SystemVerilog

```
virtual function int unsigned byte_unpack(
    const ref logic [7:0] bytes[],
    input int unsigned offset = 0,
    input int len = -1,
    input int kind = -1);
```

OpenVera

Not supported.

Description

Unpacks the specified number of bytes of data from the specified offset in the specified dynamic array into this descriptor. If the number of bytes to unpack is specified as *-1*, the maximum number of bytes will be unpacked. Returns the number of bytes unpacked. If there is not enough data in the dynamic array to completely fill the descriptor, the remaining properties are set to unknown and a warning may be issued.

If the data can be interpreted or unpacked in different ways, the **`kind`** argument can be used to specify which interpretation or packing to use.

Example

Example A-67

```
class eth_frame extends vmm_data;
   ...
```

```
   typedef enum {UNTAGGED, TAGGED, CONTROL}
      frame_formats_e;
   rand frame_formats_e format;
   ...
   rand bit [47:0] dst;
   rand bit [47:0] src;
   rand bit        cfi;
   rand bit [ 2:0] user_priority;
   rand bit [11:0] vlan_id;
   ...
   virtual function int unsigned byte_unpack(
      const ref logic [7:0] array[],
      input int unsigned    offset = 0,
      input int             len    = -1,
      input int             kind   = -1);
      integer i;

      i = offset;
      this.format = UNTAGGED;
      ...
      if ({array[i], array[i+1]} === 16'h8100) begin
         this.format = TAGGED;
         i += 2;
         ...
         {this.user_priority, this.cfi, this.vlan_id} =
            {array[i], array[i+2]};
         i += 2;
         ...
      end
      ...
   endfunction: byte_unpack
   ...
endclass: eth_frame
```

vmm_data::byte_size()

Return the number of bytes required to pack the content of this descriptor.

SystemVerilog

```
virtual function int unsigned byte_size(int kind = -1);
```

OpenVera

Not supported.

Description

Returns the number of bytes required to pack the content of this descriptor. This method will be more efficient than **`vmm_data::byte_pack()`** for simply knowing how many bytes are required by the descriptor because no packing is actually done.

If the data can be interpreted or packed in different ways, the **`kind`** argument can be used to specify which interpretation or packing to use.

vmm_data::max_byte_size()

Return the maximum number of bytes required to pack the content of this descriptor.

SystemVerilog

```
virtual function int unsigned max_byte_size(
    int kind = -1);
```

OpenVera

Not supported.

Description

Returns the maximum number of bytes that will ever be required to pack the content of any instance of this descriptor. A value of 0 indicates an unknown maximum size. Can be used to allocate memory buffers in the DUT or verification environment of suitable sizes.

If the data can be interpreted or packed in different ways, the *`kind`* argument can be used to specify which interpretation or packing to use.

vmm_data::save()

Append the content of this descriptor to the specified file.

SystemVerilog

```
virtual function void save(int file);
```

OpenVera

Not supported.

Description

Appends the content of this descriptor to the specified file. The format is user defined and may be binary. By default, simply packs the descriptor and saves the value of the bytes, in sequence, as binary values and terminated by a newline.

vmm_data::load()

Set the content of this descriptor.

SystemVerilog

```
virtual function bit load(int file);
```

OpenVera

Not supported.

Description

Sets the content of this descriptor from the data in the specified file. The format is user defined and may be binary. By default, interprets a complete line as binary byte data and unpacks it.

Should return FALSE (i.e., zero) if the loading operation was not successful.

vmm_data::do_what_e

Specifies which methods are to be provided by a shorthand implementation.

SystemVerilog

```
enum {DO_PRINT, DO_COPY, DO_COMPARE,
      DO_PACK, DO_UNPACK, DO_ALL} do_what_e;
```

OpenVera

Not supported.

Description

Used to specify which methods are to include the specified data members in their default implementation. Multiple methods can be specified by using an `add` or an `or` in the individual symbolic values. All methods are specified by specifying the `DO_ALL` symbol.

Example

Example A-68

```
‘vmm_data_member_scalar(len,
                        DO_PRINT + DO_COPY + DO_COMPARE);
```

vmm_data::do_how_e

Specifies how vmm_data references are interpreted by a shorthand implementation.

SystemVerilog

```
enum {DO_NOCOPY, DO_NOCOMPARE, DO_NONE,
      DO_REFCOPY, DO_REFCOMPARE, DO_REF,
      DO_DEEPCOPY, DO_DEEPCOMPARE, DO_DEEP} do_what_e;
```

OpenVera

Not supported.

Description

Used to specify how the copy and compare methods deal with a reference to a vmm_data instance in their default implementation. Multiple mechanisms can be specified by using an add or an or in the individual symbolic values. Following are the meanings of the DO_NONE, DO_REF, and DO_DEEP symbols:

- DO_NONE - skip all comparison and copy operations
- DO_REF - use the reference itself in comparison and copy operations
- DO_DEEP - do deep compare and deep copy operations

Example

Example A-69

```
‘vmm_data_member_vmm_data(parent, DO_ALL, DO_REF);
```

vmm_data::do_psdisplay()

Override the shorthand `psdisplay()` method.

SystemVerilog

```
virtual function string do_psdisplay(string prefix = "")
```

OpenVera

Not supported.

Description

This method overrides the default implementation of the `vmm_data::psdisplay()` method created by the `vmm_data` shorthand macros. If defined, it will be used instead of the default implementation.

The default implementation of this method in the `vmm_data` base class **must not** be called (for example, do not call `super.do_psdisplay()`).

Example

Example A-70

```
class eth_frame extends vmm_data;
   . . .
   virtual function string do_psdisplay(string prefix = "")
    $sformat(psdisplay, "%sEthernet Frame #%0d.%0d.%0d:\n",
             prefix, this.stream_id, this.scenario_id,
             this.data_id);
      . . .
   endfunction
endclass
```

vmm_data::do_is_valid()

Override the shorthand `is_valid()` method.

SystemVerilog

```
virtual bit function do_is_valid(bit silent = 1,
                                 int kind  = -1);
```

OpenVera

Not supported.

Description

This method overrides the default implementation of the `vmm_data::is_valid()` method created by the `vmm_data` shorthand macros. If defined, it will be used instead of the default implementation.

The default implementation of this method in the `vmm_data` base class **must not** be called (for example, do not call `super.do_is_valid()`).

Example

Example A-71

```
class eth_frame extends vmm_data;
   . . .
   virtual bit function do_is_valid(bit silent = 1,
         int kind = -1);
      do_is_valid = 1;
      if (!do_is_valid && !silent) begin
      `vmm_error(this.log, "Ethernet Frame is not valid");
      end
```

```
   endfunction
   . . .
endclass
```

vmm_data::do_allocate()

Override the shorthand `allocate()` method.

SystemVerilog

```
virtual vmm_data function do_allocate();
```

OpenVera

Not supported.

Description

This method overrides the default implementation of the `vmm_data::allocate()` method created by the `vmm_data` shorthand macros. If defined, it will be used instead of the default implementation.

The default implementation of this method in the `vmm_data` base class **must not** be called (for example, do not call `super.do_allocate()`).

Example

Example A-72

```
class eth_frame extends vmm_data;
   . . .
   virtual vmm_data function do_allocate ();
      `ifdef ETH_USE_COMPOSITION
     eth_frame i = new;
        i.vlan = new
        do_allocate = 1;
      `else
     eth_frame i = new;
```

```
            do_allocate = i;
         `endif
      endfunction
      . . .
endclass
```

vmm_data::do_copy()

Override the shorthand `copy()` method.

SystemVerilog

```
virtual vmm_data function copy(vmm_data to = null);
```

OpenVera

Not supported.

Description

This method overrides the default implementation of the `vmm_data::copy()` method created by the `vmm_data` shorthand macros. If defined, it will be used instead of the default implementation.

The default implementation of this method in the `vmm_data` base class **must not** be called (for example, do not call `super.do_copy()`).

Example

Example A-73

```
class eth_frame extends vmm_data;
   . . .
   virtual vmm_data function do_copy(vmm_data to = null);
      eth_frame cpy;
      if (to != null) begin
         if (!$cast(cpy, to)) begin
       `vmm_error(this.log, "Cannot copy to non-eth_frame\n
           instance");
        return null;
```

```
      end
       end else cpy = new;
       . . .
       `ifdef ETH_USE_COMPOSITION
      if (this.vlan != null) begin
          cpy.vlan = new;
         cpy.vlan.user_priority = this.vlan.user_priority;
          cpy.vlan.cfi           = this.vlan.cfi;
          cpy.vlan.id            = this.vlan.id;
        end
       `else
         cpy.user_priority = this.user_priority;
         cpy.cfi           = this.cfi;
         cpy.vlan_id       = this.vlan_id;
       `endif
       . . .
       do_copy = cpy;
   endfunction
   . . .
endclass
```

vmm_data::do_compare()

Override the shorthand **`compare()`** method.

SystemVerilog

```
virtual bit function do_compare(input  vmm_data to = null,
                                output string diff,
                                input int kind  = -1);
```

OpenVera

Not supported.

Description

This method overrides the default implementation of the `vmm_data::compare()` method created by the `vmm_data` shorthand macros. If defined, it will be used instead of the default implementation.

The default implementation of this method in the `vmm_data` base class **must not** be called (for example, do not call `super.do_compare()`).

Example

Example A-74

```
class eth_frame extends vmm_data;
   . . .
   virtual bit function do_compare(input vmm_data to =
         null,output string diff, input int kind = -1);
      eth_frame fr;
      do_compare = 1;
      . . .
```

```
      `ifdef ETH_USE_COMPOSITION
     if (fr.vlan == null) begin
         diff = "No vlan data";
       do_compare = 0;
      end

      if (fr.vlan.user_priority !==
           this.vlan.user_priority) begin
       $sformat(diff, "user_priority (3'd%0d !== 3'd%0d)",
                 this.vlan.user_priority,
                 fr.vlan.user_priority);
       do_compare = 0;
      end
     . . .
      `else
     if (fr.user_priority !== this.user_priority) begin
       $sformat(diff, "user_priority (3'd%0d !== 3'd%0d)",
                 this.user_priority, fr.user_priority);
       do_compare = 0;
      end
     . . .
      `endif
     . . .
   endfunction
endclass
```

vmm_data::do_byte_size()

Override the shorthand `byte_size()` method.

SystemVerilog

```
virtual int function do_byte_size(int kind  = -1);
```

OpenVera

Not supported.

Description

This method overrides the default implementation of the `vmm_data::byte_size()` method created by the `vmm_data` shorthand macros. If defined, it will be used instead of the default implementation.

The default implementation of this method in the `vmm_data` base class **must not** be called (for example, do not call `super.do_byte_size()`).

Example

Example A-75

```
class eth_frame extends vmm_data;
   . . .
   virtual int function do_byte_size(int kind = -1);
      `ifdef TAGGED
      do_byte_size = 14 + data.size();
      `else
      do_byte_size = 14 + data.size() + 4;
      `endif
   endfunction
```

```
   . . .
endclass
```

vmm_data::do_max_byte_size()

Override the shorthand `max_byte_size()` method.

SystemVerilog

```
virtual int function do_max_byte_size(int kind  = -1);
```

OpenVera

Not supported.

Description

This method overrides the default implementation of the `vmm_data::max_byte_size()` method created by the `vmm_data` shorthand macros. If defined, it will be used instead of the default implementation.

The default implementation of this method in the `vmm_data` base class **must not** be called (for example, do not call `super.do_max_byte_size()`).

Example

Example A-76

```
class eth_frame extends vmm_data;
   . . .
   virtual int function do_max_byte_size(int kind = -1);
      `ifdef JUMBO_PACKET
      do_max_byte_size = 9000;
      `else
      do_max_byte_size = 1500;
      `endif
   endfunction
```

```
   . . .
endclass
```

vmm_data::do_byte_pack()

Override the shorthand `byte_pack()` method.

SystemVerilog

```
virtual int function do_byte_pack(ref logic [7:0] bytes[],
                                  input int unsigned offset = 0,
                                 input int kind = -1);
```

OpenVera

Not supported.

Description

This method overrides the default implementation of the `vmm_data::byte_pack()` method created by the `vmm_data` shorthand macros. If defined, it will be used instead of the default implementation.

The default implementation of this method in the `vmm_data` base class **must not** be called (for example, do not call `super.do_byte_pack()`).

Example

Example A-77

```
class eth_frame extends vmm_data;
   . . .
   virtual int function do_byte_pack(ref logic [7:0]
        bytes[],input int unsigned offset = 0,
        input int kind = -1);
      int i;
      . . .
```

```
      `ifdef ETH_USE_COMPOSITION
        {bytes[i], bytes[i+1]} = {this.vlan.user_priority,
            this.vlan.cfi, this.vlan.id};
      `else
         {bytes[i], bytes[i+1]} = {this.user_priority,
            this.cfi, this.vlan_id};
      `endif
      . . .
   endfunction

endclass
```

vmm_data::do_byte_unpack()

Override the shorthand `byte_unpack()` method.

SystemVerilog

```
virtual int function do_byte_unpack(
                            const ref logic [7:0] bytes[],
                            input int unsigned offset = 0,
                            input int len = -1,
                           input int kind = -1);
```

OpenVera

Not supported.

Description

This method overrides the default implementation of the `vmm_data::byte_unpack()` method created by the `vmm_data` shorthand macros. If defined, it will be used instead of the default implementation.

The default implementation of this method in the `vmm_data` base class **must not** be called (for example, do not call `super.do_byte_unpack()`).

Example

Example A-78

```
class eth_frame extends vmm_data;
   . . .
  virtual int function do_byte_unpack(const ref logic [7:0]
       bytes[],input int unsigned offset = 0,
       input int len = -1,input int kind = -1);
```

```
      . . .
      `ifdef ETH_USE_COMPOSITION
      {this.vlan.user_priority, this.vlan.cfi,
          this.vlan.id} = {bytes[i], bytes[i+1]};
      `else
      {this.user_priority, this.cfi, this.vlan_id} =
         {bytes[i], bytes[i+1]};
      `endif
      . . .
   endfunction
   . . .
endclass
```

vmm_env

This class is a base class used to implement verification environments.

Summary

vmm_env::log

Message service interface for the verification environment.

SystemVerilog

```
vmm_log log;
```

OpenVera

Not supported.

Description

This property is set by the constructor using the specified environment name and may be modified at run time.

vmm_env::notify

Notification service interface and predefined notifications.

SystemVerilog

```
vmm_notify notify;
enum {GEN_CFG,
     BUILD,
     RESET_DUT,
     CFG_DUT,
     START,
     WAIT_FOR_END,
     STOP,
     CLEANUP,
     REPORT};
```

OpenVera

Not supported.

Description

Notification service interface and predefined notifications used to indicate the progression of the verification environment. The predefined notifications are used to signal the start of the corresponding predefined virtual methods. All notifications are ON/OFF.

vmm_env::new()

Create an instance of the verification environment.

SystemVerilog

```
function new(string name = “Verif Env”);
```

OpenVera

Not supported.

Description

Creates an instance of the verification environment, with the specified name. The name is used as the name of the message service interface.

vmm_env::run()

Run the simulation.

SystemVerilog

```
task run()
```

OpenVera

Not supported.

Description

Runs all remaining steps of the simulation, including **`vmm_env::stop()`**, **`vmm_env::cleanup()`** and **`vmm_env::report()`**. This method must be explicitly invoked in the test programs.

vmm_env::gen_config()

Randomize the test configuration descriptor.

SystemVerilog

```
virtual function void gen_cfg();
```

OpenVera

Not supported.

Description

If this method has not been explicitly invoked in the test program, it will be implicitly invoked by the **`vmm_env::build()`** method.

vmm_env::build()

Build the verification environment.

SystemVerilog

```
virtual function void build();
```

OpenVera

Not supported.

Description

Builds the verification environment according to the value of the test configuration descriptor. If this method has not been explicitly invoked in the test program, it will be implicitly invoked by the **`vmm_env::reset_dut()`** method.

Example

Example A-79

```
class my_test extends vmm_test;
   ...
   virtual task run(vmm_env env);
      tb_env my_env;
      $cast(my_env, env);
      my_env.build();
      my_env.gen[0].start_xactor();
      my_env.run();
   endtask
endclass
```

vmm_env::reset_dut()

Reset the DUT to make it ready for configuration.

SystemVerilog

```
virtual task reset_dut();
```

OpenVera

Not supported.

Description

Physically resets the DUT to make it ready for configuration. If this method has not been explicitly invoked in the test program, it will be implicitly invoked by the **`vmm_env::cfg_dut()`** method.

vmm_env::cfg_dut()

Configure the DUT.

SystemVerilog

```
virtual task cfg_dut();
```

OpenVera

Not supported.

Description

Configures the DUT according to the value of the test configuration descriptor. If this method has not been explicitly invoked in the test program, it will be implicitly invoked by the **`vmm_env::start()`** method.

vmm_env::start()

Start the test.

SystemVerilog

```
virtual task start();
```

OpenVera

Not supported.

Description

Starts all the components of the verification environment to start the actual test. If this method has not been explicitly invoked in the test program, it will be implictly invoked by the **`vmm_env::wait_for_end()`** method.

vmm_env::end_test

Cause the **`vmm_env::wait_for_end()`** method to return.

SystemVerilog

```
event end_test;
```

OpenVera

Not supported.

Description

Event that, when triggered, should cause the **`vmm_env::wait_for_end()`** method to return. It is up to the user-defined implementation of the **`vmm_env::wait_for_end()`** method to detect that this event has been triggered and return.

vmm_env::wait_for_end()

Wait for an indication that the test has reached completion.

SystemVerilog

```
virtual task wait_for_end();
```

OpenVera

Not supported.

Description

Waits for an indication that the test has reached completion or its objective—whatever these may be. When this task returns, it signals that the end of simulation condition has been detected. If this method has not been explicitly invoked in the test program, it will be implictly invoked by the **`vmm_env::stop()`** method.

Example

Example A-80

```
class tb_env extends vmm_env;
   ...
   virtual task wait_for_end();
      super.wait_for_end();
      ...
      wait (this.cfg.run_for_n_tx_frames == 0 &&
            this.cfg.run_for_n_tx_frames == 0);
      ...
   endtask: wait_for_end
   ...
endclass: tb_env
```

vmm_env::stop()

Terminate the simulation cleanly.

SystemVerilog

```
virtual task stop();
```

OpenVera

Not supported.

Description

Stops all the components of the verification environment to terminate the simulation cleanly. If this method has not been explicitly invoked in the test program, it will be implicitly invoked by the **`vmm_env::cleanup()`** method.

vmm_env::cleanup()

Perform clean-up operations.

SystemVerilog

```
virtual task cleanup();
```

OpenVera

Not supported.

Description

Performs clean-up operations to let the simulation terminate gracefully. Clean-up operations may include letting the DUT drain off all buffered data, reading statistics registers in the DUT and sweeping the scoreboard for leftover expected responses. If this method has not been explicitly invoked in the test program, it will be implicitly invoked by the **`vmm_env::run()`** method.

vmm_env::report()

Report success or failure of the test and close all files.

SystemVerilog

```
virtual task report();
```

OpenVera

Not supported.

Description

Reports final success or failure of the test and close all files. If this method has not been explicitly invoked in the test program, it will be implicitly invoked by the **`vmm_env::run()`** method.

vmm_env::end_vote

End-of-test consensus object.

SystemVerilog

```
vmm_consensus end_vote;
```

OpenVera

```
vmm_consensus end_vote;
```

Description

Predefined end-of-test consensus instance that can be used in the extension of the `vmm_env::wait_for_end()` method to determine that the simulation has reached its logical end. The name of the consensus is the name of the environment specified in the `vmm_env` constructor. The instance name of the consensus is "End-of-test Consensus".

Triggering the `vmm_env::end_test` event does not force the consensus. A consensus does not trigger the `end_test` event. This class property and the `end_test` event are not functionally related in the base class.

Example

Example A-81

```
initial begin
   apb_env env;
  vmm_voter test_voter = env.end_vote.register_voter("Test
case Stimulus");
   . . .
end
```

‘vmm_env_member_begin()

Start of shorthand section.

SystemVerilog

```
‘vmm_env_member_begin(class-name)
```

OpenVera

Not supported.

Description

Start the shorthand section providing a default implementation for the `psdisplay()`, `start()` and `stop()` methods.

The class-name specified must be the name of the `vmm_env` extension class that is being implemented.

The shorthand section can only contain shorthand macros and must be terminated by a “‘`vmm_env_member_end()`” .

Example

Example A-82

```
class tb_env extends vmm_env;
   ...
   ‘vmm_env_member_begin(tb_env)
      ...
   ‘vmm_env_member_end(tb_env)
   ...
endclass
```

‘vmm_env_member_end()

End of shorthand section.

SystemVerilog

```
‘vmm_env_member_end(class-name)
```

OpenVera

Not supported.

Description

Terminate the shorthand section providing a default implementation for the psdisplay(), start() and stop() methods.

The class-name specified must be the name of the vmm_env extension class that is being implemented.

The shorthand section must have been started by a “‘vmm_env_member_begin()” .

Example

Example A-83

```
class my_env extends vmm_env;
    . . .
   `vmm_env_member_begin(my_vmm_env)
   `vmm_env_member_end(my_vmm_env)
    . . .
endclass
```

‘vmm_env_member_scalar*()

Shorthand implementation for a scalar data member.

SystemVerilog

```
‘vmm_env_member_scalar(member-name,
                       vmm_env::do_what_e do_what)

‘vmm_env_member_scalar_array(member-name,
                             vmm_env::do_what_e do_what)

‘vmm_env_member_scalar_aa_scalar(member-name,
                             vmm_env::do_what_e do_what)

‘vmm_env_member_scalar_aa_string(member-name,
                             vmm_env::do_what_e do_what)
```

OpenVera

Not supported.

Description

Add the specified scalar-type, array of scalars, scalar-indexed associative array of scalars or string-indexed associative array of scalars data member to the default implementation of the methods specified by the `’do_what’` argument.

A scalar is an integral type, such as bit, bit vector, and packed unions.

The shorthand implementation must be located in a section started by a “‘vmm_env_member_begin()” .

Example

Example A-84

```
class my_vmm_env extends vmm_env;
   bit [31:0] address;
   . . .
   `vmm_env_member_begin(my_vmm_env)
      `vmm_env_member_scalar(address,DO_ALL)
      . . .
   `vmm_env_member_end(my_vmm_env)
   . . .
endclass
```

‘vmm_env_member_string*()

Shorthand implementation for a string data member.

SystemVerilog

```
‘vmm_env_member_string(member-name,
                       vmm_env::do_what_e do_what)

‘vmm_env_member_string_array(member-name,
                             vmm_env::do_what_e do_what)

‘vmm_env_member_string_aa_scalar(member-name,
                                 vmm_env::do_what_e do_what)

‘vmm_env_member_string_aa_string(member-name,
                                 vmm_env::do_what_e do_what)
```

OpenVera

Not supported.

Description

Add the specified string-type, array of strings, scalar-indexed associative array of strings or string-indexed associative array of strings data member to the default implementation of the methods specified by the `’do_what’` argument.

The shorthand implementation must be located in a section started by a “‘vmm_env_member_begin()” .

Example

Example A-85

```
class my_vmm_env extends vmm_env;
```

```
   string name;
   . . .
   `vmm_env_member_begin(my_vmm_env)
      `vmm_env_member_string(name,DO_PRINT)
      . . .
   `vmm_env_member_end(my_vmm_env)
   . . .
endclass
```

‘vmm_env_member_enum*()

Shorthand implementation for an enumerated data member.

SystemVerilog

```
‘vmm_env_member_enum(member-name,
                     vmm_env::do_what_e do_what)

‘vmm_env_member_enum_array(member-name,
                           vmm_env::do_what_e do_what)

‘vmm_env_member_enum_aa_scalar(member-name,
                               vmm_env::do_what_e do_what)

‘vmm_env_member_enum_aa_string(member-name,
                               vmm_env::do_what_e do_what)
```

OpenVera

Not supported.

Description

Add the specified enum-type, array of enums, scalar-indexed associative array of enums or string-indexed associative array of enums data member to the default implementation of the methods specified by the `’do_what’` argument.

The shorthand implementation must be located in a section started by a “‘vmm_env_member_begin()” .

Example

Example A-86

```
typedef enum {blue,green,red,black} my_colors;
```

```
class my_vmm_env extends vmm_env;
   my_colors  color;
   . . .
   `vmm_env_member_begin(my_vmm_env)
      `vmm_env_member_enum(color,DO_PRINT)
      . . .
   `vmm_env_member_end(my_vmm_env)
   . . .
endclass
```

‘vmm_env_member_vmm_data*()

Shorthand implementation for a vmm_data-based data member.

SystemVerilog

```
‘vmm_env_member_vmm_data(member-name,
                         vmm_env::do_what_e do_what)

‘vmm_env_member_vmm_data_array(member-name,
                         vmm_env::do_what_e do_what)

‘vmm_env_member_vmm_data_aa_scalar(member-name,
                         vmm_env::do_what_e do_what)

‘vmm_env_member_vmm_data_aa_string(member-name,
                         vmm_env::do_what_e do_what)
```

OpenVera

Not supported.

Description

Add the specified vmm_data-type, array of vmm_datas, scalar-indexed associative array of vmm_datas or string-indexed associative array of vmm_datas data member to the default implementation of the methods specified by the `’do_what’` argument.

The shorthand implementation must be located in a section started by a “`‘vmm_env_member_begin()`” .

Example

Example A-87

```
class my_data extends vmm_data;
   . . .
endclass : my_data

class my_vmm_env extends vmm_env;
   my_data   data;
   . . .
   `vmm_env_member_begin(my_vmm_env)
      `vmm_env_member_vmm_data(data,DO_PRINT)
      . . .
   `vmm_env_member_end(my_vmm_env)
   . . .
endclass
```

‘vmm_env_member_channel*()

Shorthand implementation for a channel data member.

SystemVerilog

```
‘vmm_env_member_channel(member-name,
                            vmm_env::do_what_e do_what)

‘vmm_env_member_channel_array(member-name,
                                  vmm_env::do_what_e do_what)

‘vmm_env_member_channel_aa_scalar(member-name,
                                  vmm_env::do_what_e do_what)

‘vmm_env_member_channel_aa_string(member-name,
                                  vmm_env::do_what_e do_what)
```

OpenVera

Not supported.

Description

Add the specified channel-type, array of channels, dynamic array of channels, scalar-indexed associative array of channels or string-indexed associative array of channels data member to the default implementation of the methods specified by the `’do_what’` argument.

The shorthand implementation must be located in a section started by a “`‘vmm_env_member_begin()`” .

Example

Example A-88

```
class my_vmm_env extends vmm_env;
   my_data_channel my_channel;
   . . .
   `vmm_env_member_begin(my_vmm_env)
      `vmm_env_member_channel(my_channel,DO_ALL);
      . . .
   `vmm_env_member_end(my_vmm_env)
   . . .
endclass
```

‘vmm_env_member_xactor*()

Shorthand implementation for a transactor data member.

SystemVerilog

```
‘vmm_env_member_xactor(member-name,
                           vmm_env::do_what_e do_what)

‘vmm_env_member_xactor_array(member-name,
                                 vmm_env::do_what_e do_what)

‘vmm_env_member_xactor_aa_scalar(member-name,
                                 vmm_env::do_what_e do_what)

‘vmm_env_member_xactor_aa_string(member-name,
                                 vmm_env::do_what_e do_what)
```

OpenVera

Not supported.

Description

Add the specified transactor-type, array of transactors, dynamic array of transactors, scalar-indexed associative array of transactors or string-indexed associative array of transactors data member to the default implementation of the methods specified by the `’do_what’` argument.

The shorthand implementation must be located in a section started by a `“‘vmm_env_member_begin()”` .

Example

Example A-89

```
class my_data_gen extends vmm_xactor;
   . . .
endclass

class my_vmm_env extends vmm_env;
   my_data_gen my_xactor;
   . . .
   `vmm_env_member_begin(my_vmm_env)
      `vmm_env_member_xactor(my_xactor,DO_ALL);
      . . .
   `vmm_env_member_end(my_vmm_env)
   . . .
endclass
```

‘vmm_env_member_subenv*()

Shorthand implementation for a transactor data member.

SystemVerilog

```
‘vmm_env_member_subenv(member-name,
                           vmm_env::do_what_e do_what)

‘vmm_env_member_subenv_array(member-name,
                                 vmm_env::do_what_e do_what)

‘vmm_env_member_subenv_aa_scalar(member-name,
                                 vmm_env::do_what_e do_what)

‘vmm_env_member_subenv_aa_string(member-name,
                                 vmm_env::do_what_e do_what)
```

OpenVera

Not supported.

Description

Add the specified sub-environment-type, array of sub-environments, dynamic array of sub-environments, scalar-indexed associative array of sub-environments or string-indexed associative array of sub-environments data member to the default implementation of the methods specified by the `’do_what’` argument.

The shorthand implementation must be located in a section started by a “‘vmm_env_member_begin()” .

Example

Example A-90

```
class my_subenv extends vmm_subenv
   . . .
endclass

class my_vmm_env extends vmm_env;
   my_subenv  subenv ;
   . . .
   `vmm_env_member_begin(my_vmm_env)
      `vmm_env_member_subenv(sub_env,DO_ALL);
      . . .
   `vmm_env_member_end(my_vmm_env)
endclass
```

‘vmm_env_member_user_defined()

User-defined shorthand implementation data member.

SystemVerilog

```
‘vmm_env_member_user_defined(member-name)
```

OpenVera

Not supported.

Description

Add the specified user-defined default implementation of the methods specified by the `’do_what’` argument.

Refer to “User-defined vmm_env or vmm_subenv Member Default Implementation” on page 2-9 for details on how to specify the shorthand implementation for a data member.

The shorthand implementation must be located in a section started by a `“‘vmm_env_member_begin()”` .

Example

Example A-91

```
class my_vmm_env extends vmm_env;
   bit [7:0] env_id;
   . . .
   `vmm_env_member_begin(my_vmm_env)
      `vmm_env_member_user_defined(env_id);
      . . .
   `vmm_env_member_end(my_vmm_env)
```

```
   function bit do_env_id(vmm_env::do_what_e do_what)
      do_env_id = 1;
      case(do_what)
      . . .
   endfunction
endclass
```

vmm_env::do_what_e

Specifies which methods are to be provided by a shorthand implementation.

SystemVerilog

```
enum {DO_PRINT, DO_START, DO_STOP,
      DO_VOTE, DO_ALL} do_what_e;
```

OpenVera

Not supported.

Description

Used to specify which methods are to include the specified data members in their default implementation. `"DO_PRINT"` includes the member in the default implementation of the `psdisplay()` method. `"DO_START"` includes the member in the default implementation of the start() method, if applicable. `"DO_STOP"` includes the member in the default implementation of the `stop()` method, if applicable. `"DO_VOTE"` automatically registers the member with the `vmm_env::end_vote` consensus instance, if applicable.

Multiple methods can be specified by adding or or'ing the individual symbolic values. All methods are specified by specifying the `"DO_ALL"` symbol.

Example

Example A-92

```
‘vmm_env_member_subenv(tcpip_stack, DO_ALL - DO_STOP);
```

vmm_env::do_psdisplay()

Override the shorthand `psdisplay()` method.

SystemVerilog

```
virtual function string do_psdisplay(string prefix = "")
```

OpenVera

Not supported.

Description

This method overrides the default implementation of the `vmm_env::psdisplay()` method created by the `vmm_env` shorthand macros. If defined, it will be used instead of the default implementation.

Example

Example A-93

```
class my_vmm_env extends vmm_env;
   . . .

  virtual function string do_psdisplay(string prefix = "");
   $sformat(do_psdisplay,"%s Printing environment members",
        prefix);
      . . .
   endfunction
   . . .
endclass
```

vmm_env::do_vote()

Override the shorthand voter registration.

SystemVerilog

```
protected virtual task do_vote()
```

OpenVera

Not supported.

Description

This method overrides the default implementation of the voter registration created by the `vmm_env` shorthand macros. If defined, it will be used instead of the default implementation.

Example

Example A-94

```
class my_vmm_env extends vmm_env;
   . . .
   protected virtual task do_vote();
      //Register with this.end_vote
      . . .
   endtask
   . . .
endclass
```

vmm_env::do_start()

Override the shorthand `start()` method.

SystemVerilog

```
protected virtual task do_start()
```

OpenVera

Not supported.

Description

This method overrides the default implementation of the `vmm_env::start()` method created by the `vmm_env` shorthand macros. If defined, it will be used instead of the default implementation.

Example

Example A-95

```
class my_vmm_env extends vmm_env;
   . . .
   protected virtual task do_start();
      //vmm_env::start() operations
      . . .
   endtask
   . . .
endclass
```

vmm_env::do_stop()

Override the shorthand stop() method.

SystemVerilog

```
protected virtual task do_stop()
```

OpenVera

Not supported.

Description

This method overrides the default implementation of the `vmm_env::stop()` method created by the `vmm_env` shorthand macros. If defined, it will be used instead of the default implementation.

Example

Example A-96

```
class my_vmm_env extends vmm_env;
   . . .
   protected virtual task do_stop();
      //vmm_env::stop() operations
      . . .
   endtask
   . . .
endclass
```

vmm_log

The **vmm_log** class implements an interface to the message service.

Several methods apply to multiple message service interfaces, not just the one where the method is invoked. All message service interfaces that match the specified name and instance name are affected by these methods. If the name or instance name is enclosed between slashes (for example, “/.../”), then they are interpreted as *sed*-style regular expressions. If a value of *“”* is specified, then the name or instance name of the current message service interface is specified. If the **recurse** parameter is TRUE (that is, non-zero), then all interfaces logically under the matching message service interfaces are also specified.

Summary

vmm_log::new()

Create a new instance of a message service interface.

SystemVerilog

```
function new(string name,
     string instance,
     vmm_log under = null);
```

OpenVera

Not supported.

Description

Creates a new instance of a message service interface, with the specified interface name and instance name. Furthermore, a message service interface can optionally be specified as hierarchically below another message service instance to create a logical hierarchy of message service interfaces.

vmm_log::is_above

Specify that this message service instance is hierarchically above the specified message service interface.

SystemVerilog

```
virtual function void is_above(vmm_log log);
```

OpenVera

Not supported.

Description

This method is the corollary of the **`under`** argument of the constructor and need not be used if the specified message service interface has already been constructed as being under this message service interface.

vmm_log::copy()

Copy the configuration of this message service interface to the specified message service interface.

SystemVerilog

```
virtual function vmm_log copy(vmm_log to = null);
```

OpenVera

Not supported.

Description

Copies the configuration of this message service interface to the specified message service interface (or a new interface if none is specified) and returns a reference to the interface copy. The current configuration of the message service interface is copied, except the hierarchical relationship information, which is not modified.

vmm_log::get_name()

Return the message service interface name.

SystemVerilog

```
virtual function string get_name();
```

OpenVera

Not supported.

Description

Returns the name of the message service interface.

vmm_log::get_instance()

Return the message service interface instance name.

SystemVerilog

```
virtual function string get_instance();
```

OpenVera

Not supported.

Description

Returns the instance name of the message service interface.

vmm_log::set_name()

Set the message service interface name.

SystemVerilog

```
virtual function void set_name(string name);
```

OpenVera

Not supported.

Description

Set the name of the message service interface.

vmm_log::set_instance()

Set the message service interface instance name.

SystemVerilog

```
virtual function void set_instance(string inst);
```

OpenVera

Not supported.

Description

Sets the instance name of the message service interface.

vmm_log::kill()

Remove internal references to the message service interface.

SystemVerilog

```
virtual function void kill();
```

OpenVera

Not supported.

Description

Remove any internal referene to this message service interface so it may be reclaimed by the garbage collection once all use references are also removed. Once this method has been invoked, it is no longer possible to control this message service interface by name.

vmm_log::list()

List message service interfaces that match a specified name and instance name.

SystemVerilog

```
virtual function void list(string name = “/./”,
     string instance = “/./”,
     bit recurse = 0);
```

OpenVera

Not supported.

Description

Lists all message service interfaces that match the specified name and instance name. If the **`recurse`** parameter is TRUE (that is, non-zero), then all interfaces logically under the matching message service interface are also listed.

vmm_log::enum(*message-type*)

Enumerated type defining symbolic values for message types.

SystemVerilog

```
enum {FAILURE_TYP,
     NOTE_TYP,
     DEBUG_TYP,
     TIMING_TYP,
     XHANDLING_TYP,
     REPORT_TYP,
     PROTOCOL_TYP,
     TRANSACTION_TYP,
     COMMAND_TYP,
     CYCLE_TYP,
     INTERNAL_TYP,
     DEFAULT_TYP,
     ALL_TYPS};
```

OpenVera

Not supported.

Description

Enumerated type defining symbolic values for message types used when specifying a message type in properties or method arguments. The **`vmm_log::DEFAULT_TYP`** and **`vmm_log::ALL_TYPS`** are special symbolic values usable only with some control methods and are not used to issue actual messages. Multiple message types can be specified to some control methods by combining the value of the required types using the bitwise-or or addition operator.

vmm_log::enum(*message-severity*)

Enumerated type defining symbolic values for message severities.

SystemVerilog

```
enum {FATAL_SEV,
     ERROR_SEV,
     WARNING_SEV,
     NORMAL_SEV,
     TRACE_SEV,
     DEBUG_SEV,
     VERBOSE_SEV,
     DEFAULT_SEV,
     ALL_SEVS};
```

OpenVera

Not supported.

Description

Enumerated type defining symbolic values for message severities used when specifying a message severity in properties or method arguments. The **`vmm_log::DEFAULT_SEV`** and **`vmm_log::ALL_SEVS`** are special symbolic values usable only with some control methods and are not used to issue actual messages. Multiple message severities can be specified to some control methods by combining the value of the required severities using the bitwise-or or addition operator.

vmm_log::enum(*simulation-handling-value*)

Symbolic values for simulation handling.

SystemVerilog

```
enum {IGNORE,
     CONTINUE,
     DUMP_STACK,
     STOP_PROMPT,
     DEBUGGER,
     COUNT_ERROR,
     ABORT_SIM,
     DEFAULT_HANDLING};
```

OpenVera

Not supported.

Description

Enumerated type defining symbolic values for simulation handling used when specifying a new simulation handling when promoting or demoting a message using the **`vmm_log::modify()`** method.

Unless otherwise specified, message types are assigned the default severity and simulation handling shown in Table A-5.

Table A-5 Default Message Severities and Handling

Message Type	Default Severity	Default Handling
FAILURE_TYP	ERROR_SEV	COUNT_ERROR
NOTE_TYP	NORMAL_SEV	CONTINUE
DEBUG_TYP	DEBUG_SEV	CONTINUE
TIMING_TYP XHANDLING_TYP	WARNING_SEV	CONTINUE
TRANSACTION_TYP COMMAND_TYP	TRACE_SEV	CONTINUE

Message Type	Default Severity	Default Handling
REPORT_TYP PROTOCOL_TYP	DEBUG_SEV	CONTINUE
CYCLE_TYP	VERBOSE_SEV	CONTINUE
Any type	FATAL_SEV	ABORT_SIM

vmm_log::vmm_log_format()

Set the message formatter to the specified message formatter instance.

SystemVerilog

```
virtual function vmm_log_format
     set_format(vmm_log_format fmt);
```

OpenVera

Not supported.

Description

Globally sets the message formatter to the specified message formatter instance. A reference to the previously used message formatter instance is returned. A default global message formatter is provided.

vmm_log::set_typ_image()

Replace the image used to display the specified message.

SystemVerilog

```
virtual function string set_typ_image(int typ,
          string image);
```

OpenVera

Not supported.

Description

Globally replaces the image used to display the specified message type with the specified image. The previous image is returned. Default images are provided.

vmm_log::set_sev_image()

Replace the image used to display the specified message severity.

SystemVerilog

```
virtual function string set_sev_image(int sev,
          string image);
```

OpenVera

Not supported.

Description

Globally replaces the image used to display the specified message severity with the specified image. The previous image is returned. Default images are provided.

Example

Example A-97

The following is an example of colorizing the severity display on ANSI terminals.

```
log.set_sev_image(vmm_log::WARNING,
                  "\033[33mWARNING\033[0m");
log.set_sev_image(vmm_log::ERROR_SEV,
                  "\033[31mERROR\033[0m");
log.set_sev_image(vmm_log::FATAL_SEV,
                   "\033[41m*FATAL*\033[0m");
```

vmm_log::start_msg()

Prepare to issue a message.

SystemVerilog

```
virtual function bit start_msg(int typ,
    int severity = DEFAULT_TYP,
    string fname = "/./",
    int     line  = -1);
```

OpenVera

```
virtual function bit (integer type,
    integer severity = DEFAULT_TYP,
    integer msg_id = -1);
```

Description

Prepare to issue a message of the specified type and severity. If the message service interface instance is configured ignore messages of the specified type or severity, the function returns FALSE.

When using SystemVerilog, the current filename and line number where the message is created can be supplied by using the `‘__FILE__` and `‘__LINE__` symbols. For backward compatibility, the `‘VMM_LOG_FORMAT_FILE_LINE` symbol must be defined to enable the inclusion of the filename and line number to the message formatter.

Example

Example A-98

```
program test
  . . .
```

```
   initial begin
      . . .
      env.log.text.start_msg(vmm_log::NOTE_TYP,

vmm_log::DEFAULT_SEV,`__FILE__,~__LINE__);
      env.log.text("Starting Test My_Test");
      env.log.text();
      . . .
   end
```

vmm_log::text()

Add the specified text to the message being constructed.

SystemVerilog

```
virtual function bit text(string msg = "");
```

OpenVera

Not supported.

Description

Adds the specified text to the message being constructed. This method specifies a single line of message text. A newline character is automatically appended when the message is issued. Additional lines of messages can be produced by calling this method multiple times, once per line. If an empty string is specified as message text, all previously specified lines of text are flushed to the output, but the message is not terminated. This method may return FALSE if the message will be filtered out based on the text.

A message must be flushed and terminated by calling the **`vmm_log::end_msg()`** method to trigger the message display and the simulation handling. A message can be flushed multiple times by calling the **`vmm_log::text("")`** method, but the simulation handling and notification will take effect on the message termination.

If additional lines are produced using the **`$display()`** system task or other display mechanisms, they will not be considered by the filters, nor included in explicit log files. They may also be displayed out of order if they are produced before the previous lines of the message are flushed.

For single-line messages, the **‘vmm_fatal()**, **‘vmm_error()**, **‘vmm_warning()**, **‘vmm_note()**, **‘vmm_trace()**, **‘vmm_debug()**, **‘vmm_verbose()**, **‘vmm_report()**, **‘vmm_command()**, **‘vmm_transaction(**), **‘vmm_protocol()** and **‘vmm_cycle()** macros can be used as a shorthand notation.

Table A-6 Message Type and Severity for Shorthand Macros

Macro	Message Type	Message Severity
‘vmm_fatal(vmm_log log, string txt);	Failure	Fatal
‘vmm_error(vmm_log log, string txt);	Failure	Error
‘vmm_warning(vmm_log log, string txt);	Failure	Warning
‘vmm_note(vmm_log log, string txt);	Note	Default
‘vmm_trace(vmm_log log, string txt);	Debug	Trace
‘vmm_debug(vmm_log log, string txt);	Debug	Debug
‘vmm_verbose(vmm_log log, string txt);	Debug	Verbose
‘vmm_report(vmm_log log, string txt);	Report	Default
‘vmm_command(vmm_log log, string txt);	Command	Default
‘vmm_transaction(vmm_log log, string txt);	Transaction	Default
‘vmm_protocol(vmm_log log, string txt);	Protocol	Default
‘vmm_cycle(vmm_log log, string txt);	Cycle	Default

vmm_log::end_msg()

Flush and terminate the current message.

SystemVerilog

```
virtual function void end_msg();
```

OpenVera

Not supported.

Description

Flushes and terminates the current message and triggers the message display and the simulation handling. A message can be flushed multiple times using the **`vmm_log::text("")`** method, but the simulation handling and notification will only take effect on message termination.

vmm_log::enable_types()

Specify the message types to be displayed by the specified message service interfaces

SystemVerilog

```
virtual function void enable_types(int typs,
     string name = "",
     string inst = "",
     bit recursive = 0);
```

OpenVera

Not supported.

Description

Specifies the message types to be displayed by the specified message service interfaces. Message service interfaces are specified by value or regular expression for both the name and instance name. If no name and no instance are explicitly specified, this message service interface is implicitly specified.

If the name or instance named are specified between “/” characters, then the specification is interpreted as a regular expression that must be matched against all known names and instance names, respectively. Both names must match to consider a message service interface as specified. If the **`recursive`** argument is TRUE, all message service interface hierarchically below the specified message service interfaces are included in the specification, whether their name and instance name matches or not. A message service interface must exist to be specified.

The **types** argument specifies the messages types to enable or disable. Types are specified as the bitwise-or or sum of all relevant types.

By default, all message types are enabled.

vmm_log::disable_types()

Specify the message types to be disabled by the specified message service interfaces

SystemVerilog

```
virtual function void disable_types(int typs,
    string name = "",
    string inst = "",
    bit recursive = 0);
```

OpenVera

Not supported.

Description

Specifies the message types to be disabled by the specified message service interfaces. Message service interfaces are specified by value or regular expression for both the name and instance name. If no name and no instance are explicitly specified, this message service interface is implicitly specified.

If the name or instance named are specified between “/” characters, then the specification is interpreted as a regular expression that must be matched against all known names and instance names, respectively. Both names must match to consider a message service interface as specified. If the **`recursive`** argument is TRUE, all message service interface hierarchically below the specified message service interfaces are included in the specification, whether their name and instance name matches or not. A message service interface must exist to be specified.

The **types** argument specifies the messages types to enable or disable. Types are specified as the bitwise-or or sum of all relevant types.

By default, all message types are enabled.

vmm_log::set_verbosity()

Specify the minimum message severity to be displayed.

SystemVerilog

```
virtual function void set_verbosity(int severity,
    string name = "",
    string inst = "",
    bit recursive = 0);
```

OpenVera

Not supported.

Description

Specify the minimum message severity to be displayed when sourced by the specified message service interfaces. See the documentation for the **`enable_types()`** method for the interpretation of the **`name`**, **`inst`** and **`recursive`** arguments and how they are used to specify message service interfaces.

The default minimum severity can be changed by using the **`+vmm_log_default=`***`sev`* runtime command-line option, where *`sev`* is the desired minimum severity and is a one of the following: **`error`**, **`warning`**, **`normal`**, **`trace`**, **`debug`** or **`verbose`**. The default verbosity level can be later modified using this method.

The minimum severity level can be globally forced by using the **`+vmm_force_verbosity=`***`sev`* runtime command-line option. The specified verbosity overrides the verbosity level specified using this method.

vmm_log::get_verbosity()

Return the minimum message severity to be displayed.

SystemVerilog

```
virtual function int get_verbosity();
```

OpenVera

Not supported.

Description

Returns the minimum message severity to be displayed when sourced by this message service interface.

vmm_log::modify()

Modifies the specified type, severity or simulation handling for a message source.

SystemVerilog

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

OpenVera

Not supported.

Description

Modifies the specified message source by any of the specified message service interfaces with the new specified type, severity or simulation handling. The message can be specified by type, severity, numeric ID or by text pattern. By default, messages of any type, severity, ID or text is specified. A message must match all specified criteria.

This method returns a unique message modifier identifier that can be used to remove it using the **`vmm_log::unmodify()`** method. All message modifiers are applied in the same order they were defined before a message is issued.

vmm_log::unmodify()

Removes a message modification from the message service interfaces.

SystemVerilog

```
virtual function void unmodify(int mod_id = -1,
     string name = "",
     string instance = "",
     bit recursive = 0);
```

OpenVera

Not supported.

Description

Removes the specified message modification from the specified message service interfaces. By default, all message modifications are removed.

vmm_log::log_start()

Append messages produced by the specified message service interfaces.

SystemVerilog

```
virtual function void log_start(int file,
     string name = "",
     string instance = "",
     bit recurse = 0)
```

OpenVera

Not supported.

Description

Appends all messages produced by the specified message service interfaces to the specified file. The **`file`** argument must be a file descriptor, as returned by the **`$fopen()`** system task. By default, all message service interfaces append their messages to the standard output. Specifying a new output file does not stop messages from being appended to previously specified files.

vmm_log::log_stop()

Stop logging messages from a specified message service interface.

SystemVerilog

```
virtual function void log_stop(int file,
    string name = "",
    string instance = "",
    bit recurse = 0);
```

OpenVera

Not supported.

Description

Messages issued by the specified message service interfaces are no longer appended to the specified file. The **`file`** argument must be a file descriptor, as returned by the **`$fopen()`** system task. If the specified **`file`** argument is 0, messages are no longer sent to the standard simulation output and transcript. If the **`file`** argument is specified as –1, appending to all files, except the standard output, is stopped.

vmm_log::stop_after_n_errors()

Abort the simulation after a specified number of messages has been issued.

SystemVerilog

```
virtual function void stop_after_n_errors(int n);
```

OpenVera

Not supported.

Description

Aborts the simulation after the specified number of messages with a simulation handling of **`COUNT_ERROR`** has been issued. This value is global and all messages from any message service interface count toward this limit. A zero or negative value specifies no maximum. The default value is 10. The message specified by the **`vmm_log_format::abort_on_error()`** is displayed before the simulation is aborted.

vmm_log::get_message_count()

Return the total number of messages of the specified severities.

SystemVerilog

```
virtual function int
           get_message_count(int severity = ALL_SEVS,
    string name = "",
    string instance = "",
    bit recurse = 0);
```

OpenVera

Not supported.

Description

Returns the total number of messages of the specified severities that have been issued from the specified message service interfaces. Message severities can be specified as a sum of individual message severities to specify more than one severity.

vmm_log::create_watchpoint()

Create a watchpoint descriptor.

SystemVerilog

```
virtual function int
           create_watchpoint(int types = ALL_TYPS,
    int severity = ALL_SEVS,
    string text = "",
    logic issued = 1'bx);
```

OpenVera

Not supported.

Description

Creates a watchpoint descriptor that will be triggered when the specified message is used. The message can be specified by type, severity or by text pattern. By default, messages of all types, severities and text are specified. A message must match all specified criteria to trigger the watchpoint. The **`issued`** parameter specifies if the watchpoint is triggered when the message is physically issued (1'b1), physically not issued, that is, filtered out (1'b0) or regardless if the message is physically issued or not (1'bx).

A watchpoint will be repeatedly triggered every time a message matching the watchpoint specification is issued by a message service interface associated with the watchpoint.

vmm_log::add_watchpoint()

Add the specified watchpoint to the specified message service interfaces.

SystemVerilog

```
virtual function void
           add_watchpoint(int watchpoint_id,
    string name = "",
    string instance = "",
    bit recurse = 0);
```

OpenVera

Not supported.

Description

If a message matching the watchpoint specification is issued by one of the specified message service interfaces associated with the watchpoint, the watchpoint will be triggered.

vmm_log::remove_watchpoint()

Remove the specified watchpoint from the specified message service interfaces.

SystemVerilog

```
virtual function void
           remove_watchpoint(int watchpoint_id,
          string name = "",
     string instance = "",
     bit recurse = 0);
```

OpenVera

Not supported.

Description

If a message matching the watchpoint specification is issued by one of the specified message service interfaces associated with the watchpoint, the watchpoint will be triggered.

vmm_log::wait_for_watchpoint()

Wait for the specified watchpoint to be triggered by a message.

SystemVerilog

```
virtual task wait_for_watchpoint(int watchpoint_id,
     ref vmm_log_msg msg);
```

OpenVera

Not supported.

Description

Waits for the specified watchpoint to be triggered by a message issued by one of the message service interfaces attached to the watchpoint. A descriptor of the message that triggered the watchpoint will be returned.

vmm_log::wait_for_message()

Waits for a one-time watchpoint for a specified message.

SystemVerilog

```
virtual task wait_for_msg(string name = "",
    string instance = "",
    bit recurse = 0,
        int typs = ALL_TYPS,
    int severity = ALL_SEVS,
    string text = "",
    logic issued = 1'bx,
    ref vmm_log_msg msg);
```

OpenVera

Not supported.

Description

Sets up and waits for a one-time watchpoint for the specified message on the specified message service interface. The watchpoint is triggered only once and removed after being triggered.

vmm_log::report()

Report a failure if a message service interface issued an error or fatal message.

SystemVerilog

```
virtual task report(string name = “/./”,
     string instance = “/./”,
     bit recurse = 0);
```

OpenVera

Not supported.

Description

Reports a failure if any of the specified message service interfaces have issued any error or fatal messages. Reports a success otherwise. The text of the pass or fail message is specified using the **`vmm_log_format::pass_or_fail()`** method.

vmm_log::prepend_callback()

Prepend a callback façade instance with the message service.

SystemVerilog

```
virtual function void
           prepend_callback(vmm_log_callbacks cb);
```

OpenVera

Not supported.

Description

Globally prepends the specified callback façade instance with the message service. Callback methods will be invoked in the order in which they were registered.

A warning is issued if the same callback façade instance is registered more than once. Callback façade instances can be unregistered and re-registered dynamically.

Example

Example A-99

```
env.build();
begin
   gen_rx_errs cb = new;
   env.phy.prepend_callback(cb);
end
```

vmm_log::append_callback()

Append a callback façade instance with the message service.

SystemVerilog

```
virtual function void
          append_callback(vmm_log_callbacks cb);
```

OpenVera

Not supported.

Description

Globally appends the specified callback façade instance with the message service. Callback methods will be invoked in the order in which they were registered.

A warning is issued if the same callback façade instance is registered more than once. Callback façade instances can be unregistered and re-registered dynamically.

Example

Example A-100

```
class tb_env extends vmm_env;
   virtual function void build();
      ...
      begin
         sb_mac_cbs cb = new;
         this.mac.append_callback(cb);
      end
      ...
   endfunction: build
```

```
   ...
endclass: tb_env
```

vmm_log::unregister_callback()

Unregister the specified callback façade instance.

SystemVerilog

```
virtual function void unregister_callback(
    vmm_log_callbacks cb);
```

OpenVera

Not supported.

Description

Globally unregisters the specified callback façade instance with the message service. A warning is issued if the specified façade instance is not currently registered with the service. Callback façade instances can later be re-registered.

vmm_log::reset()

Initialize the message service instance iterator.

SystemVerilog

```
function void reset(string name    = "/./",
     string inst    = "/./",
     bit    recurse = 0);
```

OpenVera

```
task reset(string name    = "/./",
     string inst    = "/./",
     bit    recurse = 0);
```

Description

Reset the message service instance iterator for this instance of the message service and initialize it to iterator using the specified name and instance name and optional recursion.

It is then possible to iterate over all known instances of the message service interface that match the specified pattern using the “vmm_log::for_each()” method.

There is one iterator per message service instance.

Example

Example A-101

```
env.log.reset();
for (vmm_log log = env.log.for_each();
     log != null;
     log = env.log.for_each()) begin
end
```

vmm_log::for_each()

Iterate over message service instances.

SystemVerilog

```
function vmm_log for_each();
```

OpenVera

```
function rvm_log for_each();
```

Description

Return a reference to the next known message service interface that matches the iterator specification specified in the last invocation of “vmm_log::reset()” . Returns *NULL* if no more instances match.

There is one iterator per message service instance.

Example

Example A-102

```
env.log.reset();
for (vmm_log log = env.log.for_each();
     log != null;
     log = env.log.for_each()) begin
   ...
end
```

vmm_log::uses_hier_inst_name()

Check if hierarchical instance names are in use.

SystemVerilog

```
function bit uses_hier_inst_name();
```

OpenVera

Not supported.

Description

Returns TRUE if the message service interface instances use hierarchical instance name, as defined by calling “vmm_log::use_hier_inst_name()” . Returns FALSE if the original, flat instance names are in used, as defined by calling “vmm_log::use_orig_inst_name()” .

Example

Example A-103

```
env.build();
if (!env.log.uses_hier_inst_name())
   env.log.use_hier_inst_name();
```

vmm_log::use_hier_inst_name()

Switch to hierarchical instance names.

SystemVerilog

```
function void use_hier_inst_name();
```

OpenVera

Not supported.

Description

Rewrite the instance name of all message service interface instances into a dot-separated hierarchical form. The original instance names can later be restored using “vmm_log::use_orig_inst_name()” .

An instance name is made hierarchical if the message service instance is specified as being under another message service interface. Message service interface hierarchies can be built by specifying the *under* argument to the constructor or by using the `vmm_log::is_above()` method.

For example, the code in Example A-104 will result in instances names `"top"`, `"top.m1"`, `"top.c1"` and `"s1"`. The instance name for `"s1"` is not modified because it is not specified as being under another message service interface and thus creates a new hierarchical root.

Example

Example A-104

```
function tb_env::new();
   super.new("top");
endfunction

function void tb_env::build();
   super.build();
   this.chan = new("Master to slave", "c1");
   this.master = new("m1", this.chan);
   this.slave  = new("s1", this.chan);
   this.log.is_above(this.master.log);
   this.log.is_above(this.chan);
   this.log.use_hier_inst_name();
endfunction
```

vmm_log::use_orig_inst_name()

Switch to original, flat instance names.

SystemVerilog

```
function void use_orig_inst_name();
```

OpenVera

Not supported.

Description

Rewrite the instance name of all message service interface instances into the original, flat form specified when the message service instance was constructed.

Example

Example A-105

```
env.build();
if (env.log.uses_hier_inst_name())
   env.log.use_orig_inst_name();
```

vmm_log::catch()

Add a user-defined message handler

SystemVerilog

```
function int catch(
     vmm_log_catcher catcher,
     string name = "",
     string inst = "",
     bit recurse = 0,
     int typs = ALL_TYPS,
     int severity = ALL_SEVS,
     string text = "");
```

OpenVera

Not supported.

Description

Install the specified message handler to catch any message of the specified type and severity, issued by the specified message service interface instances, which contains the specified text. By default, catches all messages issued by this message service interface instance. A unique message handler identifier is returned that can be used to later uninstall the message handler using vmm_log::uncatch().

Messages will be considered caught by the first user-defined handler found that can handle the message. User-defined message handlers are considered in reverse order of installation i.e. the last handler installed will be considered first. Once caught, messages are handed off to the vmm_log_catcher::caught() method and will not be issued. A user-defined message handler may choose to explicitly issue the

message using the vmm_log_catcher::issue() method or throw the message back to the message service by using the vmm_log_catcher::throw() method to be potentially caught by another suitable message handler or be issued.

Watchpoints are triggered after message catching. If the message has been modified in the catcher, the modified message will trigger applicable watchpoints, if any.

Example

Example A-106

```
class err_catcher extends vmm_log_catcher;
   . . .
endclass

alu_env env;
err_catcher ctcher;

initial begin
  . . .
  ctcher = new(10);
  . . .
  env.build();
  env.sb.log.catch(ctcher,"","", ,vmm_log::ERROR_SEV,
      "/Mismatch/");
end
```

vmm_log::uncatch()

Remove a user-defined message handler.

SystemVerilog

```
function bit uncatch(int catcher_id);
```

OpenVera

Not supported.

Description

Uninstall the specified user-defined message handler. The message handler is identifier by the unique identifier that was returned by the `vmm_log::catch()` method when it was originally installed.

Returns TRUE if the specified message handler was successfully uninstalled. Returns FALSE otherwise.

Example

Example A-107

```
class err_catcher extends vmm_log_catcher;
  . . .
endclass

alu_env env;
err_catcher ctcher;

initial begin
  . . .
  env.build();
  ctcher_id = env.sb.log.catch(ctcher, , , ,
```

```
      vmm_log::ERROR_SEV,"/Mismatch/");
  . . .
  env.sb.log.uncatch(ctcher_id);
end
```

vmm_log::uncatch_all()

Remove all user-defined message handlers.

SystemVerilog

```
function void uncatch_all();
```

OpenVera

Not supported.

Description

Uninstall all user-defined message handlers. All message handlers, even those that were registered with or through a different message service interface are uninstalled.

Example

Example A-108

```
class err_catcher extends vmm_log_catcher;
  . . .
endclass

alu_env env;
err_catcher ctcher1, ctcher2;

initial begin
  . . .
  env.build();
  ctcher_id1 = env.log.catch(ctcher1, , , ,
     vmm_log::ERROR_SEV,"/MON_ERROR_008/");
  ctcher_id2 = env.log.catch(ctcher2, , , ,
      vmm_log::ERROR_SEV,"/MON_ERROR_010/");
  . . .
```

```
  if(env.mon.error_cnt >10)
      env.log.uncatch_all();
end
```

vmm_log_msg

This class describes a message issued by a message service interface that caused a watchpoint to be triggered. It is returned by the **`vmm_log::wait_for_watchpoint()`** and **`vmm_log::wait_for_msg()`** method.

Summary

vmm_log_message::log

A reference to the message reporting interface that has issued the message.

SystemVerilog

```
vmm_log log;
```

OpenVera

Not supported.

vmm_log_message::timestamp

The simulation time when the message was issued.

SystemVerilog

```
time timestamp;
```

OpenVera

Not supported.

vmm_log_message::original_typ

Original message type as specified in the code creating the message.

SystemVerilog

```
int original_typ;
```

OpenVera

Not supported.

vmm_log_message::original_severity

Original message severity as specified in the code creating the message.

SystemVerilog

```
int original_severity;
```

OpenVera

Not supported.

vmm_log_message::effective_typ

Effective message type as potentially modified by the **`vmm_log::modify()`** method.

SystemVerilog

```
int effective_typ;
```

OpenVera

Not supported.

vmm_log_message::effective_severity

Effective message severity as potentially modified by the **`vmm_log::modify()`** method.

SystemVerilog

```
int effective_severity;
```

OpenVera

Not supported.

vmm_log_message::text[]

Formatted text of the message.

SystemVerilog

```
string text[$];
```

OpenVera

Not supported.

Description

Each element of the array contains one line of text as built by individual calls to the **`vmm_log::text()`** method.

vmm_log_message::issued

Indicates if the message has been physically issued or not.

SystemVerilog

```
bit issued;
```

OpenVera

Not supported.

Description

If non-zero, then the message has been issued.

vmm_log_message::handling

The simulation handling after the message is physically issued.

SystemVerilog

```
int handling;
```

OpenVera

Not supported.

vmm_log_callbacks

This class provides a facade for the callback methods provided by the messge service. Callbacks are associated with the message service itself, not a particular message service interface instance.

Summary

vmm_log_callback::pre_finish()

Simulation termination callback

SystemVerilog

```
virtual function void pre_finish(vmm_log log,
                                 ref bit finished);
```

OpenVera

Not supported.

Description

This callback method is invoked by the message service after the “vmm_log_callback::pre_abort()” callback method and immediately before $finish() is invoked to terminate the simulation.

The value of the "finished" parameter is 0 by default. If its value is ultimately returned as 1 by the sequence of callback methods, it indicates that the callback method has taken the responsibility of terminating the simulation. The final report and $finish() will therefore not be called.

Use this callback method if you wish to delay the termination of the simulation after an abort condition has been detected.

Example

Example A-109 Terminating the simulation of 100 time units

```
virtual function void pre_finish(vmm_log log,
                                 ref bit finished);
   fork
      begin
```

```
            #100;
            log.report();
            $finish();
         end
      join_none
      finished = 1;
   endfunction
```

vmm_log_callback::pre_abort()

Abort condition callback

SystemVerilog

```
virtual function void pre_abort(vmm_log log);
```

OpenVera

```
virtual function pre_abort(rvm_log log);
```

Description

This callback method is invoked by the message service when a message was issued with an ABORT simulation handling or the maximum number of message with a COUNT_ERROR handling have been issued. This callback method is invoked before the “vmm_log_callback::pre_finish()” callback method.

The message service interface provided as an argument may be used to issue further messages.

vmm_log_callback::pre_stop()

Stop condition callback

SystemVerilog

```
virtual function void pre_stop(vmm_log log);
```

OpenVera

```
virtual function pre_stop(rvm_log log);
```

Description

This callback method is invoked by the message service when a message was issued with a STOP simulation handling.

The message service interface provided as an argument may be used to issue further messages.

vmm_log_callback::pre_debug()

Debug condition callback

SystemVerilog

```
virtual function void pre_debug(vmm_log log);
```

OpenVera

```
virtual function pre_debug(rvm_log log);
```

Description

This callback method is invoked by the message service when a message was issued with a DEBUGGER simulation handling.

The message service interface provided as an argument may be used to issue further messages.

vmm_log_catcher

VMM provides a mechanism to execute user-specific code, if a certain message is issued from the testbench environment, using the **vmm_log_catcher** class.

The **vmm_log_catcher** class is based on regexp to specify matching **vmm_log** messages.

If a message with the specified regexp is issued during simulation, the user-specified code is executed.

The **vmm_log_catcher::caught()** method can be used to modify the caught message, changing its type and severity. You can choose to ignore the message, in which case it will not be displayed. The message can be displayed as is after executing user-specified code. The updated message can be displayed by calling **vmm_log_catcher::issue()** in the caught method.

The caught message, modified or unmodified, can be passed to other catchers that have been registered, using the **vmm_log_catcher::throw** function.

The messages to be caught are registered with the **vmm_log** class using the **vmm_log::catch()** method.

```
class error_catcher extends vmm_log_catcher;

virtual function void caught(vmm_log_msg msg);
   msg.text[0] = {" Acceptable Error" , msg.text[0]};
   msg.effective_severity = vmm_log::WARNING_SEV;
   issue(msg);
endfunction
endclass
```

Registration should be done in the program block.

```
initial begin
   env = new();
   error_catcher catcher = new();
   env.build();
   catcher_id =
      env.sb.log.catch(catcher,,,1,,vmm_log::ERROR_SEV,"/
      Mismatch/");
   env.run();
end
```

The **`error_catcher`** class extends the **`vmm_log_catcher`** class and implements the **`caught()`** method. The **`caught()`** method prepends " Acceptable Error" to the original message and changes the severity to WARNING_SEV.

In the initial block of the program block, an object of **`error_catcher`** is created and a handle passed to the **`catch()`** method to register the catcher. Any **`vmm_log`** message from scoreboard (sb), having ERROR_SEV severity and including the string "Mismatch" will be caught and changed to WARNING_SEV with "Acceptable Error" prepended to it.

If the message is to be caught from all **`vmm_log`** instances, the **`catch()`** method can be called as:

```
env.sb.log.catch(catcher,"/./","/./
",1,vmm_log::ERROR_SEV,"/Mismatch/");
```

To unregister a catcher **`vmm_log::uncatch(`***`catcher-id`***`)`** or **`vmm_log::uncatch_all()`** methods can be used.

Summary

vmm_log_catcher::caught()

Handle a caught message.

SystemVerilog

```
virtual function void caught(vmm_log_msg msg);
```

OpenVera

Not supported.

Description

This method specifies how to handle a caught message. Unless re-issued using the `vmm_log_catcher::issue()` method or thrown back to the message service using the `vmm_log_catcher::throw()` method, this message will not be issued.

How a message is handled once caught is up to the user. Whatever behavior is specified in the extension of this method defines how a message is handled. If left empty, the message will be ignored.

This method must be overloaded.

Example

Example A-110

```
virtual function void caught(vmm_log_msg msg);
   if (num_errors < max_errors) begin
      msg.text[0] = {"ACCEPTABLE ERROR: ", msg.text[0]};
      msg.effective_severity = vmm_log::WARNING_SEV;
      . . .
```

```
   end
  else
       . .
endfunction
```

vmm_log_catcher::issue()

Issue a caught message.

SystemVerilog

```
protected function void issue(vmm_log_msg msg);
```

OpenVera

Not supported.

Description

Immediately issue the specified message. The message is not subject to being caught any further by this or another user-defined message handler.

The message described by the `vmm_log_msg` descriptor may be modified before being issued.

Example

Example A-111

```
virtual function void caught(vmm_log_msg msg);
   if (num_errors > max_errors) begin
      issue(msg);
   end
   . . .
endfunction
```

vmm_log_catcher::throw()

Throw back a caught message.

SystemVerilog

```
protected function void throw(vmm_log_msg msg);
```

OpenVera

Not supported.

Description

Throw the specified message back to the message service. The message is will be subject to being caught another user-defined message handler but not by this one.

The message described by the `vmm_log_msg` descriptor may be modified before being thrown back.

Example

Example A-112

```
virtual function void caught(vmm_log_msg msg);
   if (num_errors < max_errors)
   throw(msg);
endfunction
```

vmm_log_format

This class is used to specify how messages are formatted before being displayed or logged to files. The default implementation of these methods produces the default message format.

Summary

vmm_log_format::format_msg()

Format a message.

SystemVerilog

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

OpenVera

```
virtual function string format_msg(string name,
     string instance,
     string msg_typ,
     string severity,
     string lines[$]);
```

Description

Return a fully formatted image of the message as specified by the arguments. The **`lines`** parameter contains one line of message text for each non-empty call to the **`vmm_log::text()`** method. A line may contain newline characters.

This method is called by all message service interfaces to format a message on the first occurrence of a call to the **`vmm_log::end_msg()`** method or empty **`vmm_log::text()`** method call after a call to **`vmm_log::start_msg()`**. Subsequent calls to these methods call the **`vmm_log_format::continue_msg()`** method.

For backward compatibility when using SystemVerilog, the **‘VMM_LOG_FORMAT_FILE_LINE** symbol must be defined to enable the inclusion of the filename and line number to the message formatter.

Example

Example A-113

```
class env_log_fmt extends vmm_log_format;
   function string format_msg(string name = "", string
instance = "",
                           string msg_type,  string severity,
                                ref string lines[$]);
   for(int i=0;i<lines.size;i++)
      $sformat(format_msg,
       "%0t, (%s) (%s) [%0s:%0s] \n \t \t %s ",
       $time, name, instance, msg_type, severity, lines[i]);
   endfunction
endclass

class my_env extends vmm_env;
   . . .
   env_log_fmt  env_fmt = new();
   function new();
      this.log.set_format(env_fmt);
      `vmm_note(log,"Inside New");
   endfunction
endclass
```

vmm_log_format::continue_msg()

Format the continuation of a message.

SystemVerilog

```
virtual function string continue_msg(
    string name,
    string instance,
    string msg_typ,
    string severity,
    ref string lines[$]);
```

OpenVera

```
virtual function string continue_msg(string name,
     string instance,
     string msg_typ,
     string severity,
     string lines[$]);
```

Description

This method is called by all message service interfaces to format the continuation of a message n subsequent calls to the **`vmm_log::end_msg()`** method or empty **`vmm_log::text("")`** method call. The first call to the **`vmm_log::end_msg()`** method or empty **`vmm_log::text("")`** method uses the **`vmm_log_format::format_msg()`** method.

a message on subsequent occurrences of a call to the `"vmm_log::end_msg()"` method or empty `"vmm_log::text()"` method call after a call to `"vmm_log::start_msg()"`. The first call to these methods call the `"vmm_log_format::format_msg()"` method.

For backward compatibility when using SystemVerilog, the `VMM_LOG_FORMAT_FILE_LINE symbol must be defined to enable the inclusion of the filename and line number to the message formatter.

Example

Example A-114

```
. . .
string line[$];
string str;
super.build();
str = "Continue Msg string";

for(int idx = 0; idx < 5 ; idx++)
  line.push_back(str);
`vmm_note(log,$psprintf("%0s",this.format.continue_msg
             ("msg","log","","DEBUG_SEV",line)));
. . .
```

vmm_log_format::abort_on_error()

Called when the total number of **`COUNT_ERROR`** messages exceeds the error message threshold.

SystemVerilog

```
virtual function string abort_on_error(int count,
    int limit);
```

OpenVera

Not supported.

Description

The string returned by the method describes the cause of the simulation aborting. If *`null`* is returned, no explanation is displayed.

This method is called and the returned string is displayed before the **`vmm_log_callbacks::pre_abort()`** callback methods are invoked.

vmm_log_format::pass_or_fail()

Format the final pass/fail message at the end of simulation.

SystemVerilog

```
virtual function string pass_or_fail(bit pass,
    string name,
    string instance,
    int fatals,
    int errors,
    int warnings,
    int dem_errs,
    int dem_warns);
```

OpenVera

Not supported.

Description

This method is called by the **`vmm_log::report()`** method to format the final pass/fail message at the end of simulation.

The **`pass`** argument, if true, indicates that the simulation was successful.

The **`name`** and **`instance`** arguments are the specified name and instance names specified to the **`vmm_log::report()`** method.

The **`fatals`** argument is the total number of **`vmm_log::FATAL_SEV`** messages that were issued.

The **`errors`** argument is the total number of **`vmm_log::ERROR_SEV`** messages that were issued.

The **warnings** argument is the total number of **vmm_log::WARNING_SEV** messages that were issued.

The **dem_errs** argument is the total number of **vmm_log::ERROR_SEV** messages that were demoted.

The **dem_warns** argument is the total number of **vmm_log::WARNING_SEV** messages that were demoted.

vmm_ms_scenario

Base class for all user-defined multi-stream scenario descriptors. This class extends from vmm_scenario.

Summary

vmm_ms_scenario::new()

Instantiate a multi-stream scenario descriptor.

SystemVerilog

```
function new(vmm_scenario parent = null)
```

OpenVera

Not supported.

Description

Create a new instance of a multi-stream scenario descriptor.

If a parent scenario descriptor is specified, this instance of a multi-stream scenario descriptor is assumed to be instantiated inside the specified scenario descriptor, creating a hierarchical multi-stream scenario descriptor.

If no parent scenario descriptor is specified, it is assumed to be a top-level scenario descriptor.

Example

Example A-115

```
class my_scenario extends vmm_ms_scenario;
   . . .
   function new;
      super.new(null);
      . . .
   endfunction: new
   . . .
endclass
```

```
program test;
   . . .
   my_scenario sc0 = new;
   . . .
endprogram
```

vmm_ms_scenario::execute()

Execute a multi-stream scenario.

SystemVerilog

```
virtual task execute(ref int n)
```

OpenVera

Not supported.

Description

Execute the scenario. Increments the argument "*n*" by the total number of transactions that were executed in this scenario.

This method must be overloaded to procedurally define a multi-stream scenario.

Example

Example A-116

```
class my_scenario extends vmm_ms_scenario;
   my_atm_cell_scenario atm_scenario;
   my_cpu_scenario cpu_scenario;
   . . .
   function new;
      super.new(null);
      atm_scenario = new;
      cpu_scenario = new;
   endfunction: new

   task execute(ref int n);
      fork
        begin
```

```
            atm_cell_channel out_chan;
            int unsigned nn = 0;
            $cast(out_chan, this.get_channel(
                 "ATM_SCENARIO_CHANNEL"));
            atm_scenario.randomize with {length == 1;};
            atm_scenario.apply(out_chan, nn);
            n += nn;
          end
          begin
            cpu_channel out_chan;
            int unsigned nn = 0;
            $cast(out_chan,this.get_channel(
                "CPU_SCENARIO_CHANNEL"));
            cpu_scenario.randomize with {length == 1;};
            cpu_scenario.apply(out_chan, nn);
            n += nn;
          end
        join
    endtask: execute
    . . .
endclass: my_scenario
```

vmm_ms_scenario::get_context_gen()

Returns the multi-stream scenario generator executing this scenario.

SystemVerilog

```
function vmm_ms_scenario_gen get_context_gen()
```

OpenVera

Not supported.

Description

Returns a reference to the multi-stream scenario generator that is providing the context for the execution of this multi-stream scenario descriptor. Returns `NULL` if this multi-stream scenario descriptor has not been registered with a multi-stream scenario generator.

vmm_ms_scenario::get_ms_scenario()

Returns a registered multi-stream scenario descriptor.

SystemVerilog

```
function vmm_ms_scenario get_ms_scenario(string scenario,
    string gen = "")
```

OpenVera

Not supported.

Description

Returns a copy of the multi-stream scenario registered under the specified scenario name in the multi-stream generator registered under the specified generator name. Returns `NULL` of no such scenario exists.

If no generator name is specified, look into the scenario registry of the generator executing this scenario.

The scenario can then be executed within the context of the generator where it is registered by calling its vmm_ms_scenario::execute() method.

Example

Example A-117

```
`vmm_scenario_gen(atm_cell, "atm trans")

program test_ms_scenario;
  vmm_ms_scenario_gen atm_ms_gen = new("Atm Scenario Gen",
12);
```

```
   vmm_ms_scenario first_ms_scen = new;
   vmm_ms_scenario buffer_ms_scen = new;
   . . .
   initial begin
      atm_ms_gen.register_ms_scenario("FIRST
SCEN",first_ms_scen);
      . . .
      buffer_ms_scen = atm_ms_gen.get_ms_scenario("FIRST
SCEN");
      if(buffer_ms_scen != null)
         vmm_log(log,"Returned scenario \n");
         . . .
      else
         vmm_log(log,"Returned null, scenario doesn't
exists\n");
      . . .
   end

endprogram
```

vmm_ms_scenario::get_channel()

Returns a registered output channel.

SystemVerilog

```
function vmm_channel get_channel(string name)
```

OpenVera

Not supported.

Description

Returns the output channel registered under the specified logical name in the multi-stream generator where the multi-stream scenario generator is registered. Returns `NULL` of no such channel exists.

Example

Example A-118

```
`vmm_channel(atm_cell)
`vmm_scenario_gen(atm_cell, "atm trans")

program test_ms_scenario;
  vmm_ms_scenario_gen atm_ms_gen = new("Atm Scenario Gen",
12);
   atm_cell_channel my_chan=new("MY_CHANNEL", "EXAMPLE");
   atm_cell_channel buffer_channel = new("MY_BUFFER",
"EXAMPLE");
   . . .
   initial begin
      . . .
    buffer_channel = atm_ms_gen.get_channel("MY_CHANNEL");
     if(buffer_channel != null)
        vmm_log(log,"Returned channel \n");
```

```
            . . .
         else
            vmm_log(log,"Returned null value\n");
         . . .
      end

   endprogram
```

vmm_ms_scenario_gen

This class is a pre-defined multi-stream scenario generator.

VMM provides this class to model general purpose scenarios. It is now possible to generate heterogeneous scenarios and have them controlled by a unique transactor.

The multi-stream scenario generation mechanism provides an efficient way of generating and synchronizing stimulus to various BFMs. This helps user in reusing block level scenarios in subsystem and system levels and controlling/synchronizing the execution of those scenarios of same/different streams. Single stream scenarios can also be reused in multi-stream scenarios. **`vmm_ms_scenario`** and **`vmm_ms_scenario_gen`** are the base classes provided by VMM for this functionality. This section describes the various kinds of usage of multi-stream scenario generation with these base classes.

Generated scenarios can be transferred to any number of channels of various types anytime during simulation, making this solution very scalable, dynamic and completely controllable by the user. Furthermore, it is possible to model sub-scenarios that can be attached and controlled by an overall scenario in a hierarchical way. User can determine the number of scenarios or the number of transactions to be generated, either on a MSS basis or on a given scenario generator, making this use model scalable from block to system level.

It is also possible to add/remove scenarios as simulation advances, facilitating detection of corner cases or address other constraints on the fly. In case multiple scenario generators should access a common channel, it is possible to give the channel access to only

one generator on a given time slot. In this case, other generators do wait until the channel is released, thereby making it a blocking transaction.

The following methods are available. See“Multi-Stream Scenarios” on page 4-11 for guidelines on how the multi-stream scenario generator can be used and how multi-stream scenarios—including hierarchical scenarios—are defined and executed.

Summary

vmm_ms_scenario_gen::stop_after_n_scenarios

Number of multi-stream scenarios to generate.

SystemVerilog

```
int unsigned stop_after_n_scenarios
```

OpenVera

Not supported.

Description

Automatically stop the multi-stream scenario generator when the number of generated multi-streams scenarios reaches or surpasses the specified value. A value of zero specifies an infinite number of multi-stream scenarios.

Only the multi-stream scenarios explicitly executed by this instance of the multi-stream scenario generator are counted. Sub-scenarios executed as part of a higher-level multi-stream scenario are not counted.

Example

Example A-119

```
`vmm_scenario_gen(atm_cell, "atm trans")

class my_ms_scenario extends vmm_ms_scenario;
   . . .
endclass
program test_ms_scenario;
   . . .
   vmm_ms_scenario_gen ms_gen = new("MS Scenario Gen", 10);
```

```
   my_ms_scenario ms_scen = new;
   . . .
   initial begin
      . . .
      ms_gen.stop_after_n_scenarios = 10;
      . . .
   end

endprogram
```

vmm_ms_scenario_gen::stop_after_n_insts

Number of transaction descriptor to generate.

SystemVerilog

```
int unsigned stop_after_n_insts
```

OpenVera

Not supported.

Description

Automatically stop the multi-stream scenario generator when the number of generated transaction descriptors reaches or surpasses the specified value. A value of zero indicates an infinite number of transaction descriptors.

The number of transaction descriptor instances generated by the execution of a multi-stream scenario is the number of transactions reported by the `vmm_ms_scenario::execute()` method when it returns. Entire scenarios are executed before the generator is stopped so the actual number of transaction descriptors generated may be greater than the specified value.

Example

Example A-120

```
`vmm_scenario_gen(atm_cell, "atm trans")

class my_ms_scenario extends vmm_ms_scenario;
   . . .
endclass
```

```
program test_ms_scenario;
   . . .
   vmm_ms_scenario_gen ms_gen = new("MS Scenario Gen", 10);
   my_ms_scenario ms_scen = new;
   . . .
   initial begin
      . . .
      ms_gen.stop_after_n_instances = 100;
      . . .
   end

endprogram
```

vmm_ms_scenario_gen::scenario_count

Number of multi-stream scenarios generated so far.

SystemVerilog

```
protected int scenario_count;
```

OpenVera

Not supported.

Description

Current count of the number of top-level multi-stream scenarios generated the multi-stream scenario generator. When is reaches or surpasses the value in `vmm_ms_scenario_gen::stop_after_n_scenarios`, the generator stops.

Only the multi-stream scenarios explicitly executed by this instance of the multi-stream scenario generator are counted. Sub-scenarios executed as part of a higher-level multi-stream scenario are not counted.

Example

Example A-121

```
class my_ms_scen extends vmm_ms_scenario_gen;
     . . .
     function void print_ms_gen_fields();
          . . .
          `vmm_note(log,$psprintf("Present scenario count is
%
```

```
d\n",this.scenario_count));
    endfunction
    . . .
endclass

program test_scen;
    . . .
    my_ms_scen my_gen= new("MY MS SCENARIO",10);
    . . .
    initial begin
        fork
         begin
        @event;
           my_gen.print_ms_gen_fields();
         end
         ..
        join
    . . .
    end
end
```

vmm_ms_scenario_gen::inst_count

Number of transaction descriptor generated so far.

SystemVerilog

```
protected int inst_count;
```

OpenVera

Not supported.

Description

Current count of the number of individual transaction descriptor instances generated by the multi-stream scenario generator. When is reaches or surpasses the value in `vmm_ms_scenario_gen::stop_after_n_insts`, the generator stops.

The number of transaction descriptor instances generated by the execution of a multi-stream scenario is the number of transactions reported by the `vmm_ms_scenario::execute()` method when it returns.

Example

Example A-122

```
class my_ms_scen extends vmm_ms_scenario_gen;
     . . .
     function void print_ms_gen_fields();
          . . .
          `vmm_note(log,$psprintf("Present instance count is
%
```

```
d\n",this.inst_count));
    endfunction
    . . .
endclass

program test_scen;
    . . .
    my_ms_scen my_gen= new("MY MS SCENARIO",10);
    . . .
    initial begin
    . . .
    my_gen.print_ms_gen_fields();
    . . .
    end
end
```

vmm_ms_scenario_gen::get_n_scenarios()

Number of multi-stream scenarios generated so far.

SystemVerilog

```
function int unsigned get_n_scenarios()
```

OpenVera

Not supported.

Description

Return the current value of the `vmm_ms_scenario_gen::scenario_count` property.

Example

Example A-123

```
class my_ms_scen extends vmm_ms_scenario_gen;
     . . .
     function void print_ms_gen_fields();
          . . .
          `vmm_note(log,$psprintf("Present scenario count is
%


d\n",this.get_n_scenarios()));
     endfunction
     . . .
endclass

program test_scen;
     . . .
     my_ms_scen my_gen= new("MY MS SCENARIO",10);
```

```
    . . .
    initial begin
    . . .
    my_gen.print_ms_gen_fields();
    . . .
    end
end
```

vmm_ms_scenario_gen::get_n_insts()

Number of transaction descriptors generated so far.

SystemVerilog

```
function int unsigned get_n_insts()
```

OpenVera

Not supported.

Description

Return the current value of the `vmm_ms_scenario_gen::inst_count` property.

Example

Example A-124

```
class my_ms_scen extends vmm_ms_scenario_gen;
     . . .
     function void print_ms_gen_fields();
          . . .
          `vmm_note(log,$psprintf("Present instance count is
%


d\n",this.get_n_scenarios()));
     endfunction
     . . .
endclass

program test_scen;
     . . .
     my_ms_scen my_gen= new("MY MS SCENARIO",10);
```

```
    . . .
    initial begin
    . . .
    my_gen.print_ms_gen_fields();
    . . .
    end
end
```

vmm_ms_scenario_gen::GENERATED

Notification of a newly generated scenario.

SystemVerilog

```
typedef enum int {GENERATED} symbols_e
```

OpenVera

Not supported.

Description

Notification in `vmm_xactor::notify` that is indicated every time a new multi-stream scenario is generated and about to be executed.

Example

Example A-125

```
program test_scen;
    . . .
    vmm_ms_scenario_gen my_ms_gen= new("MY MS
SCENARIO",10);
    . . .
    initial begin
    . . .
    `vmm_note(log,"Waiting for notification : GENERATED
\n");

my_ms_gen.notify.wait_for(vmm_ms_scenario_gen::GENERATED);
    . . .
    end
end
```

vmm_ms_scenario_gen::DONE

Notification of a generation completed.

SystemVerilog

```
typedef enum int {DONE} symbols_e
```

OpenVera

Not supported.

Description

Notification in `vmm_xactor::notify` that is indicated when the generation process has completed as specified by the `vmm_ms_scenario_gen::stop_after_n_scenarios` and `vmm_ms_scenario_gen::stop_after_n_insts` class properties.

Example

Example A-126

```
program test_scen;
     . . .
     vmm_ms_scenario_gen my_ms_gen= new("MY MS
SCENARIO",10);
     . . .
     initial begin
     . . .
     `vmm_note(log,"Waiting for notification : DONE \n");
     my_ms_gen.notify.wait_for(vmm_ms_scenario_gen::DONE);
     . . .
     end
end
```

vmm_ms_scenario_gen::register_ms_scenario()

Register a multi-stream, scenario descriptor

SystemVerilog

```
virtual function void register_ms_scenario(string name,
    vmm_ms_scenario scenario)
```

OpenVera

Not supported.

Description

Registers the specified multi-stream scenario under the specified name. The same scenario may be registered multiple times under different names, thus creating an alias to the same scenario.

Registering a scenario implicitly appends it to the scenario set if it is not already in the `vmm_ms_scenario_gen::scenario_set[$]` array.

It is an error to attempt to register a scenario under a name that already exists. Use `vmm_ms_scenario_gen::replace_ms_scenario()` to replace a registered scenario.

Example

Example A-127

```
class my_ms_scen extends vmm_ms_scenario;
    . . .
endclass
```

```
program test_scenario;
  vmm_ms_scenario_gen my_ms_gen = new("MS Scenario Gen", 9);
   my_ms_scen ms_scen = new;
     . . .
   initial begin
      . . .
      vmm_log(log,"Registering MS scenario \n");
      my_ms_gen.register_ms_scenario("MS SCEN-1",ms_scen);
      . . .
    end
endprogram
```

vmm_ms_scenario_gen::ms_scenario_exists()

Checks if a scenario is registered under a specified name

SystemVerilog

```
virtual function bit ms_scenario_exists(string name)
```

OpenVera

Not supported.

Description

Returns TRUE if there is a multi-stream scenario registered under the specified name. Returns FALSE otherwise.

Use `vmm_ms_scenario_gen::get_ms_scenario()` to retrieve a scenario under a specified name.

Example

Example A-128

```
class my_ms_scen extends vmm_ms_scenario;
    . . .
endclass

program test_scenario;
  vmm_ms_scenario_gen my_ms_gen = new("MS Scenario Gen", 9);
   my_ms_scen ms_scen = new;
     . . .
   initial begin
      . . .
      vmm_log(log,"Registering MS scenario \n");
      my_ms_gen.register_ms_scenario("MS SCEN-1",ms_scen);
      . . .
```

```
         if(my_ms_gen.ms_scenario_exists("MS-SCEN-1"))
            `vmm_note(log, "Scenario MS-SCEN-1 is
registered");
         else
              `vmm_note(log, "Scenario MS-SCEN-1 is not yet


registered");
         . . .
    end
endprogram
```

vmm_ms_scenario_gen::get_ms_scenario()

Returns the scenario registered under a specified name

SystemVerilog

```
virtual function vmm_ms_scenario get_ms_scenario(
    string name)
```

OpenVera

Not supported.

Description

Returns a copy of the multi-stream scenario descriptor registered under the specified name. Issues a warning message and returns `NULL` if there are no scenarios registered under that name.

Example

Example A-129

```
class my_ms_scen extends vmm_ms_scenario;
    . . .
endclass

program test_scenario;
  vmm_ms_scenario_gen my_ms_gen = new("MS Scenario Gen", 9);
   my_ms_scen ms_scen = new;
     my_ms_scen buffer_scen = new;
     . . .
   initial begin
      . . .
      vmm_log(log,"Registering MS scenario \n");
      my_ms_gen.register_ms_scenario("MS-SCEN-1",ms_scen);
      . . .
```

```
        buffer_scen =
my_ms_gen.get_ms_scenario("MY-SCEN_1");
        . . .
    end

endprogram
```

vmm_ms_scenario_gen::get_ms_scenario_name()

Returns a name under which a scenario is registered

SystemVerilog

```
virtual function string get_names_by_ms_scenario(
    vmm_ms_scenario scenario)
```

OpenVera

Not supported.

Description

Returns a name under which the specified multi-stream scenario descriptor is registered. Returns "" if the scenario is not registered.

Example

Example A-130

```
class my_ms_scen extends vmm_ms_scenario;
    . . .
endclass

program test_scenario;
  vmm_ms_scenario_gen my_ms_gen = new("MS Scenario Gen", 9);
   my_ms_scen ms_scen = new;
    string buffer_name;

   initial begin
      . . .
      vmm_log(log,"Registering MS scenario \n");
      my_ms_gen.register_ms_scenario("MS-SCEN-1",ms_scen);
      . . .
         buffer_name =
```

```
my_ms_gen.get_ms_scenario_name(ms_scen);
    vmm_note(log,`vmm_sformatf("Registered name for ms_scen
is : %


s\n",buffer_name));
          . . .
     end

endprogram
```

vmm_ms_scenario_gen::get_ms_scenario_index()

Returns the index of the specified scenario

SystemVerilog

```
virtual function int get_ms_scenario_index(
    vmm_ms_scenario scenario)
```

OpenVera

Not supported.

Description

Returns the index of the specified scenario descriptor in the `vmm_ms_scenario_gen::scenario_set[$]` array. A warning message is issued and returns -1 if the scenario descriptor is not found in the scenario set.

Example

Example A-131

```
class my_ms_scen extends vmm_ms_scenario;
    . . .
endclass

program test_scenario;
  vmm_ms_scenario_gen my_ms_gen = new("MS Scenario Gen", 9);
   my_ms_scen ms_scen = new;
     int buffer_index;

   initial begin
      . . .
      vmm_log(log,"Registering MS scenario \n");
      my_ms_gen.register_ms_scenario("MS-SCEN-1",ms_scen);
```

```
      . . .
         buffer_index =
my_ms_gen.get_ms_scenario_index(ms_scen);
      vmm_note(log,`vmm_sformatf("Index for ms_scen is :
%d\n",buffer_index));
         . . .
     end

endprogram
```

vmm_ms_scenario_gen::get_names_by_ms_scenario()

Returns the names under which a scenario is registered

SystemVerilog

```
virtual function void get_names_by_ms_scenario(
     vmm_ms_scenario scenario,
     ref string       name[$])
```

OpenVera

Not supported.

Description

Appends the names under which the specified multi-stream scenario descriptor is registered. Returns the number of names that were added to the array.

Example

Example A-132

```
class my_ms_scen extends vmm_ms_scenario;
   . . .
endclass

program test_scenario;
   string scen_name_arr[$];
  vmm_ms_scenario_gen my_ms_gen = new("MS Scenario Gen", 9);
   my_ms_scen ms_scen = new;
   . . .
   initial begin
      . . .
      `vmm_note(log,"Registering MS scenario \n");
      my_ms_gen.register_ms_scenario("MS-SCEN-1",ms_scen);
```

```
      my_ms_gen.register_ms_scenario("MS-SCEN-2",ms_scen);
      . . .
my_ms_gen.get_names_by_ms_scenario(ms_scen,scen_name_arr);
      . . .
   end

endprogram
```

vmm_ms_scenario_gen::get_all_ms_scenario_names()

Returns all the names in the scenario registry

SystemVerilog

```
virtual function void get_all_ms_scenario_names(
    ref string    name[$])
```

OpenVera

Not supported.

Description

Appends the names under which a multi-stream scenario descriptor is registered. Returns the number of names that were added to the array.

Example

Example A-133

```
class my_ms_scen extends vmm_ms_scenario;
   . . .
endclass

program test_scenario;
   string scen_name_arr[$];
  vmm_ms_scenario_gen my_ms_gen = new("MS Scenario Gen", 9);
   my_ms_scen ms_scen = new;
   . . .
   initial begin
      . . .
      `vmm_note(log,"Registering MS scenario \n");
      my_ms_gen.register_ms_scenario("MS-SCEN-1",ms_scen);
      my_ms_gen.register_ms_scenario("MS-SCEN-2",ms_scen);
```

```
      . . .

my_ms_gen.get_all_ms_scenario_names(ms_scen,scen_name_arr)
;
      . . .
   end

endprogram
```

vmm_ms_scenario_gen::replace_ms_scenario()

Replace a scenario descriptor

SystemVerilog

```
virtual function void replace_ms_scenario(string name,
    vmm_ms_scenario scenario)
```

OpenVera

Not supported.

Description

Registers the specified multi-stream scenario under the specified name, replacing the scenario previously registered under that name (if any). The same scenario may be registered multiple times under different names, thus creating an alias to the same scenario.

Registering a scenario implicitly appends it to the scenario set if it is not already in the `vmm_ms_scenario_gen::scenario_set[$]` array. The replaced scenario is removed from `vmm_ms_scenario_gen::scenario_set[$]` if it is not also registered under another name.

Example

Example A-134

```
class my_ms_scen extends vmm_ms_scenario;
    . . .
endclass

program test_scenario;
  vmm_ms_scenario_gen my_ms_gen = new("MS Scenario Gen", 9);
```

```
   my_ms_scen ms_scen = new;
     . . .
   initial begin
      . . .
         my_ms_gen.register_ms_scenario("MS
SCEN-1",ms_scen);
         my_ms_gen.register_ms_scenario("MS
SCEN-2",ms_scen);
         . . .
      vmm_log(log,"Replacing MS scenario \n");
      my_ms_gen.replace_ms_scenario("MS SCEN-1",ms_scen);
      . . .
    end
endprogram
```

vmm_ms_scenario_gen::unregister_ms_scenario()

Unregister a scenario descriptor

SystemVerilog

```
virtual function bit unregister_ms_scenario(
    vmm_ms_scenario scenario)
```

OpenVera

Not supported.

Description

Completely unregisters the specified multi-stream scenario descriptor and returns TRUE if it exists in the registry. The unregistered scenario is also removed from the vmm_ms_scenario_gen::scenario_set[$] array.

Example

Example A-135

```
class my_ms_scen extends vmm_ms_scenario;
    . . .
endclass

program test_scenario;
  vmm_ms_scenario_gen my_ms_gen = new("MS Scenario Gen", 9);
   my_ms_scen ms_scen = new;
     . . .
   initial begin
         my_ms_gen.register_ms_scenario("MS
SCEN-1",ms_scen);
         . . .
      if(my_ms_gen.unregister_ms_scenario(ms_scen)
```

```
                vmm_log(log,"Scenario unregistered \n");
            else
                vmm_log(log,"Unable to unregister  \n");
        . . .
    end
endprogram
```

vmm_ms_scenario_gen::unregister_ms_scenario_by_name()

Unregister a scenario descriptor

SystemVerilog

```
virtual function vmm_ms_scenario unregister_ms_scenario(
    string name)
```

OpenVera

Not supported.

Description

Unregisters the multi-stream scenario under the specified name and returns the unregistered scenario descriptor. Returns `NULL` if there is no scenario registered under the specified name.

The unregistered scenario descriptor is removed from `vmm_ms_scenario_gen::scenario_set[$]` if it is not also registered under another name.

Example

Example A-136

```
class my_ms_scen extends vmm_ms_scenario;
    . . .
endclass

program test_scenario;
  vmm_ms_scenario_gen my_ms_gen = new("MS Scenario Gen", 9);
   my_ms_scen ms_scen = new;
     my_ms_scen buffer_scen =new;
     . . .
```

```
   initial begin
         my_ms_gen.register_ms_scenario("MS
SCEN-1",ms_scen);
         . . .
      buffer_scen =
my_ms_gen.unregister_ms_scenario_by_name("MY-SCEN-


1",ms_scen);
         if(buffer_scen == null)
              vmm_log(log,"Returned null value \n");
      . . .
     end
endprogram
```

vmm_ms_scenario_gen::select_scenario

Scenario selection factory

SystemVerilog

```
vmm_ms_scenario_election select_scenario
```

OpenVera

Not supported.

Description

Randomly select the next multi-stream scenario to execute from the `vmm_ms_scenario_gen::scenario_set[$]` array. The selection is performed by calling randomize() on this class property then executing the multi-stream scenario found in the `vmm_ms_scenario_gen::scenario_set[$]` array at the index specified by the `vmm_ms_scenario_election::select` class property.

The default election instance may be replaced by a user-defined extension to modify the scenario election policy.

Example

Example A-137

```
program test_scenario;
   vmm_ms_scenario_gen parent_ms_gen =
new("Parent-MS-Scen-Gen", 11);
   my_ms_scen ms_scen_1 = new;
   . . .
   initial begin
```

```
parent_ms_gen.register_ms_scenario("MS-Scen-1",ms_scen_1);
      . . .

parent_ms_gen.select_scenario.round_robin.constrint_mode(0
);
      . . .
   end
endprogram
```

vmm_ms_scenario_gen::scenario_set[$]

Multi-stream scenarios available for execution

SystemVerilog

```
vmm_ms_scenario scenatio_set[$]
```

OpenVera

Not supported.

Description

Multi-stream scenarios available for execution by this generator. The scenario executed next is selected by randomizing the `vmm_ms_scenario_gen::select_scenario` class property.

Multi-stream scenario instances in this array should be managed through the `vmm_ms_scenario_gen::register_ms_scenario()`, `vmm_ms_scenario_gen::replace_ms_scenario()` and `vmm_ms_scenario_gen::unregister_ms_scenario()` methods.

Example

Example A-138

```
class my_ms_scen extends vmm_ms_scenario;
   . . .
endclass

program test_scenario;
   vmm_ms_scenario_gen parent_ms_gen =
new("Parent-MS-Scen-Gen", 11);
```

```
   my_ms_scen ms_scen_1 = new;
   my_ms_scen ms_scen_2 = new;
   . . .
   initial begin

parent_ms_gen.register_ms_scenario("MS-Scen-1",ms_scen_1);

parent_ms_gen.register_ms_scenario("MS-Scen-2",ms_scen_2);
      . . .
      buffer_ms_gen =
parent_ms_gen.unregister_ms_scenario(ms_scen_1);
      current_size = parent_ms_gen.scenario_set.size();
    `vmm_note(log, `vmm_sformatf("Current size of scenario
set is %


d\n",current_size);

   end
endprogram
```

vmm_ms_scenario_gen::register_channel()

Register an output channel

SystemVerilog

```
virtual function void register_channel(string name,
     vmm_channel chan)
```

OpenVera

Not supported.

Description

Registers the specified output channel under the specified logical name. The same channel may be registered multiple times under different names, thus creating an alias to the same channel.

Once registered, the output channel becomes available under the specified logical name to multi-stream scenarios via the `vmm_ms_scenario::get_channel()` method.

It is an error to attempt to register a channel under a name that already exists. Use `vmm_ms_scenario_gen::replace_channel()` to replace a registered scenario.

Example

Example A-139

```
`vmm_channel(atm_cell)
`vmm_scenario_gen(atm_cell, "atm_trans")
```

```
program test_scen;
    vmm_ms_scenario_gen my_ms_gen = new("MS Scenario Gen",
11);
    atm_cell_channel ms_chan_1=new("MS-CHANNEL-1",
"MY_CHANNEL");
    . . .
    initial begin
        . . .
        vmm_log(log,"Registering channel \n");

my_ms_gen.register_channel("MS-channel-1",ms_chan_1);
        . . .
    end
endprogram
```

vmm_ms_scenario_gen::channel_exists()

Checks if a channel is registered under a specified name

SystemVerilog

```
virtual function bit channel_exists(string name)
```

OpenVera

Not supported.

Description

Returns TRUE if there is an output channel registered under the specified name. Returns FALSE otherwise.

Use `vmm_ms_scenario_gen::get_channel()` to retrieve a channel under a specified name.

Example

Example A-140

```
`vmm_channel(atm_cell)
`vmm_scenario_gen(atm_cell, "atm_trans")

program test_scen;
    vmm_ms_scenario_gen my_ms_gen = new("MS Scenario Gen",
11);
    atm_cell_channel ms_chan_1=new("MS-CHANNEL-1",
"MY_CHANNEL");
    . . .
    initial begin
        vmm_log(log,"Registering channel \n");

my_ms_gen.register_channel("MS-CHANNEL-1",ms_chan_1);
```

```
            . . .
            if(my_ms_gen.channel_exists("MS_CHANNEL-1"))
                vmm_log(log,"Channel exists\n");
            else
                vmm_log(log,"Channel not yet registered\n");
            . . .
        end
    endprogram
```

vmm_ms_scenario_gen::get_channel()

Returns the channel registered under a specified name

SystemVerilog

```
virtual function vmm_channel get_channel(
     string name)
```

OpenVera

Not supported.

Description

Returns the output channel registered under the specified name. Issues a warning message and returns NULL if there are no channels registered under that name.

Example

Example A-141

```
`vmm_channel(atm_cell)
`vmm_scenario_gen(atm_cell, "atm_trans")

program test_scen;
     vmm_ms_scenario_gen my_ms_gen = new("MS Scenario Gen",
11);
     atm_cell_channel ms_chan_1=new("MS-CHANNEL-1",
"MY_CHANNEL");
     atm_cell_channel buffer_chan = new("BUFFER","MY_BC");
     . . .
     initial begin
          vmm_log(log,"Registering channel \n");

my_ms_gen.register_channel("MS-CHANNEL-1",ms_chan_1);
```

```
          . . .
          buffer_chan =
my_ms_gen.get_channel("MS-CHANNEL-1");
          . . .
     end
endprogram
```

vmm_ms_scenario_gen::get_channel_name()

Returns a name under which a channel is registered

SystemVerilog

```
virtual function string get_names_by_channel(
     vmm_channel chan)
```

OpenVera

Not supported.

Description

Return a names under which the specified channel is registered. Returns "" if the channel is not registered.

Example

Example A-142

```
`vmm_channel(atm_cell)
`vmm_scenario_gen(atm_cell, "atm_trans")

program test_scen;
     vmm_ms_scenario_gen my_ms_gen = new("MS Scenario Gen",
11);
     atm_cell_channel ms_chan_1=new("MS-CHANNEL-1",
"MY_CHANNEL");
     string buffer_chan_name;
     . . .
     initial begin
          vmm_log(log,"Registering channel \n");

my_ms_gen.register_channel("MS-CHANNEL-1",ms_chan_1);
          . . .
          buffer_chan_name =
```

```
my_ms_gen.get_channel_name(ms_chan_1);
          . . .
    end
endprogram
```

vmm_ms_scenario_gen::get_names_by_channel()

Returns the names under which a channel is registered

SystemVerilog

```
virtual function void get_names_by_channel(
     vmm_channel chan,
     ref string   name[$])
```

OpenVera

Not supported.

Description

Appends the names under which the specified output channel is registered. Returns the number of names that were added to the array.

Example

Example A-143

```
`vmm_channel(atm_cell)
`vmm_scenario_gen(atm_cell, "atm_trans")

program test_scen;
   vmm_ms_scenario_gen my_ms_gen = new("MS Scenario Gen",
11);
   atm_cell_channel ms_chan_1=new("MS-CHANNEL-1",
"MY_CHANNEL");
   string channel_name_array[$];
   . . .
   initial begin
      `vmm_note(log,"Registering channel \n");
     my_ms_gen.register_channel("MS-CHANNEL-1",ms_chan_1);
```

```
      . . .

my_ms_gen.get_names_by_channel(ms_chan_1,channel_name_arra
y);
      . . .
   end
endprogram
```

vmm_ms_scenario_gen::get_all_channel_names()

Returns all the names in the channel registry

SystemVerilog

```
virtual function void get_all_channel_names(
    ref string name[$])
```

OpenVera

Not supported.

Description

Appends the names under which an output channel is registered. Returns the number of names that were added to the array.

Example

Example A-144

```
`vmm_channel(atm_cell)
`vmm_scenario_gen(atm_cell, "atm_trans")

program test_scen;
   vmm_ms_scenario_gen my_ms_gen = new("MS Scenario Gen",
11);
   atm_cell_channel ms_chan_1=new("MS-CHANNEL-1",
"MY_CHANNEL");
   string channel_name_array[$];
   . . .
   initial begin
      `vmm_note(log,"Registering channel \n");
     my_ms_gen.register_channel("MS-CHANNEL-1",ms_chan_1);
      . . .
      my_ms_gen.get_all_channel_names(channel_name_array);
      . . .
```

```
    end
endprogram
```

vmm_ms_scenario_gen::replace_channel()

Replace an output channel

SystemVerilog

```
virtual function void replace_channel(string name,
     vmm_channel chan)
```

OpenVera

Not supported.

Description

Registers the specified output channel under the specified name, replacing the channel previously registered under that name (if any). The same channel may be registered multiple times under different names, thus creating an alias to the same output channel.

Example

Example A-145

```
`vmm_channel(atm_cell)
`vmm_scenario_gen(atm_cell, "atm_trans")

program test_scen;
     vmm_ms_scenario_gen my_ms_gen = new("MS Scenario Gen",
11);
     atm_cell_channel ms_chan_1=new("MS-CHANNEL-1",
"MY_CHANNEL");
     . . .
     initial begin
          vmm_log(log,"Registering channel \n");

my_ms_gen.register_channel("MS-CHANNEL-1",ms_chan_1);
```

```
my_ms_gen.register_channel("MS-CHANNEL-2",ms_chan_1);
          . . .
          vmm_log(log,"Replacing the channel \n");

my_ms_gen.replace_channel("MS-CHANNEl-1",ms_chan_1);
          . . .
     end
endprogram
```

vmm_ms_scenario_gen::unregister_channel()

Unregister an output channel

SystemVerilog

```
virtual function bit unregister_channel(
     vmm_channel chan)
```

OpenVera

Not supported.

Description

Completely unregisters the specified output channel and returns TRUE if it exists in the registry.

Example

Example A-146

```
`vmm_channel(atm_cell)
`vmm_scenario_gen(atm_cell, "atm_trans")

program test_scen;
     vmm_ms_scenario_gen my_ms_gen = new("MS Scenario Gen",
11);
     atm_cell_channel ms_chan_1=new("MS-CHANNEL-1",
"MY_CHANNEL");
     . . .
     initial begin
          vmm_log(log,"Registering channel \n");

my_ms_gen.register_channel("MS-CHANNEL-1",ms_chan_1);
          . . .
          if(my_ms_gen.unregister_channel(ms_chan_1)
               vmm_log(log,"Channel has been
```

```
unregistered\n");
          . . .
     end
endprogram
```

vmm_ms_scenario_gen::unregister_channel_by_name()

Unregister an output channel

SystemVerilog

```
virtual function vmm_channel unregister_channel(
     string name)
```

OpenVera

Not supported.

Description

Unregisters the output channel under the specified name and returns the unregistered channel. Returns NULL if there is no channel registered under the specified name.

Example

Example A-147

```
`vmm_channel(atm_cell)
`vmm_scenario_gen(atm_cell, "atm_trans")

program test_scen;
     vmm_ms_scenario_gen my_ms_gen = new("MS Scenario Gen",
11);
     atm_cell_channel ms_chan_1=new("MS-CHANNEL-1",
"MY_CHANNEL");
     atm_cell_channel buffer_chan = new("BUFFER","MY_BC");
     . . .
     initial begin
          vmm_log(log,"Registering channel \n");

my_ms_gen.register_channel("MS-CHANNEL-1",ms_chan_1);
```

```
          . . .
          vmm_log(log,"Unregistered channel by name \n");
          buffer_chan =
my_ms_gen.unregister_channel_by_name("MS-CHANNEL-


1");
          . . .
     end
endprogram
```

vmm_ms_scenario_gen::register_ms_scenario_gen()

Register a sub-generator

SystemVerilog

```
virtual function void register_ms_scenario_gen(string name,
    vmm_ms_scenario_gen gen)
```

OpenVera

Not supported.

Description

Registers the specified sub-generator under the specified logical name. The same generator may be registered multiple times under different names, therefore creating an alias to the same generator.

Once registered, the multi-stream generator becomes available under the specified logical name to multi-stream scenarios via the `vmm_ms_scenario::get_ms_scenario()` method to create hierarchical multi-stream scenarios.

It is an error to attempt to register a generator under a name that already exists. Use `vmm_ms_scenario_gen::replace_ms_scenario_gen()` to replace a registered generator.

Example

Example A-148

```
program test_scen;
    vmm_ms_scenario_gen parent_ms_gen =
```

```
new("Parent-MS-Scen-Gen", 11);
     vmm_ms_scenario_gen child_ms_gen = new("
Child-MS-Scen-Gen", 6);
     . . .
     initial begin
          vmm_log(log,"Registering sub MS generator \n");

parent_ms_gen.register_ms_scenario_gen("Child-MS-Scen-


Gen",child_ms_gen);
          . . .
     end
endprogram
```

vmm_ms_scenario_gen::ms_scenario_gen_exists()

Checks if a generator is registered under a specified name

SystemVerilog

```
virtual function bit ms_scenario_gen_exists(string name)
```

OpenVera

Not supported.

Description

Returns TRUE if there is a sub-generator registered under the specified name. Returns FALSE otherwise.

Use `vmm_ms_scenario_gen::get_ms_scenario_gen()` to retrieve a sub-generator under a specified name.

Example

Example A-149

```
program test_scen;
    vmm_ms_scenario_gen parent_ms_gen =
new("Parent-MS-Scen-Gen", 11);
    vmm_ms_scenario_gen child_ms_gen = new("
Child-MS-Scen-Gen", 6);
    . . .
    initial begin
        vmm_log(log,"Registering sub MS generator \n");

parent_ms_gen.register_ms_scenario_gen("Child-MS-Scen-


Gen",child_ms_gen);
```

```
          . . .

if(parent_ms_gen.ms_scenario_gen_exists("Child-MS-Scen-Gen
"))
               `vmm_note(log, "Generator exists in
registry");
          else
               `vmm_note(log, "Generator doesn't exists in
registry");
          . . .
     end
endprogram
```

vmm_ms_scenario_gen::get_ms_scenario_gen()

Returns the sub-generator registered under a specified name

SystemVerilog

```
virtual function vmm_ms_scenario_gen get_ms_scenario_gen(
    string name)
```

OpenVera

Not supported.

Description

Returns the sub-generator registered under the specified name. Issues a warning message and returns `NULL` if there are no generators registered under that name.

Example

Example A-150

```
program test_scenario;
    vmm_ms_scenario_gen parent_ms_gen =
new("Parent-MS-Scen-Gen", 11);
    vmm_ms_scenario_gen child_ms_gen  =
new("Child-MS-Scen-Gen", 6);
    vmm_ms_scenario_gen buffer_ms_gen =
new("Buffer-MS-Scen-Gen", 6);
    . . .
   initial begin
      vmm_log(log,"Registering sub MS generator \n");

parent_ms_gen.register_ms_scenario_gen("Child-MS-Scen-
```

```
Gen",child_ms_gen);
        . . .
         buffer_ms_gen =
parent_ms_gen.get_ms_scenario_gen("Child-MS-Scen


-Gen");
         . . .
    end

endprogram
```

vmm_ms_scenario_gen::get_ms_scenario_gen_name()

Returns a names under which a generator is registered

SystemVerilog

```
virtual function string get_names_by_ms_scenario_gen(
     vmm_ms_scenario_gen gen)
```

OpenVera

Not supported.

Description

Returns a names under which the specified sub-generator is registered. Returns "" if the generator is not registered.

Example

Example A-151

```
program test_scenario;
     string buffer_ms_gen_name;
     vmm_ms_scenario_gen parent_ms_gen =
new("Parent-MS-Scen-Gen", 11);
     vmm_ms_scenario_gen child_ms_gen  =
new("Child-MS-Scen-Gen", 6);
   . . .
     initial begin
       vmm_log(log,"Registering sub MS generator \n");

parent_ms_gen.register_ms_scenario_gen("Child-MS-Scen-


Gen",child_ms_gen);
        . . .
         buffer_ms_gen_name =
```

```
parent_ms_gen.get_ms_scenario_gen_name


(child_ms_gen);
          . . .
     end

endprogram
```

vmm_ms_scenario_gen::get_names_by_ms_scenario_gen()

Returns the names under which a generator is registered

SystemVerilog

```
virtual function void get_names_by_ms_scenario_gen(
     vmm_ms_scenario_gen gen,
     ref string           name[$])
```

OpenVera

Not supported.

Description

Appends the names under which the specified sub-generator is registered. Returns the number of names that were added to the array.

Example

Example A-152

```
program test_scenario;
   string ms_gen_names_arr[$];
   vmm_ms_scenario_gen parent_ms_gen =
       new("Parent-MS-Scen-Gen", 11);
   vmm_ms_scenario_gen child_ms_gen  =
       new("Child-MS-Scen-Gen", 6);
   . . .
   initial begin
      `vmm_note(log,"Registering sub MS generator \n");
      parent_ms_gen.register_ms_scenario_gen(
         "Child-MS-Scen-Gen", child_ms_gen);
      . . .
```

```
parent_ms_gen.get_names_by_ms_scenario_gen(child_ms_gen,
     ms_gen_names_arr);
      . . .
   end

endprogram
```

vmm_ms_scenario_gen::get_all_ms_scenario_gen_names()

Returns all the names in the generator registry

SystemVerilog

```
virtual function void get_all_ms_scenario_gen_names(
    ref string name[$])
```

OpenVera

Not supported.

Description

Appends the names under which a sub-generator is registered. Returns the number of names that were added to the array.

Example

Example A-153

```
program test_scenario;
   string ms_gen_names_arr[$];
   vmm_ms_scenario_gen parent_ms_gen =
new("Parent-MS-Scen-Gen", 11);
   vmm_ms_scenario_gen child_ms_gen  =
new("Child-MS-Scen-Gen", 6);
   . . .
   initial begin
      `vmm_note(log,"Registering sub MS generator \n");

parent_ms_gen.register_ms_scenario_gen("Child-MS-Scen-Gen"
,child_ms_gen);
     . . .

parent_ms_gen.get_all_ms_scenario_gen_names(ms_gen_names_a
rr);
```

```
      . . .
   end

endprogram
```

vmm_ms_scenario_gen::replace_ms_scenario_gen()

Replace a sub-generator

SystemVerilog

```
virtual function void replace_ms_scenario_gen(string name,
    vmm_ms_scenario_gen gen)
```

OpenVera

Not supported.

Description

Registers the specified sub-generator under the specified name, replacing the generator previously registered under that name (if any). The same generator may be registered multiple times under different names, thus creating an alias to the same sub-generator.

vmm_ms_scenario_gen::unregister_ms_scenario_gen()

Unregister a sub-generator

SystemVerilog

```
virtual function bit unregister_ms_scenario_gen(
    vmm_ms_scenario_gen gen)
```

OpenVera

Not supported.

Description

Completely unregisters the specified sub-generator and returns TRUE if it exists in the registry.

Example

Example A-154

```
program test_scenario;
   string buffer_ms_gen_name;
   vmm_ms_scenario_gen parent_ms_gen =
new("Parent-MS-Scen-Gen", 11);
   vmm_ms_scenario_gen child_ms_gen  =
new("Child-MS-Scen-Gen", 6);
   . . .
   initial begin
      vmm_log(log,"Registering sub MS generator \n");

parent_ms_gen.register_ms_scenario_gen("Child-MS-Gen-1",ch
ild_ms_gen);
      . . .

if(parent_ms_gen.unregister_ms_scenario_gen(child_ms_scen)
)
```

```
         vmm_log(log,"Scenario unregistered \n");
      else
         vmm_log(log,"Unable to unregister  \n");
   end

endprogram
```

vmm_ms_scenario_gen::unregister_ms_scenario_gen_by_name()

Unregister a sub-generator

SystemVerilog

```
virtual function vmm_ms_scenario_gen
     unregister_ms_scenario_gen(
         string name)
```

OpenVera

Not supported.

Description

Unregisters the generator under the specified name and returns the unregistered generator. Returns `NULL` if there is no generator registered under the specified name.

Example

Example A-155

```
program test_scenario;
   vmm_ms_scenario_gen parent_ms_gen =
new("Parent-MS-Scen-Gen", 11);
   vmm_ms_scenario_gen child_ms_gen  =
new("Child-MS-Scen-Gen", 6);
   vmm_ms_scenario_gen buffer_ms_gen  =
new("Buffer-MS-Scen-Gen", 6);
   . . .
   initial begin
      vmm_log(log,"Registering sub MS generator \n");

parent_ms_gen.register_ms_scenario_gen("Child-MS-Gen-1",ch
```

```
ild_ms_gen);

parent_ms_gen.register_ms_scenario_gen("Child-MS-Gen-2",ch
ild_ms_gen);
      . . .
      buffer_ms_gen =
parent_ms_gen.unregister_ms_scenario_gen_by_name("Child-MS


-Gen-1");

   end
endprogram
```

vmm_notify

The **vmm_notify** class implements an interface to the notification service. The notification service provides a synchronization mechanism for concurrent threads or transactors. Unlike **event** variables, the operation of the notification is define at configuration time. Furthermore, notification can have status and timestamp information attached to their indication.

Summary

vmm_notify::new()

Create a new instance of this class.

SystemVerilog

```
function new(vmm_log log);
```

OpenVera

Not supported.

Description

Creates a new instance of this class, using the specified message service interface to issue error and debug messages.

vmm_notify::copy()

Copy the current configuration of this notification service interface.

SystemVerilog

```
virtual function vmm_notify copy(vmm_notify to = null);
```

OpenVera

Not supported.

Description

Copies the current configuration of this notification service interface to the specified instance. If no instance is specified, a new one is allocated using the same message service interface as the original one. A reference to the target instance copied is returned.

Only the notification configuration information is copied and merged with any pre-configured notification in the destination instance. Copied notification configuration will replace any pre-existing configuration for the same notification identifier. Status and timestamp information is **not** copied.

vmm_notify::configure()

Define a new notification.

SystemVerilog

```
virtual function int
           configure(int notification_id = -1,
     sync_e sync = ONE_SHOT);
```

OpenVera

Not supported.

Description

Defines a new notification associated with the specified unique identifier. If a negative identifier value is specified, a new, unique identifier greater than 1,000,000 is returned. The thread synchronization mode of a notification is defined when the notification is configured, not when it is triggered or waited upon, using one of the **`vmm_notify::ONE_SHOT`**, **`vmm_notify::BLAST`**, or **`vmm_notify::ON_OFF`** synchronization types. This definition timing prevents a notification from being misused by the triggering or waiting threads.

Table A-7 Notification Synchronization Mode Enumerated Values

Enumerated Value	Broadcasting Operation
vmm_notify::ONE_SHOT	Only threads currently waiting for the notification to be indicated are notified.

Enumerated Value	Broadcasting Operation
vmm_notify::BLAST	All threads waiting for the notification to be indicated in the same timestep at the indication are notified. This mode eliminates certain types of race conditions.
vmm_notify::ON_OFF	The notification is level-sensitive. Notifications remain notified until explicitly reset. Threads waiting for a notification that is still notified will not wait. This mode eliminates certain types of race conditions.

A warning may be issued if a notification is configured more than once.

Notification identifiers numbered from 1,000,000 and up are reserved for automatically generated notification identifiers. Predefined notification identifiers in the VMM base classes use identifiers 999,999 and down. User-defined notification identifiers can thus use values 0 and up.

vmm_notify::is_configured()

Check if the specified notification is configured.

SystemVerilog

```
virtual function int is_configured(int notification_id);
```

OpenVera

Not supported.

Description

Checks if the specified notification is currently configured. If this method returns 0, the notification is not configured. Otherwise, it returns an integer value corresponding to the current **`vmm_notify::ONE_SHOT`**, **`vmm_notify::BLAST`** or **`vmm_notify::ON_OFF`** configuration.

vmm_notify::is_on()

Check if the specified **`vmm_notify::ON_OFF`** notification is currently in the **`notify`** state.

SystemVerilog

```
virtual function bit is_on(int notification_id);
```

OpenVera

Not supported.

Description

If this method returns TRUE, the notification is in the notify state and any call to the **`vmm_notify::wait_for()`** method will not block. A warning is issued if this method is called on any other types of notifications.

vmm_notify::wait_for()

Suspend the execution thread until the specified notification is notified.

SystemVerilog

```
virtual task wait_for(int notification_id);
```

OpenVera

Not supported.

Description

It is an error to specify an unconfigured notification. Use the **`vmm_notify::status()`** function to retrieve any status descriptor attached to the indicated notification.

Example

Example A-156

```
class consumer extends vmm_xactor;
   ...
   virtual task main();
      ...
      while (1) begin
         ...
         this.in_chan.peek(tr);
         tr.notify.wait_for(vmm_data::ENDED);
         this.in_chan.get(tr);
         ...
      end
   endtask: main
endclass: consumer
```

vmm_notify::wait_for_off()

Suspends the execution thread until the specified **`vmm_notify::ON_OFF`** notification is reset.

SystemVerilog

```
virtual task wait_for_off(int notification_id);
```

OpenVera

Not supported.

Description

It is an error to specify an unconfigured or a non-ON/OFF notification. The status returned by subsequent calls to the **`vmm_notify::status()`** function is undefined.

vmm_notify::is_waited_for()

Check if a thread is currently waiting for the specified notification.

SystemVerilog

```
virtual function bit is_waited_for(int notification_id);
```

OpenVera

Not supported.

Description

Checks if a thread is currently waiting for the specified notification, including waiting for an ON/OFF notification to be reset. It is an error to specify an unconfigured notification. The function returns TRUE if there is a thread known to be waiting for the specified notification.

Note that the knowledge about the number of threads waiting for a particular notification is not definitive and may be out of date. As threads call the **`vmm_notify::wait_for()`** method, the fact that they are waiting for the notification is recorded. Once the notification is indicated and each thread returns from the method call, the fact that they are no longer waiting is also recorded. But if the threads are externally terminated via the **`disable`** statement or a timeout, the fact that they are no longer waiting cannot be recorded. In this case, it is up to the terminated threads to report that they are no longer waiting by calling the **`vmm_notify::terminated()`** method.

When a notification is reset with a hard reset, no threads are assumed to be waiting for any notification.

vmm_notify::terminated()

Indicate that a thread waiting for the specified notification has been disabled.

SystemVerilog

```
virtual function void terminated(int notification_id);
```

OpenVera

Not supported.

Description

Indicates to the notification service interface that a thread waiting for the specified notification has been disabled and is no longer waiting.

vmm_notify::status()

Return the status descriptor associated with the notification.

SystemVerilog

```
virtual function vmm_data status(int notification_id);
```

OpenVera

Not supported.

Description

Returns the status descriptor associated with the specified notification when it was last indicated. It is an error to specify an unconfigured notification.

vmm_notify::timestamp()

Return the time when the notification was last indicated.

SystemVerilog

```
virtual function time timestamp(int notification_id);
```

OpenVera

Not supported.

Description

Returns the simulation time when the specified notification was last indicated. It is an error to specify an unconfigured notification.

vmm_notify::indicate()

Indicate the specified notification with the optional status descriptor..

SystemVerilog

```
virtual function void indicate(int notification_id,
    vmm_data status = null);
```

OpenVera

Not supported.

Example

Example A-157

```
class consumer extends vmm_xactor;
   ...
   virtual task main();
      ...
      forever begin
         ...
         this.in_chan.get(tr);
         tr.notify.indicate(vmm_data::STARTED);
         ...
      end
   endtask: main
endclass: consumer
```

vmm_notify::set_notification()

Define the notification using the notification descriptor.

SystemVerilog

```
virtual function void
      set_notification(int notification_id,
     vmm_notification ntfy = null);
```

OpenVera

Not supported.

Description

Defines the specified notification using the specified notification descriptor. If the descriptor is *null*, the notification is undefined and can only be indicated using the **vmm_notify::indicate()** method. If a notification is already defined, the new definition replaces the previous definition.

vmm_notify::get_notification()

Get the notification descriptor associated with the notification.

SystemVerilog

```
virtual function vmm_notification
          get_notification(int notification_id);
```

OpenVera

Not supported.

Description

Gets the notification descriptor associated with the specified notification, if any. If no notification descriptor is associated with the specified notification, *null* is returned.

vmm_notify::reset()

Reset the specified notification.

SystemVerilog

```
virtual function void reset(int notification_id = -1,
    reset_e rst_typ = SOFT);
```

OpenVera

Not supported.

Description

A **`vmm_notify::SOFT`** reset clears the specified **`ON_OFF`** notification and restarts the **`vmm_notification::indicate()`** and **`vmm_notification::reset()`** methods on any attached notification descriptor. A **`vmm_notify::HARD`** reset clears all status information and attached notification descriptor on the specified event and further assumes that no threads are waiting for that notification. If no notification is specified, all notifications are reset.

Example

Example A-158

The following example shows definitions of three user-defined notifications:

```
class bus_mon extends vmm_xactor;
   static int EVENT_A = 0;
   static int EVENT_B = 1;
   static int EVENT_C = 2;
```

```
   function new(...);
      super.new(...);
      super.notify.configure(this.EVENT_A);
      super.notify.configure(this.EVENT_B,
                             vmm_notify::ON_OFF);
      super.notify.configure(this.EVENT_C,
                             vmm_notify::BLAST);
   endfunction
endclass: bus_mon
```

vmm_notify::append_callback()

Register a callback extension.

SystemVerilog

```
function void append_callback(int
notification_id,
     vmm_notify_callbacks cbs);
```

OpenVera

```
task append_callback(integer event_id,
     rvm_notify_callbacks cbs);
```

Description

Append the specified callback extension to the list of registered callbacks for the specified notification. All registered callback extensions are invoked when the specified notification is indicated.

Example

Example A-159

```
class my_callbacks extends vmm_notify_callbacks;
   virtual function void indicated(vmm_data status);
      . . .
   endfunction
endclass

program vmm_notify_test;
   initial begin
      int EVENT_A = 1;
     vmm_log log = new("Notify event", "vmm_notify_test");
      vmm_notify notify = new(log);
      my_callbacks my_callbacks_inst = new;
      void'(notify.configure(EVENT_A));
```

```
      . . .
      `vmm_note(log, "Appending vmm notify call back");
      notify.append_callback(EVENT_A,my_callbacks_inst);
      . . .
   end
endprogram
```

vmm_notify::unregister_callback()

Unregister a callback extension.

SystemVerilog

```
function void unregister_callback(
    int                 notification_id,
    vmm_notify_callbacks sb);
```

OpenVera

```
task unregister_callback(integer             event_id,
    rvm_notify_callbacks sb);
```

Description

Unregister the specified callback extension from the notification service interface for the specified notification. An error is issued if the specified callback extension was not previously registered with the specified notification.

Example

Example A-160

```
class my_callbacks extends vmm_notify_callbacks;
   virtual function void indicated(vmm_data status);
      . . .
   endfunction
endclass

program vmm_notify_test;
   initial begin
      int EVENT_A = 1;
      vmm_log log = new("Notify event", "vmm_notify_test");
      vmm_notify notify = new(log);
```

```
      my_callbacks my_callbacks_inst = new;
      void'(notify.configure(EVENT_A));
      . . .
      `vmm_note(log, "Unregistering vmm notify call back");
    notify.unregister_callback(EVENT_A,my_callbacks_inst);
      . . .
   end
endprogram
```

vmm_notify::register_vmm_sb_ds()

Refer to the *VMM Scoreboard User Guide*.

vmm_notify::unregister_vmm_sb_ds()

Refer to the *VMM Scoreboard User Guide*.

vmm_notification

This class is used to describe a notification that can be autonomously indicated or reset based on a user-defined behavior, such as the composition of other notifications or external events. Notification descriptors are attached to notifications using the **`vmm_notify::set_notification()`** method.

Summary

vmm_notification::indicated()

Define a method that causes the notification attached to the descriptor to be indicated.

SystemVerilog

```
virtual task indicated(ref vmm_data status);
```

OpenVera

Not supported.

Description

Defines a method that, when it returns, causes the notification attached to the descriptor to be indicated. The value of the **`status`** argument is used as the indicated notification status descriptor. This method is automatically invoked by the notification service interface when a notification descriptor is attached to a notification using the **`vmm_notify::set_notification()`** method.

This method must be overloaded in user-defined class extensions. It can be used to implement arbitrary notification mechanisms, such as notifications based on a complex composition of other indications (for example, notification expressions) or external events.

Example

```
class my_callbacks extends vmm_notify_callbacks;
   virtual function void indicated(vmm_data status);
      . . .
   endfunction
endclass
```

vmm_notification::reset()

Define a method that causes the ON/OFF notification attached to the notification descriptor to be reset.

SystemVerilog

```
virtual task reset();
```

OpenVera

Not supported.

Description

Defines a method that, when it returns, causes the ON/OFF notification attached to the notification descriptor to be reset. This method is automatically invoked by the notification service interface when a notification definition is attached to a **`vmm_notify::ON_OFF`** notification.

This method must be overloaded in user-defined class extensions.

Example

Example A-161

Example of notification indicated when two other notifications are indicated:

```
class notify_a_and_b extends vmm_notification;
   local vmm_notify notify;
   local int        a;
   local int        b;

   function new(vmm_notify notify,
```

```
                int        a,
                int        b);
      this.notify = notify;
      this.a      = a;
      this.b      = b;
   endfunction new

   virtual task indicate(ref vmm_data status)
      fork
         this.notify.wait_for(a);
         this.notify.wait_for(b);
      join
   endtask
endclass: notify_a_and_b

class bus_mon extends vmm_xactor;

   static int EVENT_A = 0;
   static int EVENT_B = 1;
   static int EVENT_C = 2;

   function new(...);
      super.new(....);
      super.notify.configure(this.EVENT_A);
      super.notify.configure(this.EVENT_B,
                             vmm_notify::ON_OFF);
      super.notify.configure(this.EVENT_C,
                             vmm_notify::BLAST);

      begin
         notify_a_and_b AB = new(super.notify,
                                 this.EVENT_A,
                                 this.EVENT_B);
         super.notify.set_notification(this.EVENT_C,
                                       AB);
      end
   endfunction
endclass: bus_mon
```

vmm_notify_callbacks

Facade class for callback methods provided by the notification service. User-defined extensions of this class must be registered with specific instances of the notification service interface and for specific notifications using the `vmm_notify::append_callback()` method.

This class is a virtual class and cannot be instantiated on its own.

Summary

vmm_notify_callbacks::indicated()

Report that a notification has been indicated.

SystemVerilog

```
virtual function void indicated(vmm_data status);
```

OpenVera

```
virtual task indicated(rvm_data status);
```

Description

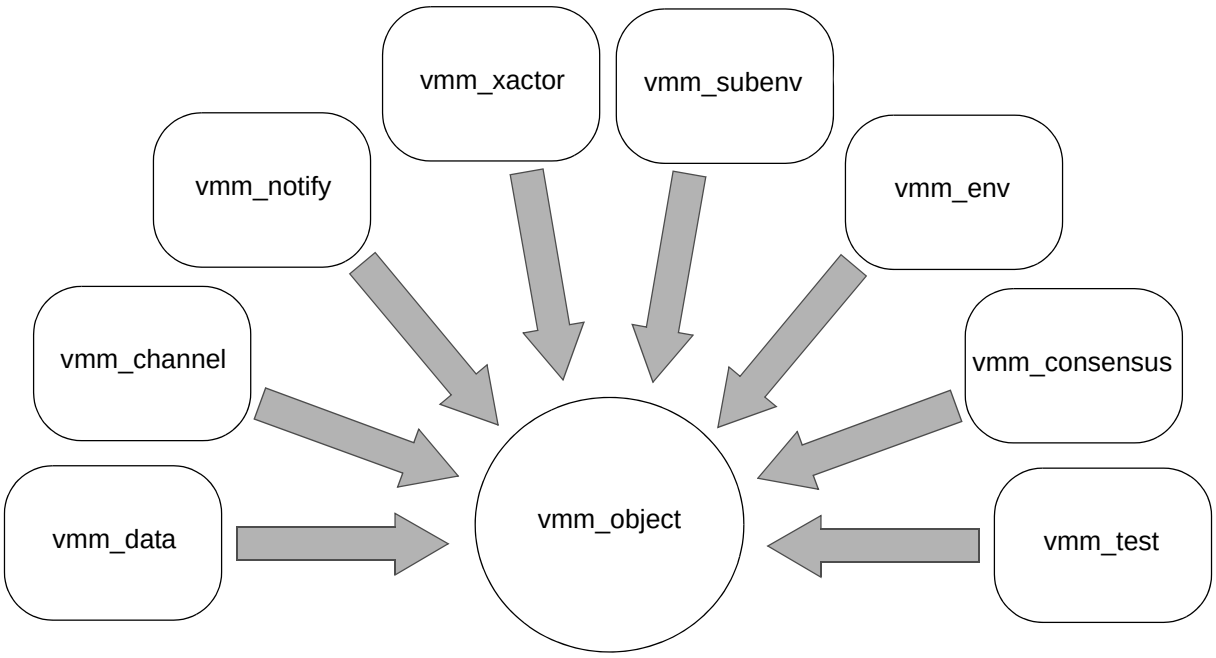
This method is invoked whenever the notification corresponding to the callback extension has been indicated. The status is a reference to the status descriptor specified to the **`vmm_notify::indicate()`** method that caused the notification to be indicated.

The purpose of this callback is very similar to the **`vmm_notify::wait_for()`** method. However, unlike the **`vmm_notify:;wait_for()`** method, it will reliably report multiple indications of the same notification during the same timestep.

vmm_object

The `vmm_object` class is an optional common base class for the `vmm_data`, `vmm_scenario`, `vmm_ms_scenario`, `vmm_channel`, `vmm_notify`, `vmm_xactor`, `vmm_subenv`, `vmm_env`, `vmm_consensus` and `vmm_test` classes.

The **`vmm_object`** class is the foundation of all other VMM classes. Since it is extended by all other classes, common methods such as object display are already defined in the other classes. This makes the user interface consistent across all VMM components.



Each VMM component can be tagged with one of the following specific enumerated values:

VMM_DATA
VMM_CHANNEL
VMM_NOTIFY
VMM_XACTOR
VMM_SUBENV
VMM_ENV
VMM_CONSENSUS
VMM_TEST

This feature is useful to cast a given object to the correct target object.

It is also possible to associate an object to its parent, providing a back reference to the place where it is instantiated, and a hierarchical name.

For backward compatibility reasons, the **vmm_object** class is optional.

This option is enabled by loading the customization definitions files as follow:

```
% vcs ... \
   +define+VMM_PRE_INCLUDE=$VMM_HOME/sv/std_lib/opt/vmm_object.svh \
   +define_VMM_POST_INCLUDE=$VMM_HOME/sv/std_lib/opt/vmm_object.sv \
   ...
% vcs ... \
  +define+VMM_PRE_INCLUDE=$VCS_HOME/etc/rvm/sv/std_lib/opt/vmm_object.svh \
  +define_VMM_POST_INCLUDE=$VCS_HOME/etc/rvm/sv/std_lib/opt/vmm_object.sv \
   ...
```

Summary

vmm_object::type_e

Type of this object.

SystemVerilog

```
typedef enum {
     VMM_UNKNOWN, VMM_OBJECT, VMM_DATA, VMM_SCENARIO,
     VMM_MS_SCENARIO, VMM_CHANNEL, VMM_NOTIFY, VMM_XACTOR,
     VMM_SUBENV, VMM_ENV, VMM_CONSENSUS, VMM_TEST
} type_e
```

OpenVera

Not supported.

Description

Value returned by the "vmm_object::type_e" method to identify the type of this *vmm_object* extension. Once the type is known, a reference to a *vmm_object* can be cast into the corresponding class type.

The *VMM_UNKNOWN* type is an internal value and never returned by the "vmm_object::type_e" method.

The VMM_OBJECT is returned when the type of the object cannot be determined, or to specify any object type to the "vmm_object::type_e" method.

Example

Example A-162

```
program test;
   class tb_env extends vmm_env;
```

```
      type_e env_c_type;
      function new();
         super.new("tb_env");
         end_vote.set_parent(this);
         env_c_type = get_type();
      endfunction
   endclass
   initial
   begin
      string disp_str;
      . . .
      $sformat(disp_str,"Type of env class is :
%s",env.env_c_type.name());
      `vmm_note(log,disp_str);
   end
endprogram
```

vmm_object::new

Constructor.

SystemVerilog

```
function new(vmm_object parent = NULL);
```

OpenVera

Not supported.

Description

Optionally specify a parent object to this object when constructing a `vmm_object` instance. See “`vmm_object::type_e`” for more details on specifying a parent object.

Example

Example A-163

```
class tb_env extends vmm_env;
   subenv s1;
   virtual function void build();
      super.build();
      s1 = new("s1", this.end_vote);
      s1.set_parent(this);
      . . .
   endfunction
endclass
. . .
class tb_object extends vmm_object;
   . . .
   vmm_object::type_e typ;
   . . .
   function new(vmm_object obj);
```

```
        super.new(obj);
        . . .
   endfunction
   . . .
   typ = obj.get_type();
   `vmm_note(log,$psprintf("Get Type for VMM Object ::
%0s",typ.name));
   . . .
endclass
. . .
initial
begin
   tb_env env;
   tb_object obj;

   env = new;
   env.build();
   if (env.s1.get_type() != vmm_object::VMM_SUBENV) begin
      `vmm_error(log, "Wrong type returned from vmm_subenv
instance");
   end
   obj = new(env.s1);
end
```

vmm_object::set_parent()

Specify a parent object.

SystemVerilog

```
function void set_parent(vmm_object parent);
```

OpenVera

Not supported.

Description

Specify a new parent object to this object. Specifying a `NULL` parent breaks the any current parent/child relationship. An object may have only one parent, but the identity of a parent can be changed dynamically.

If this object and the parent object are known to contain their own instance of the message service interface, the `vmm_log` instance in the parent is specified as being above the `vmm_log` instance in the child by calling `parent.is_above(this)`. The instance names of the message service interfaces can then be subsequently made hierarchical by using the “`vmm_log::use_hier_inst_name()`” method.

The presence of the `vmm_object` base class being optional, it is not possible to call this method in code designed to be reusable with and without this base class. To that effect, the `‘VMM_OBJECT_SET_PARENT(_parent, _child)` macro should be used instead. This macro will call this method if the `vmm_object` base class is present but do nothing if not.

Examples

Example A-164

```
this.notify = new(this.log);
this.notify.set_parent(this);
```

Example A-165

```
this.notify = new(this.log);
‘VMM_OBJECT_SET_PARENT(this.notify, this)
```

vmm_object::get_parent()

Returns a parent object.

SystemVerilog

```
function vmm_object get_parent(
     vmm_object::type_e typ = VMM_OBJECT);
```

OpenVera

Not supported.

Description

Return the parent object of the specified type, if any. Returns `NULL` if no such parent is found. Specifying `VMM_OBJECT` returns the immediate parent of any type.

Example

Example A-166

```
class tb_env extends vmm_env;
   tr_scenario_gen gen1;
   function new(string inst, vmm_consensus end_vote);
      . . .
      gen1.set_parent(this);
   endfunction
endclass

initial begin
   tb_env env;
   . . .
   if (env.gen1.randomized_obj.get_parent() != env.gen1)
begin
    `vmm_error(log, "Factory instance in atomic_gen returns
```

```
wrong parent");
   end
end
```

vmm_object::get_type()

Returns the type of the object.

SystemVerilog

```
function vmm_object::type_e get_type();
```

OpenVera

Not supported.

Description

Return the type of this `vmm_object` extension.

Returns `VMM_OBJECT` if it is not one of the known VMM class extensions. `VMM_UNKNOWN` is purely an internal value and is never returned.

Example

Example A-167

```
class tb_env extends vmm_env;
   tr_scenario_gen gen1;
   . . .
   gen1.set_parent(this);
endclass

initial begin
   tb_env env;
   . . .
   if (env.get_type() != vmm_object::VMM_ENV) begin
      `vmm_error(log, "Wrong type returned from vmm_env
instance");
   end
```

```
end
```

vmm_object::get_hier_inst_name()

Returns the hierarchical instance name of the object.

SystemVerilog

```
function string get_hier_inst_name();
```

OpenVera

Not supported.

Description

Return the hierarchical instance name of the object. The instance name is composed of the dot-separated instance names of the message service interface of all the parents of the object.

The hierarchical name is return whether or not the message services interfaces are using hierarchical or flat names.

Example

Example A-168

```
class tb_env extends vmm_env;
   tr_scenario_gen gen1;
   . . .
endclass
initial begin
   string str;
   tb_env env;
   . . .
   str = env.s1.gen1.get_hier_inst_name();
   `vmm_note(log, str);
end
```

vmm_object::display()

Display a description of the object to stdout

SystemVerilog

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

OpenVera

Not supported.

Description

Display the image returned by “vmm_object::type_e” to the standard output.

If this method conflicts with a previously declared method in a class now based on the `vmm_object` class, it can be removed by defining the `‘VMM_OBJECT_NO_DISPLAY` symbol at compile-time.

Example

Example A-169

```
class trans_data extends vmm_data;
   byte data;
   . . .
endclass

initial begin
   . . .
   trans_data trans;
   trans.display("Test Trans: ");
end
```

vmm_object::psdisplay()

Create a description of the object

SystemVerilog

```
virtual function string psdisplay(string prefix = "");
```

OpenVera

Not supported.

Description

Creates a human-readable image of the content of the object and returns it as a string. Each line of the image is prefixed with the specified prefix. The description should not contain a final newline character.

If this method conflicts with a previously declared method in a class now based on the `vmm_object` class, it can be removed by defining the `‘VMM_OBJECT_NO_DISPLAY` symbol at compile-time.

Example

Example A-170

```
class trans_data extends vmm_data;
   byte data;
   . . .
endclass
initial begin
   . . .
   trans_data trans;
   trans.psdisplay("Test Trans: ");
end
```

vmm_opts

This class provides an interface to define and access runtime options. Runtime options can be specified using a combination of command-line arguments and option files specified using a plus-separated list of filenames to the `"+vmm_opts_file="` command-line option.

VMM provides this class to grab runtime options from the simulator command line or an option file. Each option is verified and provided with a default value when it is not specified.

This class is able to learn all possible options. You can dump possible options and default value, allowing verification environment options to be self-documented.

Command-line options can be specified using a plus-separated list of runtime options to the `+vmm_opts` command-line option, or as separate command-line options prefixed with `"+vmm_"`. Using the former is preferable as a warning will be issued if an unknown option is specified.

No constructor is documented because this class is implemented using a singleton pattern. Its functionality is accessed strictly through static methods.

The **`vmm_opts`** class provides a set of static methods that support bit, int and string type runtime arguments. A built-in **`help`** method prints out the details of the available options.

Available **`vmm_opts`** methods are:

```
vmm_opts::get_string("Option Name" , "Default" , "Help for
the option");
vmm_opts::get_bit("Option Name" , "Help for the option");
```

```
vmm_opts::get_int("Option Name" , -1 , "Help for the
option");
```

The methods **`get_string`** and **`get_int`** are called with three arguments. The first argument is the name of the option, the second specifies the default value and the third is a help string for the option.

The method **`get_bit`** is called with two options: the name of the option and the help string. Calling this method enables the option in argument number one.

These methods can be called from the initial block of a program block.

```
program test();

typedef enum { NORMAL , RECORD , PLAYBACK } my_mode;

string md;
my_mode mode;
int num_transactions;
bit scb_mode;

initial
begin
...
  md = vmm_opts::get_string("MODE", "NORMAL" , "Specifies
the mode");
  case(md)
     "NORMAL"   : mode = NORMAL;
         "RECORD"   : mode = RECORD;
         "PLAYBACK" : mode = PLAYBACK;
  endcase
  num_transactions = vmm_opts::get_int("NUM_TRANS", 10,
"Number of Transactions");
  scb_mode = vmm_opts::get_bit("SCB_MODE", "Scoreboard
ON");
...
```

```
end
```

These options can be controlled at runtime by specifying runtime options using **+vmm_opts+***option-name* or **+vmm_***option-name*.

```
simv +vmm_opts+MODE=RECORD +vmm_NUM_TRANS=20
+vmm_opts+SCB_MODE
```

or

```
simv +vmm_opts+MODE=RECORD+NUM_TRANS=20+SCB_MODE , using +
as the separator
```

VMM options can also be provided using option files, where options are provided using **+opt_name**.

For example:

```
//file vmm_opts.txt
+MODE=RECORD
+SCB_MODE
//end file vmm_opts.txt
```

The file is provided at runtime using **+vmm_opts_file**.

```
simv +vmm_opts_file+vmm_opts.txt
+vmm_opts_file+vmm_opts2.txt +vmm_opts+NUM_TRANS=10
```

simv +vmm_opts+help prints the help messages for all available runtime **vmm_options**. These include user defined options and built-in VMM run time options like **+vmm_log_default**.

The help can also be printed from with the code by calling the **vmm_opts::get_help();**.

+vmm_opts+help calls **vmm_opts::get_help()** in **vmm_env::reset_dut()**.

For the help to be printed using **+vmm_opts+help**, a **vmm_env** object must be instantiated and run.

VMM runtime options defined by this simulation are:

- **MODE =** *string* (Unspec'd) – Specifies the mode
- **NUM_TRANS** = *int* (Unspec'd) – Transactions #
- **SCB_ACTIVATED** (0) – Scoreboard ON
- **channel_shared_log** (0) – All VMM channels share the same **vmm_log**
- **force_verbosity =** *str* (Unspec'd) – Overrides the message verbosity level
- **log_default =** *str* (Unspec'd) – Sets the default message verbosity
- **log_nofatal_at_1000** (0) – Supress fatal message for more than 1000…
- **log_nowarn_at_200** (0) – Supress warning message for more than 200

Summary

vmm_opts::get_bit()

Returns a Boolean option value.

SystemVerilog

```
static function bit get_bit(string name, string doc = "");
```

OpenVera

Not supported.

Description

Returns TRUE if the specified option name was specified. Returns FALSE otherwise.

The Boolean runtime option `"foo"` would be supplied using the `"+vmm_opts+...+foo+..."` command-line option, or `"+vmm_foo"` command-line option, or the line `"+foo"` in the option file.

The `"doc"` argument is a short description of the runtime argument that will be displayed by the `vmm_opts::get_help()` method. If it has been previously defined for the specified option through a prior call of this method, the documentation is not redefined.

Example

Example A-171

```
class opt extends vmm_opts;
   . . .
endclass
```

```
initial begin
   opt o;
   bit option;
   option = o.get_bit("OPT1","Run Time option");
   . . .
end
```

vmm_opts::get_int()

Returns an integer option value.

SystemVerilog

```
static function int get_int(string name,
     int dflt = 0,
     string doc = "");
```

OpenVera

Not supported.

Description

Returns the integer value specified as the argument of the specified runtime option. If the runtime option was not supplied, returns the specified default value. Different calls specifying the same option may have different default values.

The integer value "5" for runtime option "foo" would be supplied using the "+vmm_opts+...+foo=5+..." command-line option, or "+vmm_foo=5" command-line option, or the line "+foo=5" in the option file.

The "doc" argument is a short description of the runtime argument that will be displayed by the `vmm_opts::get_help()` method. If it has been previously defined for the specified option through a prior call of this method, the documentation is not redefined.

Example

Example A-172

```
class opt extends vmm_opts;
   . . .
endclass

initial begin
   int option;
   option = o.get_int("OPT1", 0, "Run Time option");
   . . .
end
```

vmm_opts::get_string()

Returns a string option value.

SystemVerilog

```
static function string get_string(string name,
    string dflt = "",
    string doc = "");
```

OpenVera

Not supported.

Description

Returns the string value specified as the argument of the specified runtime option. If the runtime option was not supplied, returns the specified default value. Different calls specifying the same option may have different default values.

The string value `"bar"` for runtime option `"foo"` would be supplied using the `"+vmm_opts+...+foo=bar+..."` command-line option, or `"+vmm_foo=bar"` command-line option, or the line `"+foo=bar"` in the option file.

The `"doc"` argument is a short description of the runtime argument that will be displayed by the `vmm_opts::get_help()` method. If it has been previously defined for the specified option through a prior call of this method, the documentation is not redefined.

Example

Example A-173

```
class opt extends vmm_opts;
   . . .
endclass

initial begin
   opt o;
   string option;
   option = o.get_string("OPT1", "", "Run Time option");
   . . .
end
```

vmm_opts::get_help()

Display a list of all known runtime options.

SystemVerilog

```
static function void get_help();
```

OpenVera

Not supported.

Description

Display a human-readable list of all runtime options queried so far.

This method is automatically called, followed by a call to `$finish()`, by the `vmm_env::reset_dut()` method if the `+vmm_help` command-line option is supplied.

Example

Example A-174

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

   if ($test$plusargs("tb_help")) begin
      vmm_opts::get_help();
      $finish;
   end
   ...
endtask
```

vmm_scenario

Base class for all user-defined scenarios. This class extends from `vmm_data`.

Summary

‘vmm_scenario_new()

Start of explicit constructor implementation.

SystemVerilog

```
‘vmm_scenario_new(class-name)
```

OpenVera

Not supported.

Description

Specify that an explicit user-defined constructor is used instead of the default constructor provided by the short-hand macros. Also declares a “vmm_log” instance that can be passed to the base class constructor. Use this macro when data members must be explicitly initialized in the constructor.

The class-name specified must be the name of the vmm_scenario extension class that is being implemented.

This macro should be followed by the constructor declaration and must precede the shorthand data member section i.e., be located before the “‘vmm_scenario_member_begin()” macro.

Example

Example A-175

```
class my_scenario extends vmm_ms_scenario;
   ...
   ‘vmm_scenario_new(my_scenario)
      function new(vmm_scenario parent = null);
         super.new(parent)
```

```
      ...
    endfunction

  `vmm_scenario_member_begin(my_scenario)
    ...
  `vmm_scenario_member_end(my_scenario)
  ...
endclass
```

‘vmm_scenario_member_begin()

Start of shorthand section.

SystemVerilog

```
‘vmm_scenario_member_begin(class-name)
```

OpenVera

Not supported.

Description

Start the shorthand section providing a default implementation for the `psdisplay()`, `is_valid()`, `allocate()`, `copy()`, and `compare()` methods. A default implementation for the constructor is also provided unless the “‘`vmm_scenario_new()`” macro as been previously specified.

The class-name specified must be the name of the `vmm_scenario` extension class that is being implemented.

The shorthand section can only contain shorthand macros and must be terminated by a “‘`vmm_scenario_member_end()`” .

Example

Example A-176

```
class my_scenario extends vmm_data;
   ...
   ‘vmm_scenario_member_begin(my_scenario)
      ...
   ‘vmm_scenario_member_end(my_scenario)
   ...
```

```
endclass
```

‘vmm_scenario_member_end()

End of shorthand section.

SystemVerilog

```
‘vmm_scenario_member_end(class-name)
```

OpenVera

Not supported.

Description

Terminate the shorthand section providing a default implementation for the `psdisplay()`, `is_valid()`, `allocate()`, `copy()`, and `compare()` methods.

The class-name specified must be the name of the `vmm_scenario` extension class that is being implemented.

The shorthand section must have been started by a “‘vmm_scenario_member_begin()” .

Example

Example A-177

```
class eth_scenario extends vmm_data;
   ...
   ‘vmm_scenario_member_begin(eth_scenario)
      ...
   ‘vmm_scenario_member_end(eth_scenario)
   ...
endclass
```

‘vmm_scenario_member_scalar*()

The shorthand implementation for a scalar data member.

SystemVerilog

```
‘vmm_scenario_member_scalar(member-name,
                            vmm_data::do_what_e do_what)

‘vmm_scenario_member_scalar_array(member-name,
                                  vmm_data::do_what_e do_what)

‘vmm_scenario_member_scalar_da(member-name,
                               vmm_data::do_what_e do_what)

‘vmm_scenario_member_scalar_aa_scalar(member-name,
                                  vmm_data::do_what_e do_what)

‘vmm_scenario_member_scalar_aa_string(member-name,
                                  vmm_data::do_what_e do_what)
```

OpenVera

Not supported.

Description

Add the specified scalar-type, fixed array of scalars, dynamic array of scalars, scalar-indexed associative array of scalars or string-indexed associative array of scalars data member to the default implementation of the methods specified by the `do_what` argument.

A scalar is an integral type, such as bit, bit vector, and packed unions.

The shorthand implementation must be located in a section started by “‘vmm_scenario_member_begin()” .

Example

Example A-178

```
class eth_scenario extends vmm_data;
   rand bit [47:0] da;
   ...
   ‘vmm_scenario_member_begin(eth_scenario)
      ‘vmm_scenario_member_scalar(da, DO_ALL);
      ...
   ‘vmm_scenario_member_end(eth_scenario)
   ...
endclass
```

‘vmm_scenario_member_string*()

The shorthand implementation for a string data member.

SystemVerilog

```
‘vmm_scenario_member_string(member-name,
                            vmm_data::do_what_e do_what)

‘vmm_scenario_member_string_array(member-name,
                                  vmm_data::do_what_e do_what)

‘vmm_scenario_member_string_da(member-name,
                               vmm_data::do_what_e do_what)

‘vmm_scenario_member_string_aa_scalar(member-name,
                                  vmm_data::do_what_e do_what)

‘vmm_scenario_member_string_aa_string(member-name,
                                  vmm_data::do_what_e do_what)
```

OpenVera

Not supported.

Description

Add the specified string-type, fixed array of strings, dynamic array of strings, scalar-indexed associative array of strings or string-indexed associative array of strings data member to the default implementation of the methods specified by the `do_what` argument.

The shorthand implementation must be located in a section started by “‘vmm_scenario_member_begin()” .

‘vmm_scenario_member_enum*()

The shorthand implementation for an enumerated data member.

SystemVerilog

```
‘vmm_scenario_member_enum(member-name,
                        vmm_data::do_what_e do_what)

‘vmm_scenario_member_enum_array(member-name,
                              vmm_data::do_what_e do_what)

‘vmm_scenario_member_enum_da(member-name,
                            vmm_data::do_what_e do_what)

‘vmm_scenario_member_enum_aa_scalar(member-name,
                                  vmm_data::do_what_e do_what)

‘vmm_scenario_member_enum_aa_string(member-name,
                                  vmm_data::do_what_e do_what)
```

OpenVera

Not supported.

Description

Add the specified enum-type, fixed array of enums, dynamic array of enums, scalar-indexed associative array of enums or string-indexed associative array of enums data member to the default implementation of the methods specified by the `do_what` argument.

The shorthand implementation must be located in a section started by “‘vmm_scenario_member_begin()” .

‘vmm_scenario_member_vmm_data*()

The shorthand implementation for a vmm_data-based data member.

SystemVerilog

```
‘vmm_scenario_member_vmm_data(member-name,
                              vmm_data::do_what_e do_what,
                              vmm_data::do_how_e  do_how)

‘vmm_scenario_member_vmm_data_array(member-name,
                              vmm_data::do_what_e do_what,
                              vmm_data::do_how_e  do_how)

‘vmm_scenario_member_vmm_data_da(member-name,
                              vmm_data::do_what_e do_what,
                              vmm_data::do_how_e  do_how)

‘vmm_scenario_member_vmm_data_aa_scalar(member-name,
                              vmm_data::do_what_e do_what,
                              vmm_data::do_how_e  do_how)

‘vmm_scenario_member_vmm_data_aa_string(member-name,
                              vmm_data::do_what_e do_what,
                               vmm_data::do_how_e   do_how)
```

OpenVera

Not supported.

Description

Add the specified vmm_data-type, fixed array of vmm_datas, dynamic array of vmm_datas, scalar-indexed associative array of vmm_datas or string-indexed associative array of vmm_datas data member to the default implementation of the methods specified by the `do_what` argument. The `do_how` argument specifies whether the `vmm_data` values must be processed deeply or shallowly.

The shorthand implementation must be located in a section started by “‘vmm_scenario_member_begin()” .

‘vmm_scenario_member_vmm_scenario()

The shorthand implementation for a sub-scenario.

SystemVerilog

```
‘vmm_scenario_member_vmm_scenario(member-name,
                            vmm_data::do_what_e do_what)
```

OpenVera

Not supported.

Description

Add the specified vmm_scenario-type sub-scenario member to the default implementation of the methods specified by the `do_what` argument.

The shorthand implementation must be located in a section started by “‘vmm_scenario_member_begin()” .

‘vmm_scenario_member_handle*()

The shorthand implementation for a class handle data member.

SystemVerilog

```
‘vmm_scenario_member_handle(member-name,
                            vmm_data::do_what_e do_what)

‘vmm_scenario_member_handle_array(member-name,
                                  vmm_data::do_what_e do_what)

‘vmm_scenario_member_handle_da(member-name,
                               vmm_data::do_what_e do_what)

‘vmm_scenario_member_handle_aa_scalar(member-name,
                                  vmm_data::do_what_e do_what)

‘vmm_scenario_member_handle_aa_string(member-name,
                                  vmm_data::do_what_e do_what)
```

OpenVera

Not supported.

Description

Add the specified handle-type fixed array of handles, dynamic array of handles, scalar-indexed associative array of handles or string-indexed associative array of handles data member to the default implementation of the methods specified by the `do_what` argument.

The shorthand implementation must be located in a section started by “‘vmm_scenario_member_begin()” .

‘vmm_scenario_member_user_defined()

User-defined shorthand implementation data member.

SystemVerilog

```
‘vmm_scenario_member_user_defined(member-name,
                                  vmm_data::do_what_e do_what)
```

OpenVera

Not supported.

Description

Add the specified user-defined default implementation of the methods specified by the `do_what` argument.

Refer to the section entitled, “User-defined vmm_data Member Default Implementation” on page 2-6 for details on how to specify the shorthand implementation for a data member.

The shorthand implementation must be located in a section started by “`‘vmm_scenario_member_begin()`” .

vmm_scenario::stream_id

Stream identifier of the randomizing generator.

SystemVerilog

```
int stream_id
```

OpenVera

Not supported.

Description

This data member is set by the scenario generator before randomization to the generator’s stream identifier. This state variable can be used to specific stream-specific constraints or to differentiate stimulus from different streams in a scoreboard.

Example

Example A-179

```
class atm_cell extends vmm_data;
   rand int payload[3];
   . . .
endclass

`vmm_scenario_gen(atm_cell, "atm trans")

class atm_cell_ext extends atm_cell;
   . . .
   constraint test {
      payload[0] == stream_id;
      . . .}
endclass
```

vmm_scenario::scenario_id

Scenario identifier of the randomizing generator.

SystemVerilog

```
int scenario_id
```

OpenVera

Not supported.

Description

This data member is set by the scenario generator before randomization to the generator’s current scenario counter value. This state variable can be used to specifiy scenario-specific constraints or to identify the order of different scenarios within a stream.

Example

Example A-180

```
class atm_cell extends vmm_data;
   rand int payload[3];
   . . .
endclass

`vmm_scenario_gen(atm_cell, "atm trans")

class atm_cell_ext extends atm_cell;
   . . .
   constraint test {
      payload[1] == scenario_id;
      . . .}
```

```
endclass
```

vmm_scenario::scenario_kind

Scenario kind identified.

SystemVerilog

```
rand int unsigned scenario_kind
```

OpenVera

Not supported.

Description

Used to randomly select one of the scenario kinds defined in this random scenario descriptor.

Example

Example A-181

```
`vmm_scenario_gen(atm_cell, "atm trans")

class my_scenario extends atm_cell_scenario;
   . . .
   constraint start_up_const {
      (trans_type == 0 ) -> { scenario_kind inside
{RESET_SEQ,START_UP_SEQ}};
      . . . }
endclass
```

vmm_scenario::length

Length of the scenario.

SystemVerilog

```
rand int unsigned length
```

OpenVera

Not supported.

Description

Random number of transaction descriptor in this random scenario. Constrained to be less than or equal to the maximum number of transactions in the selected scenario kind.

Example

Example A-182

```
`vmm_scenario_gen(atm_cell, "atm trans")

class my_scenario extends atm_cell_scenario;
   . . .
   constraint scen_length {
      if (scenario_kind == START_UP_SEQ)
         { length == 2 } ;
      . . . }
endclass
```

vmm_scenario::repeated

Scenario identifier of the randomizing generator.

SystemVerilog

```
rand int unsigned repeated
```

OpenVera

Not supported.

Description

The number of time the entire scenario is repeated. A repetition value of zero specifies that the scenario will not be repeated, and will be applied only once.

Constrained to zero by default by the “vmm_scenario::repetition” constraint block.

Note that is best to repeat the same transaction instead of creating a scenario of many transactions constrained to be identical.

Example

Example A-183

```
`vmm_scenario_gen(atm_cell, "atm trans")

class my_scenario extends atm_cell_scenario;
   . . .
   constraint scen_repetitions
      {
      if (scenario_kind == START_UP_SEQ)
         { //Note: Default constraint is 0 for repeated.
```

```
            repeated < 4 } ;
      . . .
      }
endclass
```

vmm_scenario::repeat_thresh

Repetition warning threshold.

SystemVerilog

```
static int unsigned repeat_thresh
```

OpenVera

Not supported.

Description

Specifies a threshold value that triggers a warning about possibly unconstrained “vmm_scenario::repeated” data member. Defaults to 100.

Example

Example A-184

```
`vmm_scenario_gen(atm_cell, "atm trans")

class my_scenario extends atm_cell_scenario;
   . . .
   constraint scen_rep_thresh
      {
      if (scenario_kind == START_UP_SEQ)
      { //Note: Default constraint is 100 for repeat_thresh.
            repeat_thresh < 120 } ;
      . . .
      }
endclass
```

vmm_scenario::repetition

Constraint preventing the scenario from being repeated.

SystemVerilog

```
constraint repetition {
   repeated == 0;
}
```

OpenVera

Not supported.

Description

The “vmm_scenario::repeated” data member specifies the number of times a scenario is repeated. It is not often used but, if left unconstrained, can cause stimulus to be erroneously repeatedly applied over 2 billion times on average.

This constraint block constrains this data member to prevent repetition by default. To have a scenario be repeated a random number of times, simply override this constraint block.

Example

Example A-185

```
class many_atomic_scenario
      extends eth_frame_atomic_scenario;
   constraint repetition {
      repeated < 10;
   }
endclass
```

vmm_scenario::define_scenario()

Define a new scenario kind.

SystemVerilog

```
function int unsigned define_scenario(string name,
    int unsigned max_len);
```

OpenVera

Not supported.

Description

Defines a new scenario kind included in this scenario descriptor and return a unique scenario kind identifier. The "vmm_scenario::scenario_kind" data member will randomly select one of the defined scenario kinds. The new scenario kind may have up to the specified number of random transactions.

The scenario kind identifier should be stored in a state variable that can then be subsequently used to specified kind-specific constraints.

Example

Example A-186

```
`vmm_scenario_gen(atm_cell, "atm trans")

class my_scenario extends atm_cell_scenario;
   int unsigned START_UP_SEQ;
   int unsigned RESET_SEQ;
   . . .
   function new()
      START_UP_SEQ = define_scenario("START_UP_SEQ",5);
```

```
      RESET_SEQ    = define_scenario("RESET_SEQ",11);
      . . .
   endfunction
   . . .
endclass
```

vmm_scenario::redefine_scenario()

Redefine an existing scenario kind.

SystemVerilog

```
function void redefine_scenario(int unsigned scenario_kind,
    string name,
    int unsigned max_len);
```

OpenVera

Not supported.

Description

Redefines an existing scenario kind included in this scenario descriptor. The scenario kind may be redefined with a different name or maximum number of random transactions.

Use this method to modify, refine or replace an existing scenario kind in a pre-defined scenario descriptor.

Example

Example A-187

```
class my_scenario extends atm_cell_scenario;
   int unsigned START_UP_SEQ;
   . . .
   function new()
     redefine_scenario(this.START_UP_SEQ,"WAKE_UP_SEQ",5);
      . . .
   endfunction
   . . .
endclass
```

vmm_scenario::scenario_name()

Name of a scenario kind.

SystemVerilog

```
function string scenario_name(int unsigned scenario_kind);
```

OpenVera

Not supported.

Description

Return the name of the specified scenario kind, as defined by the "vmm_scenario::define_scenario()" or "vmm_scenario::redefine_scenario()" methods.

Example

Example A-188

```
class my_scenario extends atm_cell_scenario;
   int unsigned START_UP_SEQ;
   . . .
   function new()
     redefine_scenario(this.START_UP_SEQ,"WAKE_UP_SEQ",5);
      . . .
   endfunction
   . . .
   function post_randomize();
      $display("Name of the redefined scenario is %s
\n",scenario_name


(scenario_kind));
      . . .
```

```
    endfunction
endclass
```

vmm_scenario::psdisplay()

Create an image of the scenario descriptor.

SystemVerilog

```
virtual function string psdisplay(string prefix = "")
```

OpenVera

Not supported.

Description

Create human-readable image of the content of the scenario descriptor.

Example

Example A-189

```
class my_scenario extends atm_cell_scenario;
   int unsigned START_UP_SEQ;
   . . .
   function new()
     redefine_scenario(this.START_UP_SEQ,"WAKE_UP_SEQ",5);
      . . .
   endfunction
   . . .
endclass

initial begin
   . . .
   my_scenario scen_inst = new();
   . . .
   $display("Data of the redefined scenario is %s
\n",scen_inst.psdisplay());
```

```
    . . .
end
```

vmm_scenario::set_parent_scenario()

Define higher-level hierarchical scenario.

SystemVerilog

```
function void set_parent_scenario(
     vmm_scenario parent)
```

OpenVera

Not supported.

Description

Specify the single stream or multiple-stream scenario that is the parent of this scenario. This will allow this scenario to grab a channel that has already been grabbed by the parent scenario.

Example

Example A-190

```
class atm_cell extends vmm_data;
   rand int payload[3];
   . . .
endclass

`vmm_scenario_gen(atm_cell, "atm trans")

program test_scenario;
   . . .
   atm_cell_scenario parent_scen = new;
   atm_cell_scenario child_scen = new;
   . . .
   initial begin
      . . .
```

```
      vmm_log(log,"Setting parent to a child scenarion \n");
       child.scen.set_parent_scenario(parent_scen);
       . . .
   end

endprogram
```

vmm_scenario::get_parent_scenario()

Returns the higher-level hierarchical scenario.

SystemVerilog

```
function vmm_scenario get_parent_scenario()
```

OpenVera

Not supported.

Description

Returns the single stream or multiple-stream scenario that was specified as the parent of this scenario. A scenario with no parent is a top-level scenario.

Example

Example A-191

```
class atm_cell extends vmm_data;
   . . .
endclass

`vmm_scenario_gen(atm_cell, "atm trans")

program test_scenario;
   . . .
   atm_cell_scenario parent_scen = new;
   atm_cell_scenario child_scen = new;
   . . .
   initial begin
      . . .
     vmm_log(log,"Setting parent to a child scenarion \n");
      child.scen.set_parent_scenario(parent_scen);
```

```
      . . .
      if(child_scen.get_parent_scenario() == parent_scen)
        vmm_log(log,"Child scenario has proper parent \n");
        . . .
      else
       vmm_log(log,"Child scenario has improper parent \n");
      . . .
   end

endprogram
```

vmm_scenario_gen

A macro is used to define a class named *class-name*_**scenario_gen** for any user-specified class derived from **vmm_data**[1], using a process similar to the **‘vmm_channel** macro.

The scenario generator class is an extension of the **vmm_xactor** class and as such, inherits all of the public interface elements provided in the base class.

Summary

1. With a constructor callable without any arguments.

‘vmm_scenario_gen()

Define a scenario generator class to generate sequences of related instances.

SystemVerilog

```
‘vmm_scenario_gen(class_name, "Class Description")
```

OpenVera

Not supported.

Description

Defines a scenario generator class to generate sequences of related instances of the specified class. The specified class must be derived from the **`vmm_data`** class and the *`class-name`*`_`**`channel`** class must exist. It must also have a constructor with no arguments or that has default values for all of its arguments.

The macro defines classes named

- *`class-name`*`_`**`scenario_gen`**
- *`class-name`*`_`**`scenario`**
- *`class-name`*`_`**`scenario_election`**
- *`class-name`*`_`**`scenario_gen_callbacks`**

‘vmm_scenario_gen_using()

Define a scenario generator class to generate sequences of related instances.

SystemVerilog

```
‘vmm_scenario_gen_using( class-name , channel-type,
    "Class Description")
```

OpenVera

Not supported.

Description

Defines a scenario generator class to generate sequences of related instances of the specified class, using the specified *class-name*_**channel** output channel. The generated class must be compatible with the specified channel type and both must exist.

This macro should be used only when generating instances of a derived class that must be applied to a channel of the base class.

vmm_scenario_gen::new()

Create a new instance of a scenario generator transactor.

SystemVerilog

```
function new(string instance,
     int stream_id = -1,
     class-name_channel out_chan = null);
```

OpenVera

Not supported.

Description

Creates a new instance of a scenario generator transactor with the specified instance name and optional stream identifier. The generator can be optionally connected to the specified output channel. If no output channel is specified, one will be created internally in the *class-name*_**scenario_gen::out_chan** property.

The name of the transactor is defined as the user-defined class description string specified in the class implementation macro appended with “*Scenario Generator*”.

vmm_scenario_gen::out_chan

Reference the output channel for the instances generated by this transactor.

SystemVerilog

```
class-name_channel out_chan;
```

OpenVera

Not supported.

Description

The output channel may have been specified via the constructor. If no output channel was specified, a new instance is automatically created. The reference in this property may be dynamically replaced but the generator should be stopped during the replacement.

vmm_scenario_gen::stop_after_n_insts

Stop generation after the specified number of transaction or data descriptor instances have been generated.

SystemVerilog

```
int unsigned stop_after_n_insts;
```

OpenVera

Not supported.

Description

The generator will stop after the specified number of transaction or data descriptor instances have been generated and consumed by the output channel. The generator must be reset before it can be restarted. If the value of this property is 0, the generator will not stop on its own based on the number of generated instances (but may still stop based on the number of generated scenarios).

The default value of this property is 0.

vmm_scenario_gen::get_n_insts()

Return the actual number of instances generated.

SystemVerilog

```
function int unsigned get_n_insts();
```

OpenVera

Not supported.

Description

The generator stops after the **`stop_after_n_insts`** limit on the number of instances has been reached and only after entire scenarios have been applied. It can thus generate a few more instances than configured. This method returns the actual number of instances that were generated.

vmm_scenario_gen::stop_after_n_scenarios

Stop generation after the specified number of scenarios have been generated.

SystemVerilog

```
int unsigned stop_after_n_scenarios;
```

OpenVera

Not supported.

Description

The generator will stop after the specified number of scenarios have been generated and entirely consumed by the output channel. The generator must be reset before it can be restarted. If the value of this property is 0, the generator will not stop on its own based on the number of generated scenarios (but may still stop based on the number of generated instances).

The default value of this property is 0.

vmm_scenario_gen::get_n_scenarios()

Return the actual number of scenarios generated.

SystemVerilog

```
function int unsigned get_n_scenarios();
```

OpenVera

Not supported.

Description

The generator stops after the **`stop_after_n_scenarios`** limit on the number of scenarios has been reached and only after entire scenarios have been applied. It can thus generate a few less scenarios than configured. This method returns the actual number of scenarios that were generated.

vmm_scenario_gen::scenario_set[$]

Set of available scenario descriptors that may be repeatedly randomized.

SystemVerilog

```
class-name_scenario scenario_set[$];
```

OpenVera

Not supported.

Description

Set of available scenario descriptors that may be repeatedly randomized to create the random content of the output stream. The *class-name*_**scenario_gen::select_scenario** property is used to determine which scenario descriptor, out of the available set of descriptors, is randomized next. The individual instances of the output stream are then created by calling the *class-name*_**scenario::apply()** method of the randomized scenario descriptor.

By default, this property contains one instance of the atomic scenario descriptor *class-name*_**atomic_scenario**. Out of the box, the scenario generator will generate individual random descriptors.

The **vmm_data::stream_id** property of the randomized instance is assigned the value of the generator's stream identifier before randomization. The **vmm_data::scenario_id** property of the randomized instance is assigned a unique value before

randomization. It will be reset to 0 when the generator is reset and after the specified number of instances or scenarios has been generated.

vmm_scenario_gen::select_scenario

Determine which scenario descriptor will be randomized next.

SystemVerilog

```
class-name_scenario_election select-scenario;
```

OpenVera

Not supported.

Description

References the scenario descriptor selector that is repeatedly randomized to determine which scenario descriptor, out of the available set of scenario descriptors, will be randomized next.

By default, a round-robin selection process is used. The constraint blocks or randomized properties in this instance can be turned off or the instance can be replaced with a user-defined extension to modify the election rules.

vmm_scenario_gen::enum {GENERATED}

Notification identifier for the **`vmm_xactor::notify`** notification service interface.

SystemVerilog

```
enum {GENERATED};
```

OpenVera

Not supported.

Description

Notification identifier for the **`vmm_xactor::notify`** notification service interface provided by the **`vmm_xactor`** base class. It is configured as a **`vmm_notify::ONE_SHOT`** notification and is indicated immediately before a scenario is applied to the output channel. The randomized scenario is specified as the status of the notification.

vmm_scenario_gen::enum {DONE}

Notification identifier for the **`vmm_xactor::notify`** notification service interface.

SystemVerilog

```
enum {DONE};
```

OpenVera

Not supported.

Description

Notification identifier for the **`vmm_xactor::notify`** notification service interface provided by the **`vmm_xactor`** base class. It is configured as a **`vmm_notify::ON_OFF`** notification and is indicated when the generator stops because the specified number of instances or scenarios has been generated. No status information is specified.

vmm_scenario_gen::inject_obj()

Inject the specified descriptor in the output stream.

SystemVerilog

```
virtual task inject_obj(class-name obj);
```

OpenVera

Not supported.

Description

Unlike injecting the descriptor directly in the output channel, it counts toward the number of instances and scenarios generated by this generator and will be subjected to the callback methods as an atomic scenario. The method returns once the descriptor has been consumed by the output channel or it has been dropped by the callback methods.

This method can be used to inject directed stimulus while the generator is running (with unpredictable timing) or when the generated is stopped.

vmm_scenario_gen::inject()

Inject the specified scenario descriptor in the output stream.

SystemVerilog

```
virtual task inject(class-name_scenario scenario);
```

OpenVera

Not supported.

Description

Unlike injecting the descriptors directly in the output channel, it counts toward the number of instances and scenarios generated by this generator and will be subjected to the callback methods. The method returns once the scenario has been consumed by the output channel or it has been dropped by the callback methods.

This method can be used to inject directed stimulus while the generator is running (with unpredictable timing) or when the generated is stopped.

vmm_scenario::define_scenario()

Define a new scenario kind.

SystemVerilog

```
function int unsigned define_scenario(string name,
    int unsigned max-len);
```

OpenVera

Not supported.

Description

Defines a new scenario kind included in this scenario descriptor and returns a unique scenario kind identifier. The "vmm_scenario::scenario_kind" data member will randomly select one of the defined scenario kinds.

vmm_scenario_gen::scenario_count

Number of scenarios generated so far.

SystemVerilog

```
protected int scenario_count;
```

OpenVera

```
protected integer scenario_count;
```

Description

Current count of the number of scenarios generated by or injected through the scenario generator. When it reaches or surpasses the value in `vmm_scenario_gen::stop_after_n_scenarios`, the generator stops.

Example

Example A-192

```
class generator_ext extends pkt_scenario_gen;
   . . .
   virtual task inject(pkt_scenario scenario);
       scenario.scenario_id = this.scenario_count;
   . . .
   endtask

endclass
```

vmm_scenario_gen::inst_count

Number of instances generated so far.

SystemVerilog

```
protected int inst_count;
```

OpenVera

```
protected integer inst_count;
```

Description

Current count of the number of individual instances generated by or injected through the scenario generator. When is reaches or surpasses the value in `vmm_scenario_gen::stop_after_n_insts`, the generator stops.

Example

Example A-193

```
class generator_ext extends pkt_scenario_gen;
   . . .
   function void reset_xactor(reset_e rst_typ = SOFT_RST);
      this.inst_count     = 0;
      . . .
   endfunction

endclass
```

vmm_scenario_gen::register_scenario()

Register a scenario descriptor

SystemVerilog

```
virtual function void register_scenario(string name,
    vmm_scenario scenario)
```

OpenVera

Not supported.

Description

Registers the specified scenario under the specified name. The name under which a scenario is registered does not need to be the same as the name of a kind of scenario defined in the scenario descriptor using **`vmm_scenario::define_scenario()`**. The same scenario may be registered multiple times under different names, therefore, creating an alias to the same scenario.

Registering a scenario implicitly appends it to the scenario set if it is not already in the **`vmm_scenario_gen::scenario_set[$]`** array.

It is an error to attempt to register a scenario under a name that already exists. Use **`vmm_scenario_gen::replace_scenario()`** to replace a registered scenario.

Example

Example A-194

```
class atm_cell extends vmm_data;
   . . .
endclass

`vmm_scenario_gen(atm_cell, "atm trans")

program test_scenario;
   atm_cell_scenario_gen atm_gen = new("Atm Scenario Gen",
12);
   atm_cell_scenario parent_scen = new;
   . . .
   initial begin
      . . .
      vmm_log(log,"Registering scenario \n");
     atm_gen.register_scenario("PARENT SCEN", parent_scen);
      . . .
   end

endprogram
```

vmm_scenario_gen::scenario_exists()

Checks if a scenario is registered under a specified name.

SystemVerilog

```
virtual function bit scenario_exists(string name)
```

OpenVera

Not supported.

Description

Returns TRUE if there is a scenario registered under the specified name. Returns FALSE otherwise.

Use **vmm_scenario_gen::get_scenario()** to retrieve a scenario under a specified name.

Example

Example A-195

```
class atm_cell extends vmm_data;
   . . .
endclass

`vmm_scenario_gen(atm_cell, "atm trans")

program test_scenario;
   atm_cell_scenario_gen atm_gen = new("Atm Scenario Gen",
12);
   atm_cell_scenario parent_scen = new;
   . . .
   initial begin
      . . .
```

```
      vmm_log(log,"Registering scenario \n");
     atm_gen.register_scenario("PARENT SCEN", parent_scen);
      . . .
      if(atm_gen.scenario_exists("PARENT SCEN") begin
        vmm_log(log,"Scenario exists and you can use \n");
         . . .
      end
   end

endprogram
```

vmm_scenario_gen::get_scenario()

Returns the scenario registered under a specified name

SystemVerilog

```
virtual function vmm_scenario get_scenario(string name)
```

OpenVera

Not supported.

Description

Returns the scenario descriptor registered under the specified *name*. Issues a warning message and returns `NULL` if there are no scenarios registered under that name.

Example

Example A-196

```
class atm_cell extends vmm_data;
   . . .
endclass

`vmm_scenario_gen(atm_cell, "atm trans")

program test_scenario;
   atm_cell_scenario_gen atm_gen = new("Atm Scenario Gen",
12);
   atm_cell_scenario atm_scenario = new;
   . . .
   initial begin
      . . .
    if(atm_gen.get_scenario("PARENT SCEN") == atm_scenario)
         vmm_log(log,"Scenario matching \n");
```

```
      . . .
   end

endprogram
```

vmm_scenario_gen::get_scenario_name()

Returns the name of the specified scenario.

SystemVerilog

```
virtual function int get_scenario_index(
    vmm_scenario scenario)
```

OpenVera

Not supported.

Description

Returns a name under which the specified scenario descriptor is registered. Returns "" if the scenario is not registered.

Example

Example A-197

```
class atm_cell extends vmm_data;
   . . .
endclass

`vmm_scenario_gen(atm_cell, "atm trans")

program test_scenario;
   atm_cell_scenario_gen atm_gen = new("Atm Scenario Gen",
12);
   atm_cell_scenario atm_scenario = new;
   . . .
   initial begin
      . . .
      scenario_name =
atm_gen.get_scenario_name(atm_scenario);
      vmm_note(log,`vmm_sformatf("Registered name for
```

```
atm_scenario is : %s\n",scenario_name));
      . . .
   end

endprogram
```

vmm_scenario_gen::get_scenario_index()

Returns the index of the specified scenario.

SystemVerilog

```
virtual function int get_scenario_index(
     vmm_scenario scenario)
```

OpenVera

Not supported.

Description

Returns the index of the specified scenario descriptor in the scenario set array. A warning message is issued and returns -1 if the scenario descriptor is not found in the scenario set.

Example

Example A-198

```
class atm_cell extends vmm_data;
   . . .
endclass

`vmm_scenario_gen(atm_cell, "atm trans")

program test_scenario;
   atm_cell_scenario_gen atm_gen = new("Atm Scenario Gen",
12);
   atm_cell_scenario atm_scenario = new;
   . . .
   initial begin
      . . .
     scen_index = atm_gen.get_scenario_index(atm_scenario);
```

```
      if(scen_index == 5)
        `vmm_note(log, `vmm_sformatf("INDEX MATCHED %0d",
index));
      else
         `vmm_error(log,`vmm_sformatf("INDEX NOT MATCHING
%0d", index));
      . . .
   end

endprogram
```

vmm_scenario_gen::get_names_by_scenario()

Returns the names under which a scenario is registered

SystemVerilog

```
virtual function void get_names_by_scenario(
    vmm_scenario scenario,
    ref string    name[$])
```

OpenVera

Not supported.

Description

Appends the names under which the specified scenario descriptor is registered. Returns the number of names that were added to the array.

Example

Example A-199

```
class atm_cell extends vmm_data;
   . . .
endclass

`vmm_scenario_gen(atm_cell, "atm trans")

program test_scenario;
   string scen_names_arr[$];
   atm_cell_scenario_gen atm_gen = new("Atm Scenario Gen",
12);
   atm_cell_scenario atm_scenario = new;
   . . .
   initial begin
```

```
      . . .

atm_gen.get_names_by_scenario(atm_scenario,scen_names_arr)
;
   end

endprogram
```

vmm_scenario_gen::get_all_scenario_names()

Returns all the names in the scenario registry

SystemVerilog

```
virtual function void get_all_scenario_names(
    ref string    name[$])
```

OpenVera

Not supported.

Description

Appends the names under which a scenario descriptor is registered. Returns the number of names that were added to the array.

Example

Example A-200

```
class atm_cell extends vmm_data;
   . . .
endclass

`vmm_scenario_gen(atm_cell, "atm trans")

program test_scenario;
   string scen_names_arr[$];
   atm_cell_scenario_gen atm_gen = new("Atm Scenario Gen",
12);
   atm_cell_scenario atm_scenario = new;
   . . .
   initial begin
      . . .
      atm_gen.get_all_scenario_names(scen_names_arr);
```

```
   end

endprogram
```

vmm_scenario_gen::replace_scenario()

Replace a scenario descriptor

SystemVerilog

```
virtual function void replace_scenario(string name,
    vmm_scenario scenario)
```

OpenVera

Not supported.

Description

Registers the specified scenario under the specified name, replacing the scenario previously registered under that name, if any. The name under which a scenario is registered does not need to be the same as the name of a kind of scenario defined in the scenario descriptor using `vmm_scenario::define_scenario()`. The same scenario may be registered multiple times under different names, therefore, creating an alias to the same scenario.

Registering a scenario implicitly appends it to the scenario set if it is not already in the `vmm_scenario_gen::scenario_set[$]` `array`. The replaced scenario is removed from the scenario set if it is not also registered under another name.

Example

Example A-201

```
`vmm_scenario_gen(atm_cell, "atm trans")

program test_scenario;
```

```
   atm_cell_scenario_gen atm_gen = new("Atm Scenario Gen",
12);
   atm_cell_scenario parent_scen = new;
   . . .
   initial begin
      . . .
     atm_gen.register_scenario("MY SCENARIO", parent_scen);
     atm_gen.register_scenario("PARENT SCEN", parent_scen);
      . . .
      if(atm_gen.scenario_exists("MY SCENARIO") begin
      atm_gen.replace_scenario("MY SCENARIO", parent_scen);
     vmm_log(log,"Scenario exists and has been replaced\n");
         . . .
      end
   end

endprogram
```

vmm_scenario_gen::unregister_scenario()

Unregister a scenario descriptor.

SystemVerilog

```
virtual function bit unregister_scenario(
    vmm_scenario scenario)
```

OpenVera

Not supported.

Description

Completely unregisters the specified scenario descriptor and returns TRUE if it exists in the registry. The unregistered scenario is also removed from the scenario set.

Example

Example A-202

```
`vmm_scenario_gen(atm_cell, "atm trans")

program test_scenario;
   atm_cell_scenario_gen atm_gen = new("Atm Scenario Gen",
12);
   atm_cell_scenario atm_scenario = new;
    . . .
   initial begin
      . . .
      if(atm_gen.unregister_scenario(atm_scenario))
         vmm_log(log,"Scenario has been unregistered \n");
         . . .
      else
         vmm_log(log,"Unable to unregister scenario\n");
```

```
      . . .
   end

endprogram
```

vmm_scenario_gen::unregister_scenario_by_name()

Unregister a scenario descriptor.

SystemVerilog

```
virtual function vmm_scenario unregister_scenario(string
name)
```

OpenVera

Not supported.

Description

Unregisters the scenario under the specified name and returns the unregistered scenario descriptor. Returns `NULL` if there is no scenario registered under the specified name.

The unregistered scenario descriptor is removed from the scenario set if it is not also registered under another name.

Example

Example A-203

```
`vmm_scenario_gen(atm_cell, "atm trans")

program test_scenario;
   atm_cell_scenario_gen atm_gen = new("Atm Scenario Gen",
12);
   atm_cell_scenario atm_scenario = new;
   atm_cell_scenario buffer_scenario = new;

   . . .
   initial begin
      . . .
```

```
      buffer_scenario =
atm_gen.unregister_scenario_by_name("PARENT SCEN");
      if(buffer_scenario != null)
         vmm_log(log,"Scenario has been unregistered \n");
         . . .
      else
         vmm_log(log,"Returned null value\n");
      . . .
   end

endprogram
```

*class-name*_scenario

This class implements a base class for describing scenarios or sequences of transaction descriptors. This class named *class-name*_**scenario** is automatically declared and implemented for any user-specified class named *class-name* by the scenario generator macro, using a process similar to the **'vmm_channel** macro.

Summary

*class-name*_**scenario::log**

Message service interface to be used to issue generic messages.

SystemVerilog

```
static vmm_log log;
```

OpenVera

Not supported.

Description

Message service interface to be used to issue generic messages when the message service interface of the scenario generator is not available or in scope.

*class-name*_**scenario::stream_id**

Stream identifier.

SystemVerilog

```
int stream_id;
```

OpenVera

Not supported.

Description

Stream identifier. It is set by the scenario generator before the scenario descriptor is randomized. Can be used to express stream-specific constraints.

*class-name*_**scenario::scenario_id**

Scenario identifier.

SystemVerilog

```
int scenario_id;
```

OpenVera

Not supported.

Description

Scenario identifier within the stream. It is set by the scenario generator before the scenario descriptor is randomized and incremented after each randomization. Can be used to express scenario-specific constraints. The scenario identifier is reset to 0 when the scenario generator is reset or when the specified number of scenarios has been generated.

*class-name*_scenario::define_scenario()

Define a new scenario.

SystemVerilog

```
function int unsigned
                define_scenario(string name,
    int unsigned max-len);
```

OpenVera

Not supported.

Description

Defines a new scenario with the specified name and the specified maximum number of transactions or data descriptors. Returns a unique scenario identifier that should be assigned to an **`int unsigned`** property.

*class-name*_scenario::redefine_scenario()

Redefine the name and maximum number of descriptors in a scenario.

SystemVerilog

```
function void
   redefine_scenario(int unsigned scenario-kind,
     string name,
     int unsigned max-len);
```

OpenVera

Not supported.

Description

Redefines the name and maximum number of descriptors in a previously defined scenario. Used to redefine an existing scenario instead of creating a new one and constraining the original scenario out of existence.

*class-name*_scenario::scenario_name()

Returns the name associated with the specified scenario identifier.

SystemVerilog

```
function string
            scenario_name(int unsigned scenario-kind);
```

OpenVera

Not supported.

*class-name*_**scenario::scenario-kind**

Select the identifier of the scenario that is generated.

SystemVerilog

```
rand int unsigned scenario-kind;
```

OpenVera

Not supported.

Description

When randomized, selects the identifier of the scenario that is generated. Constrained to the known scenario identifiers defined using the *class-name*_**scenario::define_scenario()** method. Can be constrained to modify the distribution of generated scenarios.

*class-name*_**scenario::length**

Randomized number of items in the scenario.

SystemVerilog

```
rand int unsigned length;
```

OpenVera

Not supported.

Description

Defines how many instances in the *class-name*_**scenario::items[]** property are part of the scenario.

*class-name*_**scenario::items[]**

Instances that are randomized to form the scenarios.

SystemVerilog

```
rand class-name items[];
```

OpenVera

Not supported.

Description

Instances of user-specified *`class-name`* that are randomized to form the scenarios. Only elements from index 0 to *`class-name`*_**`scenario`***`::length`*-1 are part of the scenario.

The constraint blocks and **`rand`** attributes of the instances in the randomized array may be turned **`ON`** or **`OFF`** to modify the constraints on scenario items. They can also be replaced with extensions.

By default, the output stream is formed by copying the values of the items in this array onto the output channel.

*class-name*_**scenario::using**

Instance used in **`pre_randomize()`** when invoking **`fill_scenario()`**.

SystemVerilog

```
class-name using;
```

OpenVera

Not supported.

Description

Instance used in the default implementation of the **`pre_randomize()`** method when invoking the **`fill_scenario()`** method. Set to *`null`* by default. Can be replaced by an instance of a derived class to subject the items of the scenario to different constraints or content.

*class-name*_**scenario::repeated**

Number of times the items in the scenario are repeated.

SystemVerilog

```
rand int unsigned repeated;
```

OpenVera

Not supported.

Description

A value of 0 indicates that the scenario is not repeated, hence is applied only once. The repeated instances in teh scenario count toward the total number of instances generated but only one scenario is considered generated, regardless of the number of times it is repeated.

*class-name*_**scenario::repeat_thresh**

Threshold for the number of times to repeat a scenario.

SystemVerilog

```
static int unsigned repeat_thresh;
```

OpenVera

Not supported.

Description

To avoid accidentally repeating a scenario many times because the **`repeated`** property was left unconstrained, a warning message will be issued if the value of the **`repeated`** property is greater than the value specified in this property. The default value is 100.

*class-name*_scenario::allocate_scenario()

Allocate a new set of instances in the **`items`** property.

SystemVerilog

```
function void
   allocate_scenario(class-name using = null);
```

OpenVera

Not supported.

Description

Allocates a new set of instances in the **`items`** property, up to the maximum number of items in the maximum-length scenario. Any instance previously located in the **`items`** array is replaced. If a reference to an instance is specified in the **`using`** argument, the array is filled by calling **`vmm_data::copy()`** on the specified instance. Otherwise, the array is filled with new instance of *`class-name`* class.

*class-name*_scenario::fill_scenario()

Allocate new instances in the **`items`** property.

SystemVerilog

```
function void fill_scenario(class-name using = null);
```

OpenVera

Not supported.

Description

Allocates new instances in the **`items`** property, up to the maximum number of items in the maximum-length scenario in any *`null`* element of the array. Any instance previously located in the **`items`** array is left untouched. If a reference to an instance is specified in the **`using`** argument, the array is filled by calling **`vmm_data::copy()`** on the specified instance. Otherwise, the array is filled with a new instance of *`class-name`* class.

*class-name*_**scenario::apply()**

Apply the items in the scenario descriptor to an output channel.

SystemVerilog

```
virtual task apply(class-name_channel channel,
    ref int unsigned n-insts);
```

OpenVera

Not supported.

Description

Applies the items in the scenario descriptor to the specified output channel and returns when they have all been consumed by the channel. The *n-insts* argument is set to the number of instances that were consumed by the channel. By default, copies the values of the *items* array using their **vmm_data::copy()** method.

This method may be overloaded to define procedural scenarios.

Example

Example A-204

```
class dut_ms_sequence;
   rand eth_frame_sequence to_phy;
   rand eth_frame_sequence to_mac;
   rand wb_cycle_sequence  to_host;
   ...
   virtual task apply(eth_frame_channel to_phy_chan,
                      eth_frame_channel to_mac_chan,
                      wb_cycle_channel  wb_chan);
      ...
```

```
   endtask
endclass: dut_ms_sequence
```

*class-name*_atomic_scenario

This class implements a predefined atomic scenario descriptor. An atomic scenario is composed of a single unconstrained transaction or data descriptor. This class named *class-name*`_atomic_scenario` is automatically implemented for any user-specified class named *class-name* by the scenario generator macro, using a process similar to the **`'vmm_channel`** macro.

Summary

*class-name*_**atomic_scenario::ATOMIC**

Identifier for the atomic scenario.

SystemVerilog

```
int unsigned ATOMIC;
```

OpenVera

Not supported.

Description

Symbolic scenario identifier for the atomic scenario described by this descriptor. The atomic scenario is a single, random, unconstrained, transaction descriptor (that is, an atomic descriptor).

*class-name*_**atomic_scenario::atomic-scenario**

Constraints of the atomic scenario.

SystemVerilog

```
constraint atomic-scenario;
```

OpenVera

Not supported.

Description

Specifies the constraints of the atomic scenario. By default, the atomic scenario is a single unrepeated unconstrained item. This constraint block may be overridden to redefine the atomic scenario.

*class-name*_scenario_election

This class implements a random selection process for selecting the next scenario descriptor, from a set of available descriptors, to be randomized next. This class named *class-name*_**scenario_election** is automatically implemented for any user-specified class named *class-name* by the scenario generator macros, using a process similar to the **'vmm_channel** macro.

Summary

*class-name*_scenario_election::stream_id

Stream identifier.

SystemVerilog

```
int stream_id;
```

OpenVera

Not supported.

Description

It is set by the scenario generator to the value of the generator stream identifier before the scenario selector is randomized. Can be used to express stream-specific constraints.

*class-name*_scenario_election::scenario_id

Scenario identifier within the stream.

SystemVerilog

```
int scenario_id;
```

OpenVera

Not supported.

Description

It is set by the scenario generator before the scenario selector is randomized and incremented after each randomization. Can be used to express scenario-specific constraints. The scenario identifier is reset to 0 when the scenario generator is reset or when the specified number of scenarios has been generated.

*class-name*_scenario_election::n_scenarios

Number of available scenario descriptors in the scenario set.

SystemVerilog

```
int unsigned n_scenarios;
```

OpenVera

Not supported.

Description

The final value of the *`select`* property must be in the [0:*`n_scenarios`*-1] range.

*class-name*_scenario_election::last_selected[$]

History of the last scenario selections.

SystemVerilog

```
int unsigned last_selected[$];
```

OpenVera

Not supported.

Description

A history (maximum of 10) of the last scenario selections. Can be used to express constraints based on the historical distribution of the selected scenarios (for example, “Never select the same scenario twice in a row.”).

*class-name*_scenario_election::next_in_set

The next scenario in a round-robin selection process.

SystemVerilog

```
int unsigned next_in_set;
```

OpenVera

Not supported.

Description

The next scenario descriptor index that would be selected in a round-robin selection process. Used by the **`round_robin`** constraint block.

*class-name*_scenario_election::scenario_set[$]

The set of scenario descriptors.

SystemVerilog

```
class-name_scenario scenario_set[$];
```

OpenVera

Not supported.

Description

The available set of scenario descriptors. Can be used to procedurally determine which scenario to select or to express constraints based on the scenario descriptors.

*class-name*_scenario_election::select

The index of the selected scenario to be randomized next.

SystemVerilog

```
rand int select;
```

OpenVera

Not supported.

Description

The index, within the **`scenario_set`** array, of the selected scenario descriptor to be randomized next.

*class-name*_scenario_election::round_robin

Constrain the scenario selection process to a round-robin selection.

SystemVerilog

```
constraint round_robin;
```

OpenVera

Not supported.

Description

This constraint block may be turned off to produce a random scenario selection process or allow a different constraint block to define a different scenario selection process.

*class-name*_scenario_gen_callbacks

This class implements a façade for callback containments for the scenario generator transactor. This class named *class-name*_**scenario_gen_callbacks** is automatically implemented for any user-specified class named *class-name* by the scenario generator macro, using a process similar to the **`vmm_channel** macro.

Summary

*class-name*_scenario_gen_callbacks::pre_scenario_randomize()

Callback invoked by the generator after a scenario is selected.

SystemVerilog

```
virtual task pre_scenario_randomize(
     class-name_scenario_gen gen,
     ref class-name_scenario scenario);
```

OpenVera

Not supported.

Description

Callback method invoked by the generator after a new scenario has been selected but before it is randomized. The *gen* argument refers to the generator instance that is invoking the callback method. The *scenario* argument refers to the newly selected scenario descriptor which can be modified. Note that any modifications of the randomization state of the scenario descriptor—such as turning constraint blocks ON or OFF—will remain in effect the next time the scenario descriptor is selected to be randomized. If the reference to the scenario descriptor is set to *null*, the scenario will not be randomized and a new scenario will be selected.

To minimize memory allocation and collection, it is possible that the elements of the scenarios may not be allocated. Use the *class-name*_**scenario::allocate_scenario()** or *class-name*_**scenario::fill_scenario()** to allocate the elements of the scenario if necessary.

*class-name*_scenario_gen_callbacks::post_scenario_gen()

Callback invoked by the generator after a scenario is randomized.

SystemVerilog

```
virtual task post_scenario_gen(
     class-name_scenario_gen gen,
     class-name_scenario scenario,
     ref bit dropped);
```

OpenVera

Not supported.

Description

Callback method invoked by the generator after a new scenario has been randomized but before it is applied to the output channel. The *gen* argument refers to the generator instance that is invoking the callback method. The *scenario* argument refers to the newly randomized scenario that can be modified. Note that any modifications of the randomization state of the scenario descriptor—such as turning constraint blocks ON or OFF—will remain in effect the next time the scenario descriptor is selected to be randomized. If the value of the *dropped* argument is set to non-zero, the generated instance will not be applied to the output channel.

vmm_scheduler

Channels are point-to-point transaction descriptor transfer mechanisms. If multiple sources are adding descriptors to a single channel, the descriptors are interleaved with the descriptors from the other sources in a fair but uncontrollable way. If a multi-point-to-point mechanism is required to follow a specific scheduling algorithm, a **vmm_scheduler** component can be used to identify which source stream should next be forwarded to the output stream.

This class is based on the **vmm_xactor** class.

Summary

vmm_scheduler::log

Message service interface for this scheduler.

SystemVerilog

```
vmm_log log;
```

OpenVera

Not supported.

Description

Set by the constructor and uses the name and instance name specified in the constructor.

vmm_scheduler::out_chan

Reference to the output channel.

SystemVerilog

```
protected vmm_channel out_chan;
```

OpenVera

Not supported.

Description

Set by the constructor.

vmm_scheduler::new()

Create an instance of a channel scheduler.

SystemVerilog

```
function new(string name,
     string instance,
     vmm_channel destination,
     int instance_id = -1);
```

OpenVera

Not supported.

Description

Creates a new instance of a channel scheduler object with the specified name, instance name, destination channel and optional instance identifier.

vmm_scheduler::start_xactor()

Start this **vmm_scheduler** instance.

SystemVerilog

```
virtual function void start_xactor();
```

OpenVera

Not supported.

Description

The scheduler can be stopped. Any extension of this method must call **super.start_xactor()**.

vmm_scheduler::stop_xactor()

Suspend this **`vmm_scheduler`** instance.

SystemVerilog

```
virtual function void stop_xactor();
```

OpenVera

Not supported.

Description

The scheduler can be restarted. Any extension of this method must call **`super.stop_xactor()`**.

vmm_scheduler::reset_xactor()

Reset this **vmm_scheduler** instance.

SystemVerilog

```
virtual function void
           reset_xactor(reset_e rst_typ = SOFT_RST);
```

OpenVera

Not supported.

Description

The output channel and all input channels are flushed. If a **HARD_RST** reset type is specified, the scheduler election factory instance in the **randomized_sched** property is replaced with a new default instance.

vmm_scheduler::new_source()

Add the channel instance to the scheduler.

SystemVerilog

```
virtual function int new_source(vmm_channel chan);
```

OpenVera

Not supported.

Description

Adds the specified channel instance as a new input channel to the scheduler. This method returns an identifier for the input channel that must be used to modify the configuration of the input channel or -1 if an error occurred.

Any user extension of this method must call **`super.new_source()`**.

vmm_scheduler::sched_on

Turns scheduling from the specified input channel on.

SystemVerilog

```
virtual function void sched_on(int unsigned input-id);
```

OpenVera

Not supported.

Description

By default, scheduling from an input channel is on. When scheduling is turned off, the input channel is not flushed and the scheduling of new transaction descriptors from that source channel is inhibited. The scheduling of descriptors from that source channel is resumed as soon as scheduling is turned on.

Any user extension of this method should call **`super.sched_on()`**.

vmm_scheduler::sched_off()

Turns scheduling from the specified input channel off.

SystemVerilog

```
virtual function void sched_off(int unsigned input-id);
```

OpenVera

Not supported.

Description

By default, scheduling from an input channel is on. When scheduling is turned off, the input channel is not flushed and the scheduling of new transaction descriptors from that source channel is inhibited. The scheduling of descriptors from that source channel is resumed as soon as scheduling is turned on.

Any user extension of this method should call **`super.sched_off()`**.

vmm_scheduler::schedule()

Create scheduling components with different rules.

SystemVerilog

```
virtual protected task
                  schedule(output vmm_data obj,
                  input vmm_channel sources[$],
                  int unsigned input-ids[$]);
```

OpenVera

Not supported.

Description

Overloading this method allows the creation of scheduling components with different rules. It is invoked for each scheduling cycle. The transaction descriptor returned by this method in the *obj* argument is added to the output channel. If this method returns *null*, no descriptor is added for this scheduling cycle. The input channels provided in the *sources* argument are all the currently non-empty ON input channels. Their corresponding input identifier is found in the *input-ids* argument.

New scheduling cycles are attempted whenever the output channel is not full. If no transaction descriptor is scheduled from any of the currently non-empty source channels, the next scheduling cycle will be delayed until an additional ON source channel becomes non-empty. If there are no empty input channels and no OFF channels, lock-up will occur.

The default implementation of this method randomizes the instance found in the **`randomized_sched`** property.

vmm_scheduler::get_object()

Extract the next scheduled transaction descriptor.

SystemVerilog

```
virtual protected task get_object(
                     output vmm_data obj,
                     input vmm_channel source,
                     input int unsigned input-id,
                    input int offset);
```

OpenVera

Not supported.

Description

This method is invoked by the default implementation of the **`vmm_scheduler::schedule()`** method to extract the next scheduled transaction descriptor from the specified input channel at the specified offset within the channel. Overloading this method allows access to or replacement of the descriptor that is about to be scheduled. User-defined extensions can be used to introduce errors by modifying the object, interfere with the scheduling algorithm by substituting a different object or recording of the schedule into a functional coverage model.

Any object that is returned by this method via the *obj* argument must either have been internally created or physically removed from the input source using the **`vmm_channel::get()`** method. If a reference to the object remains in the input channel (for example, by using the **`vmm_channel::peek()`** or

`vmm_channel::activate()` method), it is liable to be scheduled more than once as the mere presence of an instance in any of the input channel makes it available to the scheduler.

vmm_scheduler::randomize_sched

Factory instance randomized by the default implementation of the **`vmm_scheduler::schedule()`** method.

SystemVerilog

```
vmm_scheduler_election randomized_sched;
```

OpenVera

Not supported.

Description

Can be replaced with user-defined extensions to modify the election rules.

vmm_scheduler_election

This class implements a round-robin election process by default. In its current form, turning it into a random election process requires that this class be extended. The following modifications will simplify this process: only the **default_round_robin** constraint block needs to be turned off.

The following class properties should be read or added:

- **"vmm_scheduler_election::next_idx"**
- **"vmm_scheduler_election::source_idx"**
- **"vmm_scheduler_election::obj_offset"**

Summary

vmm_scheduler_election::instance_id

Instance identifier of a **`vmm_scheduler`** class instance.

SystemVerilog

```
int instance_id;
```

OpenVera

Not supported.

Description

Instance identifier of the **`vmm_scheduler`** class instance that is randomizing this object instance. Can be used to specified instance-specific constraints.

vmm_scheduler_election::election_id

Incremented by the **`vmm_scheduler`** instance.

SystemVerilog

```
int unsigned election_id;
```

OpenVera

Not supported.

Description

Incremented by the **`vmm_scheduler`** instance that is randomizing this object instance before every election cycle. Can be used to specified election-specific constraints.

vmm_scheduler_election::n_sources

Number of sources.

SystemVerilog

```
int unsigned n_sources;
```

OpenVera

Not supported.

Description

Equal to **`vmm_scheduler_election::sources.size()`**.

vmm_scheduler_election::sources[$]

Input source channels with transaction descriptors available to be scheduled.

SystemVerilog

```
vmm_channel sources[$];
```

OpenVera

Not supported.

vmm_scheduler_election::ids[$]

Input identifiers corresponding to the source channels .

SystemVerilog

```
int unsigned ids[$];
```

OpenVera

Not supported.

Description

Unique input identifiers corresponding to the source channels at the same index in the *sources* array.

vmm_scheduler_election::id_history[$]

A queue of input identifiers.

SystemVerilog

```
int unsigned id_history[$];
```

OpenVera

Not supported.

Description

A queue of the (up to) 10 last input identifiers that were elected.

vmm_scheduler_election::obj_history[$]

A list of transaction descriptors.

SystemVerilog

```
vmm_data obj_history[$];
```

OpenVera

Not supported.

Description

A list of the (up to) 10 last transaction descriptors that were elected.

vmm_scheduler_election::next_idx

Assign to **`source_idx`** for a round-robin process.

SystemVerilog

```
int unsigned next_idx;
```

OpenVera

Not supported.

Description

This is the value to assign to **`source_idx`** to implement a round-robin election process.

vmm_scheduler_election::source_idx

Index in the **`sources`** array of the elected source channel.

SystemVerilog

```
rand int unsigned source_idx;
```

OpenVera

Not supported.

Description

An index of –1 indicates no election. The **`vmm_scheduler_election_valid`** constraint block constrains this property to be in the 0 to **`sources.size()`**-1 range.

vmm_scheduler_election::obj_offset

Offset of the elected transaction descriptor within the elected source channel.

SystemVerilog

```
rand int unsigned obj_offset;
```

OpenVera

Not supported.

Description

Offset, within the source channel indicated by the **`source_idx`** property, of the elected transaction descriptor within the elected source channel. This property is constrained to be 0 in the **`vmm_scheduler_election_valid`** constraint block to preserve ordering of the input streams.

vmm_scheduler_election::default_round_robin

Constraints required by the default round-robin election process.

SystemVerilog

```
constraint default_round_robin;
```

OpenVera

Not supported.

vmm_scheduler_election::post_randomize()

Perform the round-robin election.

SystemVerilog

```
function void post_randomize();
```

OpenVera

Not supported.

Description

The default implementation of this method helps perform the round-robin election.

vmm_subenv

This class is a base class used to encapsulate a reusable sub-environment. The guidelines and techniques covering the usage of this class can be found in the section, “Implementing Sub-environments” on page 3-6.

Summary

vmm_subenv::new()

Create a new instance of this sub-environment base class.

SystemVerilog

```
function new(string name,
     string inst,
     vmm_consensus end_test);
```

OpenVera

```
task new(string name,
     string inst,
     vmm_consensus end_test);
```

Description

Create a new instance of this base class with the specified name and instance name. The specified name and instance names are used as the name and instance names of the log class property.

The specified end-of-test consensus object is assigned to the `end_test` class property and may be used by the sub-environment to indicate that it opposes or consents to the ending of the test.

Example

Example A-205

```
class my_vmm_subenv extends vmm_subenv;
   . . .
   function new(string name,string inst,
           vmm_consensus end_test);
      super.new(name,inst,end_test);
      . . .
   endfunction
```

```
   . . .
endclass
```

vmm_subenv::log

Message service interface for the sub-environment.

SystemVerilog

```
vmm_log log;
```

OpenVera

```
rvm_log log;
```

Description

This property is set by the constructor using the specified name and instance name. These names may be modified afterward using the `vmm_log::set_name()` or `vmm_log::set_instance()` methods.

Example

Example A-206

```
class my_vmm_subenv extends vmm_subenv;
   vmm_log log;
   . . .
   function new(string name,string inst,
          vmm_consensus end_test);
      . . .
      this.log = new("Sub env","Class");
   endfunction
   . . .
endclass
```

vmm_subenv::end_test

End-of-test consensus interface.

SystemVerilog

```
protected vmm_consensus end_test;
```

OpenVera

```
protected vmm_consensus end_test;
```

Description

Local copy of the `vmm_consensus` reference supplied to the constructor. It may be used to indicate if the sub-environment and its components consent to or oppose the ending of the test.

Unless an objection is indicated, the sub-environment will consent by default.

Example

Example A-207

```
class my_vmm_subenv extends vmm_subenv;
   protected vmm_consensus end_test;
   . . .
   function new(string name,string inst,
          vmm_consensus end_test);
      super.new(name,inst,end_test);
      . . .
   endfunction
   . . .
endclass
```

vmm_subenv::configured()

Indicate that the DUT has been configured.

SystemVerilog

```
protected function void configured();
```

OpenVera

```
protected task configured();
```

Description

Report to the base class that the sub-environment and associated DUT have been configured appropriately and that the sub-environment is ready to be started.

This method must be called by a user-defined *`configure()`* method in the extension of this base class.

Example

Example A-208

```
class my_vmm_subenv extends vmm_subenv;
   . . .
   protected function void configured(...);
      // Configuration of sub environment and corresponding
     // portion of DUT
       . . .
      super.configured();
   endfunction
   . . .
endclass
```

vmm_subenv::start()

Start the sub-environment.

SystemVerilog

```
virtual task start();
```

OpenVera

```
virtual task start_t();
```

Description

Start the sub-environment. An error is reported if this method is called before the sub-environment and DUT have been reported as configured to the sub-environment base class using the “vmm_consensus::unregister_voter()” method.

A stopped sub-environment may be restarted.

The base implementation must be called using `super.start()` by any extension of this method in a user-defined extension of this base class.

Example

Example A-209

```
class my_vmm_subenv extends vmm_subenv;
   . . .
   virtual task start()
      super.start();
      this.my_xactor.start_xactor();
      . . .
   endtask
```

```
   . . .
endclass
```

vmm_subenv::stop()

Stop the sub-environment.

SystemVerilog

```
virtual task stop();
```

OpenVera

```
virtual task stop_t();
```

Description

Stop the sub-environment to terminate the test cleanly. An error is issued if the sub-environment has not been previously started.

The base implementation must be called using `super.stop()` by any extension of this method in a user-defined extension of this base class.

Example

Example A-210

```
class my_vmm_subenv extends vmm_subenv;
   . . .
   virtual task stop()
      super.stop();
      this.my_xactor.stop_xactor();
      . . .
   endtask
   . . .
endclass
```

vmm_subenv::cleanup()

Verify end-of-test conditions.

SystemVerilog

```
virtual task cleanup();
```

OpenVera

```
virtual task cleanup_t();
```

Description

Stop the sub-environment (if not already stopped) then verify any end-of-test conditions.

The base implementation must be called using `super.cleanup()` by any extension of this method in a user-defined extension of this base class.

Example

Example A-211

```
class my_vmm_subenv extends vmm_subenv;
   . . .
   virtual task cleanup()
      super.cleanup();
      . . .
   endtask
   . . .
endclass
```

vmm_subenv::report()

Report information collected by the sub-environment.

SystemVerilog

```
virtual function void report();
```

OpenVera

```
virtual task report();
```

Description

Report status, coverage or statistical information collected by the sub-environment, but not pass or fail of the test or sub-environment.

This method needs to be extended. It may also be invoked multiple times during the simulation.

Example

Example A-212

```
class my_vmm_subenv extends vmm_subenv;
   . . .
   virtual function void report()
      super.report();
      . . .
   endfunction
   . . .
endclass
```

‘vmm_subenv_member_begin()

Start of shorthand section.

SystemVerilog

```
‘vmm_subenv_member_begin(class-name)
```

OpenVera

Not supported.

Description

Start the shorthand section providing a default implementation for the `psdisplay()`, `start()` and `stop()` methods.

The class-name specified must be the name of the `vmm_subenv` extension class that is being implemented.

The shorthand section can only contain shorthand macros and must be terminated by a “‘vmm_subenv_member_end()” .

Example

Example A-213

```
class tcpip_stack extends vmm_subenv;
   ...
   ‘vmm_subenv_member_begin(tcpip_stack)
      ...
   ‘vmm_subenv_member_end(tcpip_stack)
   ...
endclass
```

‘vmm_subenv_member_end()

End of shorthand section.

SystemVerilog

```
‘vmm_subenv_member_end(class-name)
```

OpenVera

Not supported.

Description

Terminate the shorthand section providing a default implementation for the `psdisplay()`, `start()` and `stop()` methods.

The class-name specified must be the name of the `vmm_subenv` extension class that is being implemented.

The shorthand section must have been started by a “‘vmm_subenv_member_begin()” .

Example

Example A-214

```
class my_vmm_subenv extends vmm_subenv;
   . . .
   `vmm_subenv_member_begin(my_vmm_subenv)
      . . .
   `vmm_subenv_member_end(my_vmm_subenv)
   . . .
endclass
```

‘vmm_subenv_member_scalar*()

Shorthand implementation for a scalar data member.

SystemVerilog

```
‘vmm_subenv_member_scalar(member-name,
                          vmm_subenv::do_what_e do_what)

‘vmm_subenv_member_scalar_array(member-name,
                                vmm_subenv::do_what_e do_what)

‘vmm_subenv_member_scalar_aa_scalar(member-name,
                                vmm_subenv::do_what_e do_what)

‘vmm_subenv_member_scalar_aa_string(member-name,
                                vmm_subenv::do_what_e do_what)
```

OpenVera

Not supported.

Description

Add the specified scalar-type, array of scalars, scalar-indexed associative array of scalars or string-indexed associative array of scalars data member to the default implementation of the methods specified by the `’do_what’` argument.

A scalar is an integral type, such as bit, bit vector, and packed unions.

The shorthand implementation must be located in a section started by a “`‘vmm_subenv_member_begin()`” .

Example

Example A-215

```
class my_vmm_subenv extends vmm_subenv;
   bit [31:0] address;
   . . .
   `vmm_subenv_member_begin(my_vmm_subenv)
      `vmm_subenv_member_scalar(address,DO_ALL)
       . . .
   `vmm_subenv_member_end(my_vmm_subenv)
   . . .
endclass
```

‘vmm_subenv_member_string*()

Shorthand implementation for a string data member.

SystemVerilog

```
‘vmm_subenv_member_string(member-name,
                          vmm_subenv::do_what_e do_what)

‘vmm_subenv_member_string_array(member-name,
                                vmm_subenv::do_what_e do_what)

‘vmm_subenv_member_string_aa_scalar(member-name,
                                vmm_subenv::do_what_e do_what)

‘vmm_subenv_member_string_aa_string(member-name,
                                vmm_subenv::do_what_e do_what)
```

OpenVera

Not supported.

Description

Add the specified string-type, array of strings, scalar-indexed associative array of strings or string-indexed associative array of strings data member to the default implementation of the methods specified by the `’do_what’` argument.

The shorthand implementation must be located in a section started by a “‘vmm_subenv_member_begin()” .

Example

Example A-216

```
class my_vmm_subenv extends vmm_subenv;
```

```
    string xactor_name;
    . . .
    `vmm_subenv_member_begin(my_vmm_subenv)
       `vmm_subenv_member_string(xactor_name,DO_ALL)
        . . .
    `vmm_subenv_member_end(my_vmm_subenv)
    . . .
endclass
```

‘vmm_subenv_member_enum*()

Shorthand implementation for an enumerated data member.

SystemVerilog

```
‘vmm_subenv_member_enum(member-name,
                        vmm_subenv::do_what_e do_what)

‘vmm_subenv_member_enum_array(member-name,
                              vmm_subenv::do_what_e do_what)

‘vmm_subenv_member_enum_aa_scalar(member-name,
                              vmm_subenv::do_what_e do_what)

‘vmm_subenv_member_enum_aa_string(member-name,
                              vmm_subenv::do_what_e do_what)
```

OpenVera

Not supported.

Description

Add the specified enum-type, array of enums, scalar-indexed associative array of enums or string-indexed associative array of enums data member to the default implementation of the methods specified by the `’do_what’` argument.

The shorthand implementation must be located in a section started by a “‘vmm_subenv_member_begin()” .

Example

Example A-217

```
typedef enum {blue,green,red,black} my_colors;
```

```
class my_vmm_subenv extends vmm_subenv;
   my_colors  color;
   . . .
   `vmm_subenv_member_begin(my_vmm_subenv)
      `vmm_subenv_member_enum(color,DO_ALL)
       . . .
   `vmm_subenv_member_end(my_vmm_subenv)
   . . .
endclass
```

‘vmm_subenv_member_vmm_data*()

Shorthand implementation for a vmm_data-based data member.

SystemVerilog

```
‘vmm_subenv_member_vmm_data(member-name,
                           vmm_subenv::do_what_e do_what)

‘vmm_subenv_member_vmm_data_array(member-name,
                           vmm_subenv::do_what_e do_what)

‘vmm_subenv_member_vmm_data_aa_scalar(member-name,
                           vmm_subenv::do_what_e do_what)

‘vmm_subenv_member_vmm_data_aa_string(member-name,
                           vmm_subenv::do_what_e do_what)
```

OpenVera

Not supported.

Description

Add the specified vmm_data-type, array of vmm_datas, scalar-indexed associative array of vmm_datas or string-indexed associative array of vmm_datas data member to the default implementation of the methods specified by the `’do_what’` argument.

The shorthand implementation must be located in a section started by a “‘vmm_subenv_member_begin()” .

Example

Example A-218

```
class my_data extends vmm_data;
   . . .
endclass

class my_vmm_subenv extends vmm_subenv;
   my_data   subenv_data;
   . . .
   `vmm_subenv_member_begin(my_vmm_subenv)
      `vmm_subenv_member_vmm_data(subenv_data,DO_ALL)
      . . .
   `vmm_subenv_member_end(my_vmm_subenv)
   . . .
endclass
```

‘vmm_subenv_member_channel*()

Shorthand implementation for a channel data member.

SystemVerilog

```
‘vmm_subenv_member_channel(member-name,
                           vmm_subenv::do_what_e do_what)

‘vmm_subenv_member_channel_array(member-name,
                            vmm_subenv::do_what_e do_what)

‘vmm_subenv_member_channel_aa_scalar(member-name,
                            vmm_subenv::do_what_e do_what)

‘vmm_subenv_member_channel_aa_string(member-name,
                            vmm_subenv::do_what_e do_what)
```

OpenVera

Not supported.

Description

Add the specified channel-type, array of channels, dynamic array of channels, scalar-indexed associative array of channels or string-indexed associative array of channels data member to the default implementation of the methods specified by the `’do_what’` argument.

The shorthand implementation must be located in a section started by a “`‘vmm_subenv_member_begin()`” .

Example

Example A-219

```
class my_vmm_subenv extends vmm_subenv;
   data_channel subenv_channel;
   . . .
   `vmm_subenv_member_begin(my_vmm_subenv)
      `vmm_subenv_member_channel(subenv_channel,DO_ALL)
      . . .
   `vmm_subenv_member_end(my_vmm_subenv)
   . . .
endclass
```

‘vmm_subenv_member_xactor*()

Shorthand implementation for a transactor data member.

SystemVerilog

```
‘vmm_subenv_member_xactor(member-name,
                          vmm_subenv::do_what_e do_what)

‘vmm_subenv_member_xactor_array(member-name,
                            vmm_subenv::do_what_e do_what)

‘vmm_subenv_member_xactor_aa_scalar(member-name,
                            vmm_subenv::do_what_e do_what)

‘vmm_subenv_member_xactor_aa_string(member-name,
                            vmm_subenv::do_what_e do_what)
```

OpenVera

Not supported.

Description

Add the specified transactor-type, array of transactors, dynamic array of transactors, scalar-indexed associative array of transactors or string-indexed associative array of transactors data member to the default implementation of the methods specified by the `’do_what’` argument.

The shorthand implementation must be located in a section started by a “‘vmm_subenv_member_begin()” .

Example

Example A-220

```
class my_vmm_subenv extends vmm_subenv;
   data_gen subenv_xactor;
   . . .
   `vmm_subenv_member_begin(my_vmm_subenv)
      `vmm_subenv_member_xactor(subenv_xactor,DO_ALL)
      . . .
   `vmm_subenv_member_end(my_vmm_subenv)
   . . .
endclass
```

‘vmm_subenv_member_subenv*()

Shorthand implementation for a transactor data member.

SystemVerilog

```
‘vmm_subenv_member_subenv(member-name,
                          vmm_subenv::do_what_e do_what)

‘vmm_subenv_member_subenv_array(member-name,
                            vmm_subenv::do_what_e do_what)

‘vmm_subenv_member_subenv_aa_scalar(member-name,
                            vmm_subenv::do_what_e do_what)

‘vmm_subenv_member_subenv_aa_string(member-name,
                            vmm_subenv::do_what_e do_what)
```

OpenVera

Not supported.

Description

Add the specified sub-environment-type, array of sub-environments, dynamic array of sub-environments, scalar-indexed associative array of sub-environments or string-indexed associative array of sub-environments data member to the default implementation of the methods specified by the `’do_what’` argument.

The shorthand implementation must be located in a section started by a “`‘vmm_subenv_member_begin()`” .

Example

Example A-221

```
class sub_subenv extends vmm_subenv;
   function new(....);
      super.new(...);
      . . .
   endfunction
endclass

class my_vmm_subenv extends vmm_subenv;
   sub_subenv sub_subenv_inst;
   . . .
   `vmm_subenv_member_begin(my_vmm_subenv)
      `vmm_subenv_member_subenv(sub_subenv_inst,DO_ALL)
      . . .
   `vmm_subenv_member_end(my_vmm_subenv)
   . . .
endclass
```

‘vmm_subenv_member_user_defined()

User-defined shorthand implementation data member.

SystemVerilog

```
‘vmm_subenv_member_user_defined(member-name)
```

OpenVera

Not supported.

Description

Add the specified user-defined default implementation of the methods specified by the `’do_what’` argument.

Refer to “User-defined vmm_env or vmm_subenv Member Default Implementation” on page 2-9 for details on how to specify the shorthand implementation for a data member.

The shorthand implementation must be located in a section started by a `“‘vmm_subenv_member_begin()”` .

Example

Example A-222

```
class my_vmm_subenv extends vmm_subenv;
   bit [7:0] subenv_id;
   . . .
   `vmm_env_member_begin(my_vmm_subenv)
      `vmm_subenv_member_user_defined(subenv_id)
      . . .
   `vmm_env_member_end(my_vmm_subenv)
```

```
   function bit do_subenv_id(vmm_subenv::do_what_e do_what)
      do_subenv_id = 1;
      case(do_what)
      . . .
   endfunction
endclass
```

vmm_subenv::do_what_e

Specifies which methods are to be provided by a shorthand implementation.

SystemVerilog

```
enum {DO_PRINT, DO_START, DO_STOP,
      DO_VOTE, DO_ALL} do_what_e;
```

OpenVera

Not supported.

Description

Used to specify which methods are to include the specified data members in their default implementation. `"DO_PRINT"` includes the member in the default implementation of the psdisplay() method. `"DO_START"` includes the member in the default implementation of the start() method, if applicable. `"DO_STOP"` includes the member in the default implementation of the stop() method, if applicable. `"DO_VOTE"` automatically registers the member with the `vmm_subenv::end_test` consensus instance, if applicable.

Multiple methods can be specified by adding or or'ing the individual symbolic values. All methods are specified by specifying the `"DO_ALL"` symbol.

Example

Example A-223

```
‘vmm_subenv_member_subenv(idler, DO_ALL - DO_STOP);
```

vmm_subenv::do_psdisplay()

Override the shorthand `psdisplay()` method.

SystemVerilog

```
virtual function string do_psdisplay(string prefix = "")
```

OpenVera

Not supported.

Description

This method overrides the default implementation of the `vmm_subenv::psdisplay()` method created by the `vmm_subenv` shorthand macros. If defined, it will be used instead of the default implementation.

Example

Example A-224

```
class my_vmm_subenv extends vmm_subenv;
   . . .
  virtual function string do_psdisplay(string prefix = "");
      $sformat(do_psdisplay,"%s Printing sub environment
          members \n",prefix);
      . . .
   endfunction
   . . .
endclass
```

vmm_env::do_vote()

Override the shorthand voter registration.

SystemVerilog

```
protected virtual task do_vote()
```

OpenVera

Not supported.

Description

This method overrides the default implementation of the voter registration created by the `vmm_subenv` shorthand macros. If defined, it will be used instead of the default implementation.

Example

Example A-225

```
class my_vmm_subenv extends vmm_subenv;
   . . .
   protected virtual task do_vote();
      //Register with this.end_vote
      . . .
   endtask
   . . .
endclass
```

vmm_subenv::do_start()

Override the shorthand `start()` method.

SystemVerilog

```
protected virtual task do_start()
```

OpenVera

Not supported.

Description

This method overrides the default implementation of the `vmm_subenv::start()` method created by the `vmm_subenv` shorthand macros. If defined, it will be used instead of the default implementation.

Example

Example A-226

```
class my_vmm_subenv extends vmm_subenv;
   . . .
   protected virtual task do_start();
      //vmm_subenv::start() operations
      . . .
   endtask
   . . .
endclass
```

vmm_subenv::do_stop()

Override the shorthand `stop()` method.

SystemVerilog

```
protected virtual task do_stop()
```

OpenVera

Not supported.

Description

This method overrides the default implementation of the `vmm_subenv::stop()` method created by the `vmm_subenv` shorthand macros. If defined, it will be used instead of the default implementation.

Example

Example A-227

```
class my_vmm_subenv extends vmm_subenv;
   . . .
   protected virtual task do_stop();
      //vmm_subenv::stop() operations
      . . .
   endtask
   . . .
endclass
```

vmm_test

VMM provides this class to register all possible tests in a container. Hence, one single compilation/elaboration step is enough to deal with multiple tests. Specifying a particular test is done during simulation using VMM Runtime Options.

The single compilation/elaboration step significantly improves regression time by enabling the “compile once/run many times” model. This class also can take advantage of the VCS separate compile facility.

This base class may be used to implement testcases. It enables runtime selection of the testcase to run on an environment.

Summary

Using vmt_test

VMM comes with a new base class name **vmm_test** that allows embedding all tests in a single class. The main purpose of this base class is to leverage from a single compile-elaboration-simulate step rather than multiple steps. The VMM recommendation is to gather tests in a program block, since this provides a good way of encapsulating a testbench. Drawbacks of this technique are that one test should reside in one program block and that you need to recompile, elaborate and simulate on a test basis. When dealing with

large regressions consisting of hundreds of tests, multiple elaborations can waste a significant amount of time whereas **vmm_test** only requires one elaboration. A given test can be picked up at runtime using a specific option. Switches like **+ntb_random_seed** can be used in conjunction with these tests.

To understand better how this base class works, consider the example that is provided with the VMM 1.1 release available in the sv/examples/std_lib/vmm_test directory. This example shows how to constraint some ALU transactions. These transactions are based on an **alu_data** extended object and are randomly generated by **vmm_atomic_gen** and passed to an ALU driver using **vmm_channels**.

Before examining the details of **vmm_test**, consider how tests are traditionally written:

```
1. class add_test_data extends alu_data;
2.    constraint cst_test {
3.     kind == ADD;
4.  }
5. endclass
6. program alu_test();
7.    alu_env env;
8.    initial begin
9.       add_test_data tdata = new;
10.      env.build();
11.      env.gen.randomized_obj = tdata;
12.      env.run();
13.   end
```

```
14.endprogram
```

- In lines 1-4, the **`alu_data`** object is extended to the new transactions **`add_test_data`**, which contains test-specific constraints. In this case only ADD operations are carried forward by this transaction.
- In line 6, a **`program`** block is used to instantiate the environment **`alu_test`** based on **`vmm_env`**.
- In lines 9-12, the environment is built and the new transaction **`add_test_data`** is used as the blueprint for the **`vmm_atomic_gen`** transactor.

This test is very specific and you clearly need to duplicate a similar program block with other constraints to fulfill his test plan. For example, this test can be derived many times to send `{MUL, SUB, DIV}` ALU operations to the ALU driver. In this case, multiple program blocks are required so as multiple elaborations and binaries to simulate these tests.

VMM provides a way to model a generic program block and include all test files just a single time. The previous test can now be written as follows:

```
1. class add_test_data extends alu_data;
2.    constraint cst_test {
3.       kind == ADD;
4.    }
5. endclass
6. `vmm_test_begin(test_add, alu_env, "Addition")
7.    env.build();
8.    begin
```

```
9.        add_test_data tdata = new;
10.        env.gen.randomized_obj = tdata;
11.   end
12.   env.run();
13.`vmm_test_end(test_add)
```

In line 6, the **`vmm_test`** shorthand macro **`vmm_test_begin`** is used to declare the test name (`test_add` in the example above), the name of the **`vmm_env`** where all transactors reside (a `alu_env` in the example) and a label that is used to tag this particular test.

In lines 7-12, we build the environment, insert the blueprint and kick off the test.

In line 13, the **`vmm_test`** shorthand macro **`vmm_test_end`** is used to terminate this test declaration.

Other tests can be written in the same way with variations in constraints. Since the environment is exposed after the **`vmm_test_begin`** shorthand macro, it is possible to register callbacks, replace generators or do any other kind of operations that are traditionally done in the VMM program block.

An important aspect of these tests is that whenever they are used s includes, they become statically declared and visible to the environment.

The following example shows how to include these tests in VMM program block:

```
1. `include "test_add.sv"
2. `include "test_sub.sv"
```

```
3. `include "test_mul.sv"
4. `include "test_ls.sv"
5. `include "test_rs.sv"
6. program alu_test();
7.    alu_env env;
8.    initial begin
9.       vmm_test_registry registry = new;
10.      env = new(alu_drv_port, alu_mon_port);
11.      registry.run(env);
12.   end
13.endprogram
```

In lines 1-5, all tests are simply included.

In line 9, **`registry`** which is an instance of **`vmm_test_registry`**, is constructed. This object contains all tests based on **`vmm_test_begin`** that have been previously included.

In line 11, **`registry`** is run and a handle to the environment is passed as an argument to this class. This is how all **`vmm_test`** can access the environment.

Running these tests is done by providing the test name in the command line, as follows:

```
simv +vmm_test=test_add
simv +vmm_test=test_sub
```

Note that calling **`simv`** without **`+vmm_test`** switch returns a FATAL error and lists all registered tests. This is a good way to document tests and easily retrieve existing tests.

vmm_test::log

Message service interface for the testcase.

SystemVerilog

```
vmm_log log;
```

OpenVera

Not supported.

Description

Message service interface instance that can be used to issue messages in the `vmm_test::run()` method.

The name of the message service interface is "Testcase" and the instance name is the name specified to the `vmm_test::new()` method.

Example

Example A-228

```
program test;
   class test_100 extends vmm_test;
      vmm_env env;
      function new();
         super.new("test_100", "Single Read");
      endfunction
      task run(vmm_env env1);
         `vmm_note(log,"Test Started");
         $cast(env, env1);
      endtask
   endclass
```

```
   initial begin
      test_100 T;
      T = new;
      T.run(T.env);
   end
endprogram
```

vmm_test::new()

Create a test.

SystemVerilog

```
function new(string name, string doc = "");
```

OpenVera

Not supported.

Description

Create an instance of the testcase, its message service interface and registers it in the global testcase registry under the specified name. A short description of the testcase may also be specified.

Example

Example A-229

```
class my_test extends vmm_test;
   function new();
      super.new("my_test");
   endfunction
   static my_test this_test = new();
   virtual task run(vmm_env env);
      ...
   endtask
endclass
```

vmm_test::get_name()

Returns the name of a test.

SystemVerilog

```
function string get_name();
```

OpenVera

Not supported.

Description

Return the name of the test that was specify in the constructor.

Example

Example A-230

```
class my_test extends vmm_test;
   function new();
      super.new("my_test");
   endfunction
   static my_test this_test = new();
   virtual task run(vmm_env env);
      `vmm_note(this.log, {"Running test ",
                           this.get_name()});
      ...
   endtask
endclass
```

vmm_test::get_doc()

Returns the description of a test.

SystemVerilog

```
function string get_doc();
```

OpenVera

Not supported.

Description

Return the short description of the test that was specify in the constructor.

Example

Example A-231

```
class my_test extends vmm_test;
   function new();
      super.new("my_test");
   endfunction
   static my_test this_test = new();
   virtual task run(vmm_env env);
      ‘vmm_note(this.log, {"Running test ",
                           this.get_name()});
      ...
   endtask
endclass
```

vmm_test::run()

Run a test.

SystemVerilog

```
virtual task run(vmm_env env);
```

OpenVera

Not supported.

Description

The test itself.

The default implementation of this method calls `env.run()`. If a different test implementation is required, the default implementation of this method must not be invoked using `super.run()`.

This method should not call `vmm_log::report()`.

Example

Example A-232

```
class my_test extends vmm_test;
   ...
   virtual task run(vmm_env env);
      tb_env my_env;
      $cast(my_env, env);
      my_env.build();
      my_env.gen[0].start_xactor();
      my_env.run();
   endtask
endclass
```

‘vmm_test_begin()

Shorthand macro to define a testcase class.

SystemVerilog

```
‘vmm_test_begin(testclassname, envclassname, doc string)
```

OpenVera

Not supported.

Description

Shorthand macro that may be used to define a user-defined testcase implemented using a class based on the `vmm_test` class. The first argument is the name of the testcase class and will also be used as the name of the testcase in the global testcase registry. The second argument is the name of the environment class that will be used to execute the testcase. A data member of that type named `"env"` will be defined and assigned, ready to be used. The third argument is a string documenting the purpose of the test.

This macro can be used to create the testcase class up to and including the declaration of the `vmm_test::run()` method. This macro can then be followed by variable declarations and procedural statements. The instance of the verification environment of the specified type can be accessed as `"this.env"`. It must be preceded by any `import` statement required by the test implementation.

Example

This example shows how the testcase from Example A-229 and Example A-232 can be implemented using shorthand macros.

Example A-233

```
import tb_env_pkg::*;

‘vmm_test_begin(my_test, tb_env, "Simple test")
      this.env.build();
      this.env.gen[0].stop_xactor();
      this.env.run();
‘vmm_test_end(my_test)
```

‘vmm_test_end()

Shorthand macro to define a testcase class.

SystemVerilog

```
‘vmm_test_end(testclassname)
```

OpenVera

Not supported.

Description

Shorthand macro that may be used to define a user-defined testcase implemented using a class based on the `vmm_test` class. The first argument must be the same name specified as the first argument of the `‘vmm_test_begin()` macro.

This macro can be used to end the testcase class, including the implementation of the `vmm_test::run()` method.

Example

This example shows how the testcase from Example A-229 and Example A-232 can be implemented using shorthand macros.

Example A-234

```
‘vmm_test_begin(my_test, tb_env)
      this.env.build();
      this.env.gen[0].stop_xactor();
      this.env.run();
‘vmm_test_end(my_test)
```

vmm_test_registry

Global test registry that can be optionally used to implement runtime selection of tests.

No constructor is documented because this class is implemented using a singleton pattern. Its functionality is accessed strictly through static members.

Summary

vmm_test_registry::list()

List all available tests.

SystemVerilog

```
static function void list();
```

OpenVera

Not supported.

Description

List of the tests registered with the global test registry.

This method is invoked automatically by the `vmm_test_registry::run()` method, followed by a call to `$finish()`, if the `+vmm_test_help` option is specified.

Example

Example A-235

```
program test;
   `include "test.lst"
   i2c_env env;

   initial begin
      vmm_test_registry registry = new;
      env = new;
      registry.list();
      registry.run(env);
   end
endprogram
```

vmm_test_registry::run()

Run a test.

SystemVerilog

```
static task run(vmm_env env);
```

OpenVera

Not supported.

Description

Run a testcase on the specified verification environment. Using SystemVerilog, this method must be invoked in a *program* thread to satisfy *Verification Methodology Manual* rules.

If more than one testcase has been registered, the name of a testcase must be specified using the `"+vmm_test"` runtime string option. See `vmm_opts::get_string()` for a description of how to specify runtime string options. If only one test has been registered, it is run by default without having to specify its name at runtime.

A default testcase, named `"Default"`, that simply invokes `env::run()`, is automatically always available if no testcase has been previously registered under that name.

Example

Example A-236

```
program top;
   tb_env env = new();
```

```
   initial vmm_test_registry::run(env);
endprogram
```

vmm_version

This class is used to report the version and vendor of the VMM Standard Library implementation.

Summary

vmm_version::major()

Return the major revision number.

SystemVerilog

```
function int major();
```

OpenVera

Not supported.

Description

Return the major version number of the implemented VMM Standard Library. Should always return 1.

vmm_version::minor()

Return the minor revision number.

SystemVerilog

```
function int minor();
```

OpenVera

```
function integer minor();
```

Description

Return the minor version number of the implemented VMM Standard Library. Should always return 5 if the additions and updates specified in this appendix are fully implemented.

Example

Example A-237

```
initial begin
   string minor_ver;
   vmm_version v = new;
   $sformat(minor_ver,"VMM Minor Version %d", v.minor());
   `vmm_note(log,minor_ver);
end
```

vmm_version::patch()

Return the patch number.

SystemVerilog

```
function int patch();
```

OpenVera

Not supported.

Description

Return the patch number of the implemented VMM Standard Library. The return value is vendor-dependent.

vmm_version::vendor()

Return the name of the library vendor.

SystemVerilog

```
function int major();
```

OpenVera

Not supported.

Description

Return the name of the vendor supplying the VMM Standard Library implementation. The return value is vendor-dependent.

vmm_version::display()

Display the version.

SystemVerilog

```
function void display(string, "");
```

OpenVera

Not supported.

Description

Display the version image returned by the **`psdisplay()`** method to the standard output.

vmm_version::psdisplay()

Format the major and minor version, patch, and vendor information.

SystemVerilog

```
function string psdisplay(string prefix = "");
```

OpenVera

Not supported.

Description

Create a well formatted image of the VMM Standard Library implementation version information. The format is:

prefix VMM Version *major*.*minor*.*patch* (*vendor*)

vmm_voter

This class is an interface to participate in a consensus and indicate consent or opposition to the end of test. It is created through the "vmm_consensus::register_voter()" method. Its constructor is not documented, therefore, it must not be created directly.

Summary

vmm_voter::oppose()

Oppose to a consensus.

SystemVerilog

```
function void oppose(string why = "No specified reason");
```

OpenVera

```
task oppose(string why = "No specified reason");
```

Description

Prevents consensus from being reached for the optionally specified reason. This is the default. This method may be called repeatedly to modify the reason for the opposition.

Example

Example A-238

```
initial begin
   my_env env = new();
   vmm_voter test_voter = env.end_vote.register_voter(
      "Test case Stimulus");
   test_voter.oppose("test not done");
end
```

vmm_voter::consent()

Agree to a consensus.

SystemVerilog

```
function void consent(string why = "No specified reason");
```

OpenVera

```
task consent(string why = "No specified reason");
```

Description

Allow consensus to be reached for the optionally specified reason. This method may be called repeatedly to modify the reason for the consent. A consent may be withdrawn by calling the "vmm_voter::oppose()" method.

Example

Example A-239

```
program test_consensus;

   string who[];
   string why[];
   vmm_consensus vote = new("Vote", "Main");
   vmm_voter v1;

   initial begin
      v1 = vote.register_voter("Voter #1");
      v1.consent("Consent by default");
      . . .
   end

endprogram
```

vmm_voter::forced()

Force a consensus.

SystemVerilog

```
function void forced(string why = "No specified reason");
```

OpenVera

```
task forced(string why = "No specified reason");
```

Description

Force an end of test consensus for the optionally specified reason. The end of test is usually forced by a directed testcase, but can be forced by any participant, as necessary. A forced consensus may be cancelled (if the simulation is still running) by calling the "vmm_voter::oppose()" or "vmm_voter::consent()" method.

Example

Example A-240

```
initial begin
   . . .
   vmm_voter test_voter = env.end_vote.register_voter(
     "Test case Stimulus");
   test_voter.oppose("Test not done");
   . . .
   test_voter.forced("Test is done");
end
```

vmm_xactor

This base class is to be used as the basis for all transactors, including bus-functional models, monitors and generators. It provides a standard control mechanism expected in all transactors.

Summary

vmm_xactor::new()

Create an instance of the transactor base class.

SystemVerilog

```
function new(string name,
     string instance,
     int stream_id = -1);
```

OpenVera

Not supported.

Description

Creates an instance of the transactor base class, with the specified name, instance name and optional stream identifier. The name and instance name are used to create the message service interface in the **`vmm_xactor::log`** property and the specified stream identifier is used to initialize the **`vmm_xactor::stream_id`** property.

vmm_xactor::get_name()

Return the name of this transactor.

SystemVerilog

```
virtual function string get_name();
```

OpenVera

Not supported.

vmm_xactor::get_instance()

Return the instance name of this transactor.

SystemVerilog

```
virtual function string get_instance();
```

OpenVera

Not supported.

vmm_xactor::log

Message service interface for messages issued from within this transactor instance.

SystemVerilog

```
vmm_log log;
```

OpenVera

Not supported.

vmm_xactor::stream_id

Identifier for the stream of transaction and data descriptors.

SystemVerilog

```
int stream_id;
```

OpenVera

Not supported.

Description

The **`stream_id`** is a unique identifier for the stream of transaction and data descriptors flowing through this transactor instance. It should be used to set the **`vmm_data::stream_id`** property of the descriptors as they are received or randomized by this transactor.

Example

Example A-241

```
class responder extends vmm_xactor;
   ...
   virtual task main();
      ...
      forever begin
         this.req_chan.get(tr);
         ...
         tr.stream_id = this.stream_id;
         tr.data_id   = response_id++;
         if (!tr.randomize()) ...
         ...
         this.resp_chan.sneak(tr);
      end
   endtask: main
endclass: responder
```

vmm_xactor::prepend_callback()

Prepend the specified callback façade instance with this instance of the transactor.

SystemVerilog

```
virtual function void
           prepend_callback(vmm_xactor_callbacks cb);
```

OpenVera

Not supported.

Description

Callback methods will be invoked in the order in which they were registered.

A warning is issued if the same callback façade instance is registered more than once with the same transactor. A façade instance can be registered with more than one transactor. Callback façade instances can be unregistered and re-registered dynamically.

Example

Example A-242

```
program test;
initial begin
   dut_env env = new;
   align_tx cb = new(...);
   env.build();
   foreach (env.mii[i]) begin
      env.mii[i].prepend_callback(cb);
   end
```

```
   env.run();
end
endprogram
```

vmm_xactor::append_callback()

Appends the specified callback façade instance with this instance of the transactor.

SystemVerilog

```
virtual function void
           append_callback(vmm_xactor_callbacks cb);
```

OpenVera

Not supported.

Description

Callback methods will be invoked in the order in which they were registered.

A warning is issued if the same callback façade instance is registered more than once with the same transactor. A façade instance can be registered with more than one transactor. Callback façade instances can be unregistered and re-registered dynamically.

vmm_xactor::unregister_callback()

Unregister the specified callback façade instance.

SystemVerilog

```
virtual function void
          unregister_callback(vmm_xactor_callbacks cb);
```

OpenVera

Not supported.

Description

Unregisters the specified callback façade instance for this transactor instance. A warning is issued if the specified façade instance is not currently registered with the transactor. Callback façade instances can later be re-registered with the same or another transactor.

vmm_xactor::notify

Notification service interface and pre-configure notifications.

SystemVerilog

```
vmm_notify notify;
enum {XACTOR_IDLE;
     XACTOR_BUSY;
     XACTOR_STARTED;
     XACTOR_STOPPED;
     XACTOR_RESET};
```

OpenVera

Not supported.

Description

Notification service interface and pre-configures notifications to indicate the state and state transitions of the transactor. The **`vmm_xactor::XACTOR_IDLE`** and **`vmm_xactor::XACTOR_BUSY`** notifications are **`vmm_notify::ON_OFF`**. All other events are **`vmm_notify::ONE_SHOT.`**

Example

Example A-243

```
class consumer extends vmm_xactor;
   ...
   virtual task main();
      ...
      forever begin
         transaction tr;
         this.in_chan.peek(tr);
         tr.notify.indicate(vmm_data::STARTED);
         ...
```

```
          tr.notify.indicate(vmm_data::ENDED, ...);
          this.in_chan.get(tr);
       end
   endtask: main
endclass: consumer
```

vmm_xactor::start_xactor()

Start the execution threads in this transactor instance.

SystemVerilog

```
virtual function void start_xactor();
```

OpenVera

Not supported.

Description

Starts the execution threads in this transactor instance. The transactor can later be stopped. Any extension of this method must call **`super.start_xactor()`**. The base class indicates the **`vmm_xactor::XACTOR_STARTED`** and **`vmm_xactor::XACTOR_BUSY`** notifications and resets the **`vmm_xactor::XACTOR_IDLE`** notification.

Example

Example A-244

```
class tb_env extends vmm_env;
   ...
   virtual task start();
      super.start();
      ...
      this.mac.start_xactor();
      ...
   endtask: start
   ...
endclass: tb_env
```

vmm_xactor::stop_xactor()

Stop the execution threads in this transactor instance.

SystemVerilog

```
virtual function void stop_xactor();
```

OpenVera

Not supported.

Description

Stops the execution threads in this transactor instance. The transactor can later be restarted. Any extension of this method must call **`super.stop_xactor()`**. The transactor will actually stop when the **`vmm_xactor::wait_if_stopped()`** or **`vmm_xactor::wait_if_stopped_or_empty()`** method is called. It is calls to these methods that define the granularity of stopping a transactor.

vmm_xactor::reset_xactor()

Reset the state and terminate the execution threads in this transactor instance.

SystemVerilog

```
virtual function void
          reset_xactor(reset_e rst_typ = SOFT_RST);
```

OpenVera

Not supported.

Description

Resets the state and terminates the execution threads in this transactor instance, according to the specified reset type (see Table A-8). The base class indicates the **`vmm_xactor::XACTOR_RESET`** and **`vmm_xactor::XACTOR_IDLE`** notifications and resets the **`vmm_xactor::XACTOR_BUSY`** notification.

Table A-8 Reset Types

Enumerated Value	Broadcasting Operation
vmm_xactor::SOFT_RST	Clears the content of all channels, resets all ON_OFF notifications and terminates all execution threads but maintains the current configuration, notification service and random number generation state information. The transactor must be restarted. This reset type must be implemented.
vmm_xactor::PROTOCOL_RST	Equivalent to a reset signaled via the physical interface. The information affected by this reset is user defined.

Enumerated Value	Broadcasting Operation
vmm_xactor::FIRM_RST	Like **SOFT_RST**, but resets all notification service interface and random-number-generation state information. This reset type must be implemented.
vmm_xactor::HARD_RST	Resets the transactor to the same state found after construction. The registered callbacks are unregistered.

To facilitate the implementation of this method, the actual values associated with these symbolic properties are of increasing magnitude (for example, **vmm_xactor::FIRM_RST** is greater than **vmm_xactor::SOFT_RST**). Not all reset types may be implemented by all transactors. Any extension of this method must call **super.reset_xactor(rst_type)** first to terminate the **vmm_xactor::main()** method, reset the notifications and reset the main thread seed according to the specified reset type. Calling **super.reset_xactor()** with a reset type of **vmm_xactor::PROTOCOL_RST** is functionally equivalent to **vmm_xactor::SOFT_RST**.

Example

Example A-245

```
function void
   mii_mac_layer::reset_xactor(reset_e typ = SOFT_RST);
   super.start_xactor(typ);
   ...
endfunction: reset_xactor
```

vmm_xactor::main()

Fork off this task whenever the **`start_xactor()`** method is called.

SystemVerilog

```
protected virtual task main();
```

OpenVera

Not supported.

Description

This task is forked off whenever the **`start_xactor()`** method is called. It is terminated whenever the **`reset_xactor()`** method is called. The functionality of a user-defined transactor must be implemented in this method. Any additional subthreads must be started within this method, not in the constructor. It can have a blocking or non-blocking implementation.

Any extension of this method must first fork a call to **`super.main()`**.

Example

Example A-246

```
task mii_mac_layer::main();
   super.main();
   ...
endtask: main
```

vmm_xactor::save_rng_state()

Save the state of all random generators.

SystemVerilog

```
virtual function void save_rng_state();
```

OpenVera

Not supported.

Description

This method saves, in local properties, the state of all random generators associated with this transactor instance.

vmm_xactor::restore_rng_state()

Restore the state of all random generators.

SystemVerilog

```
virtual function void restore_rng_state();
```

OpenVera

Not supported.

Description

This method restores, from local properties, the state of all random generators associated with this transactor instance.

vmm_xactor::xactor_status()

Display the current status of the transactor instance.

SystemVerilog

```
virtual function void xactor_status(string prefix = "");
```

OpenVera

Not supported.

Description

Displays the current status of the transactor instance in a human-readable format using the message service interface found in the **`vmm_log::log`** property, using **`vmm_log::NOTE_TYP`** messages. Each line of the status information is prefixed with the specified prefix.

vmm_xactor::‘vmm_callback()

Macro to simplify the syntax of invoking callback methods in a transactor.

SystemVerilog

```
‘vmm_callback(callback_class_name, method(args));
```

OpenVera

Not supported.

Example

Example A-247

Instead of:

```
foreach (this.callbacks[i]) begin
   ahb_master_callbacks cb;
   if ($cast(cb, this.callbacks[i])) continue;
   cb.ptr_tr(this, tr, drop);
end
```

Use:

```
‘vmm_callback(ahb_master_callbacks,
   ptr_tr(this, tr, drop));
```

vmm_xactor::do_what_e

Specifies which methods are to be provided by a shorthand implementation.

SystemVerilog

```
enum {DO_PRINT, DO_START, DO_STOP,
      DO_RESET, DO_ALL} do_what_e;
```

OpenVera

Not supported.

Description

Used to specify which methods are to include the specified data members in their default implementation. `"DO_PRINT"` includes the member in the default implementation of the `psdisplay()` method. `"DO_START"` includes the member in the default implementation of the `start_xactor()` method, if applicable. `"DO_STOP"` includes the member in the default implementation of the `stop_xactor()` method, if applicable. `"DO_RESET"` includes the member in the default implementation of the `reset_xactor()` method, if applicable.

Multiple methods can be specified by adding or or'ing the individual symbolic values. All methods are specified by specifying the `"DO_ALL"` symbol.

Example

Example A-248

```
‘vmm_xactor_member_xactor(idler, DO_ALL - DO_STOP);
```

vmm_xactor::do_psdisplay()

Override the shorthand `psdisplay()` method.

SystemVerilog

```
virtual function string do_psdisplay(string prefix = "")
```

OpenVera

Not supported.

Description

This method overrides the default implementation of the `vmm_xactor::psdisplay()` method created by the vmm_xactor shorthand macros. If defined, it will be used instead of the default implementation.

Example

Example A-249

```
class eth_frame_gen extends vmm_xactor;
   . . .
   virtual function string do_psdisplay(string prefix = "")
      $sformat(do_psdisplay,"%s Printing Ethernet frame \n
          generator members \n",prefix);
      . . .
   endfunction
   . . .
endclass
```

vmm_xactor::do_start_xactor()

Override the shorthand `start_xactor()` method.

SystemVerilog

```
protected virtual function void do_start_xactor()
```

OpenVera

Not supported.

Description

This method overrides the default implementation of the `vmm_xactor::start_xactor()` method created by the vmm_xactor shorthand macros. If defined, it will be used instead of the default implementation.

Example

Example A-250

```
class xact1 extends vmm_xactor;
   . . .
endclass

class xact2 extends vmm_xactor;
   . . .
endclass

class xact extends vmm_xactor;
   xact1 xact1_inst;
   xact2 xact2_inst;
   . . .
   protected virtual function void do_start_xactor ();
     `ifdef XACT_2
```

```
            xact2_inst.start_xactor();
         `else
            xact1_inst.start_xactor();
         `endif
         . . .
      endfunction
      . . .
   endclass
```

vmm_xactor::do_stop_xactor()

Override the shorthand `stop_xactor()` method.

SystemVerilog

```
protected virtual function void do_stop_xactor()
```

OpenVera

Not supported.

Description

This method overrides the default implementation of the `vmm_xactor::stop_xactor()` method created by the vmm_xactor shorthand macros. If defined, it will be used instead of the default implementation.

Example

Example A-251

```
class xact1 extends vmm_xactor;
   . . .
endclass

class xact2 extends vmm_xactor;
   . . .
endclass

class xact extends vmm_xactor;
   xact1 xact1_inst;
   xact2 xact2_inst;
   . . .
   protected virtual function void do_stop_xactor ();
     `ifdef XACT_2
```

```
         xact2_inst.stop_xactor();
      `else
         xact1_inst.stop_xactor();
      `endif
      . . .
   endfunction
   . . .
endclass
```

vmm_xactor::do_reset_xactor()

Override the shorthand `reset_xactor()` method.

SystemVerilog

```
protected virtual function void do_reset_xactor(
    vmm_xactor::reset_e rst_typ)
```

OpenVera

Not supported.

Description

This method overrides the default implementation of the **`vmm_xactor::reset_xactor()`** method created by the **`vmm_xactor`** shorthand macros. If defined, it will be used instead of the default implementation.

Example

Example A-252

```
class xact1 extends vmm_xactor;
   . . .
endclass

class xact2 extends vmm_xactor;
   . . .
endclass

class xact extends vmm_xactor;
   xact1 xact1_inst;
   xact2 xact2_inst;
   . . .
```

```
   protected virtual function void do_reset_xactor ();
     `ifdef XACT_2
        xact2_inst.reset_xactor();
     `else
        xact1_inst.reset_xactor();
     `endif
     . . .
   endfunction
   . . .
endclass
```

vmm_xactor::notifications_e

Predefined notifications.

SystemVerilog

```
typedef enum {XACTOR_IDLE,
     XACTOR_BUSY,
     XACTOR_STARTED,
     XACTOR_STOPPING,
     XACTOR_STOPPED,
     XACTOR_RESET) notifications_e;
```

OpenVera

```
static int XACTOR_IDLE;
static int XACTOR_BUSY;
static int XACTOR_STARTED;
static int XACTOR_STOPPING;
static int XACTOR_STOPPED;
static int XACTOR_RESET;
```

Description

Predefined notifications that are indicated whenever the transactor changes state.

XACTOR_IDLE

ON/OFF notification that is indicated when the transactor is idle. Must be the complement of XACTOR_BUSY.

XACTOR_BUSY

ON/OFF notification that is indicated when the transactor is busy. Must be the complement of XACTOR_IDLE.

XACTOR_STARTED

ONE_SHOT notification indicating that the transactor has been started.

XACTOR_STOPPING

ON/OFF notification indicating that a request has been made for the transactor to stop.

XACTOR_STOPPED

ONE_SHOT notification indicating that all threads in the transactor have been stopped.

XACTOR_RESET

ONE_SHOT notification indicating that the transactor has been reset.

Example

Example A-253

```
xactor.notify.wait_for(vmm_xactor::XACTOR_STARTED);
```

vmm_xactor::psdisplay()

Human-readable description of the transactor.

SystemVerilog

```
virtual function string psdisplay(string prefix = "")
```

OpenVera

Not supported.

Description

This method returns a human-readable description of the transactor. Each line is prefixed with the specified prefix.

Example

Example A-254

```
class xactor extends vmm_xactor;
   . . .
endclass

program prog;
   xactor xact = new;
   . . .
   initial begin
      . . .
      $display("Printing variables of Transactor \n %s \n
",xact.psdisplay());
      . . .
   end

endprogram
```

vmm_xactor::wait_if_stopped()

Suspend an execution thread.

SystemVerilog

```
protected task wait_if_stopped(int unsigned n_threads = 1);
```

OpenVera

```
protected task wait_if_stopped_t(integer n_threads = 1);
```

Description

Blocks the thread execution if the transactor has been stopped via the `stop_xactor()` method or if the specified input channel is currently empty. This method will indicate the `vmm_xactor::XACTOR_STOPPED` and `vmm_xactor::XACTOR_IDLE` notifications and reset the `vmm_xactor::XACTOR_BUSY` notification. The tasks will return once the transactor has been restarted using the `start_xactor()` method and the specified input channel is not empty. These methods do not block if the transactor is not stopped and the specified input channel is not empty.

Calls to this method and the “vmm_xactor::wait_if_stopped_or_empty()” methods define the granularity by which the transactor can be stopped without violating the protocol. If a transaction can be suspended in the middle of its execution, the `wait_if_stopped()` method should be called at every opportunity. If a transaction cannot be suspended, the `wait_if_stopped_or_empty()` method should only be called after the current transaction has been completed, before fetching the next transaction descriptor for the input channel.

If a transactor is implemented using more than one concurrently running thread that must be stopped, the total number of threads to be stopped must be specified in all invocations of this and the "vmm_xactor::wait_if_stopped_or_empty()" method.

Example

Example A-255

```
protected virtual task main();
   super.main();
   forever begin
      transaction tr;
      this.wait_if_stopped_or_empty(this.in_chan);
      this.in_chan.activate(tr);
      ...
      this.wait_if_stopped();
      ...
   end
endtask: main
```

vmm_xactor::wait_if_stopped_or_empty()

Suspend an execution thread or wait on a channel.

SystemVerilog

```
protected task wait_if_stopped_or_empty(vmm_channel chan,
    int unsigned n_threads = 1);
```

OpenVera

```
protected task wait_if_stopped_or_empty_t(rvm_channel chan,
    integer n_threads = 1);
```

Description

Blocks the thread execution if the transactor has been stopped via the **`stop_xactor()`** method or if the specified input channel is currently empty. This method will indicate the **`vmm_xactor::XACTOR_STOPPED`** and **`vmm_xactor::XACTOR_IDLE`** notifications and reset the **`vmm_xactor::XACTOR_BUSY`** notification. The tasks will return once the transactor has been restarted using the **`start_xactor()`** method and the specified input channel is not empty. These methods do not block if the transactor is not stopped and the specified input channel is not empty.

Calls to this method and the “vmm_xactor::wait_if_stopped()” methods define the granularity by which the transactor can be stopped without violating the protocol.

If a transactor is implemented using more than one concurrently running thread that must be stopped, the total number of threads to be stopped must be specified in all invocations of this and the “vmm_xactor::wait_if_stopped()” method.

Example

Example A-256

```
protected virtual task main();
   super.main();
   fork
      forever begin
         transaction tr;
         this.wait_if_stopped_or_empty(this.in_chan, 2);
         this.in_chan.activate(tr);
         ...
         this.wait_if_stopped(2);
         ...
      end

      forever begin
         ...
         this.wait_if_stopped(2);
         ...
      end
   join_none
endtask: main
```

vmm_xactor::get_input_channels()

Returns the input channels of this transactor.

SystemVerilog

```
function void get_input_channels(ref vmm_channel chans[$]);
```

OpenVera

Not supported.

Description

Returns the channels where this transactor has been identifier as the consumer using the `vmm_channel::set_consumer()`.

Example

Example A-257

```
class xactor extends vmm_xactor;
   . . .
endclass

program prog;
   xactor xact = new;
   vmm_channel in_chans[$];
   . . .
   initial begin
      . . .
      xact.get_input_channels(in_chans);
      . . .
   end

endprogram
```

vmm_xactor::get_output_channels()

Returns the output channels of this transactor.

SystemVerilog

```
function void get_output_channels(
    ref vmm_channel chans[$]);
```

OpenVera

Not supported.

Description

Returns the channels where this transactor has been identifier as the producer using the `vmm_channel::set_producer()`.

Example

Example A-258

```
class xactor extends vmm_xactor;
   . . .
endclass

program prog;
   xactor xact = new;
   vmm_channel out_chans[$];
   . . .
   initial begin
      . . .
      xact.get_output_channels(in_chans);
      . . .
   end

endprogram
```

vmm_xactor::inp_vmm_sb_ds()

Refer to the *VMM Scoreboard User Guide*.

vmm_xactor::exp_vmm_sb_ds()

Refer to the *VMM Scoreboard User Guide*.

vmm_xactor::register_vmm_sb_ds()

Refer to the *VMM Scoreboard User Guide*.

vmm_xactor::unregister_vmm_sb_ds()

Refer to the *VMM Scoreboard User Guide*.

vmm_xactor::kill()

Prepare a transactor for deletion.

SystemVerilog

```
function void kill();
```

OpenVera

Not supported.

Description

Prepare the transactor for deletion and reclamation by the garbage collector.

Remove this transactor as the producer of its output channels and as the consumer of its input channels. De-registers all data stream scoreboards and callback extensions.

Example

Example A-259

```
class xactor extends vmm_xactor;
   . . .
endclass

program prog;
   xactor xact = new;
   . . .
   initial begin
      . . .
      xact.kill();
      . . .
```

```
   end

endprogram
```

‘vmm_xactor_member_begin()

Start of shorthand section.

SystemVerilog

```
‘vmm_xactor_member_begin(class-name)
```

OpenVera

Not supported.

Description

Start the shorthand section providing a default implementation for the `psdisplay()`, `start_xactor()`, `stop_xactor()` and `reset_xactor()` methods.

The class-name specified must be the name of the vmm_xactor extension class that is being implemented.

The shorthand section can only contain shorthand macros and must be terminated by a “`‘vmm_xactor_member_end()`” .

Example

Example A-260

```
class eth_mac extends vmm_xactor;
   ...
   ‘vmm_xactor_member_begin(eth_mac)
      ...
   ‘vmm_xactor_member_end(eth_mac)
   ...
endclass
```

‘vmm_xactor_member_end()

End of shorthand section.

SystemVerilog

```
‘vmm_xactor_member_end(class-name)
```

OpenVera

Not supported.

Description

Terminate the shorthand section providing a default implementation for the `psdisplay()`, `start_xactor()`, `stop_xactor()` and `reset_xactor()` methods.

The class-name specified must be the name of the vmm_xactor extension class that is being implemented.

The shorthand section must have been started by a “‘vmm_xactor_member_begin()” .

Example

Example A-261

```
class eth_mac extends vmm_xactor;
   ...
   ‘vmm_xactor_member_begin(eth_mac)
      ...
   ‘vmm_xactor_member_end(eth_mac)
   ...
endclass
```

‘vmm_xactor_member_scalar*()

Shorthand implementation for a scalar data member.

SystemVerilog

```
‘vmm_xactor_member_scalar(member-name,
                          vmm_xactor::do_what_e do_what)

‘vmm_xactor_member_scalar_array(member-name,
                                vmm_xactor::do_what_e do_what)

‘vmm_xactor_member_scalar_aa_scalar(member-name,
                                vmm_xactor::do_what_e do_what)

‘vmm_xactor_member_scalar_aa_string(member-name,
                                vmm_xactor::do_what_e do_what)
```

OpenVera

Not supported.

Description

Add the specified scalar-type, array of scalars, scalar-indexed associative array of scalars or string-indexed associative array of scalars data member to the default implementation of the methods specified by the `’do_what’` argument.

A scalar is an integral type, such as bit, bit vector, and packed unions.

The shorthand implementation must be located in a section started by a “`‘vmm_xactor_member_begin()`” .

Example

Example A-262

```
class eth_frame_gen extends vmm_xactor;
   local integer fr_count;
   . . .
   `vmm_xactor_member_begin(eth_frame_gen);
      `vmm_xactor_member_scalar (fr_count, DO_ALL)
      . . .
   `vmm_xactor_member_end(eth_frame_gen)
   . . .
endclass
```

‘vmm_xactor_member_string*()

Shorthand implementation for a string data member.

SystemVerilog

```
‘vmm_xactor_member_string(member-name,
                          vmm_xactor::do_what_e do_what)

‘vmm_xactor_member_string_array(member-name,
                                vmm_xactor::do_what_e do_what)

‘vmm_xactor_member_string_aa_scalar(member-name,
                                vmm_xactor::do_what_e do_what)

‘vmm_xactor_member_string_aa_string(member-name,
                                vmm_xactor::do_what_e do_what)
```

OpenVera

Not supported.

Description

Add the specified string-type, array of strings, scalar-indexed associative array of strings or string-indexed associative array of strings data member to the default implementation of the methods specified by the `’do_what’` argument.

The shorthand implementation must be located in a section started by a “‘vmm_xactor_member_begin()” .

Example

Example A-263

```
class eth_frame_gen extends vmm_xactor;
```

```
   local string fr_name;
   . . .
   `vmm_xactor_member_begin(eth_frame_gen);
      `vmm_xactor_member_string (fr_name, DO_ALL)
      . . .
   `vmm_xactor_member_end(eth_frame_gen)
   . . .
endclass
```

‘vmm_xactor_member_enum*()

Shorthand implementation for an enumerated data member.

SystemVerilog

```
‘vmm_xactor_member_enum(member-name,
                        vmm_xactor::do_what_e do_what)

‘vmm_xactor_member_enum_array(member-name,
                              vmm_xactor::do_what_e do_what)

‘vmm_xactor_member_enum_aa_scalar(member-name,
                              vmm_xactor::do_what_e do_what)

‘vmm_xactor_member_enum_aa_string(member-name,
                              vmm_xactor::do_what_e do_what)
```

OpenVera

Not supported.

Description

Add the specified enum-type, array of enums, scalar-indexed associative array of enums or string-indexed associative array of enums data member to the default implementation of the methods specified by the `’do_what’` argument.

The shorthand implementation must be located in a section started by a “`‘vmm_xactor_member_begin()`” .

Example

Example A-264

```
class eth_frame_gen extends vmm_xactor;
```

```
   fr_type fr_type_var;
   . . .
   `vmm_xactor_member_begin(eth_frame_gen);
      `vmm_xactor_member_enum (fr_type_var, DO_ALL)
      . . .
   `vmm_xactor_member_end(eth_frame_gen)
   . . .
endclass
```

‘vmm_xactor_member_vmm_data*()

Shorthand implementation for a vmm_data-based data member.

SystemVerilog

```
‘vmm_xactor_member_vmm_data(member-name,
                           vmm_xactor::do_what_e do_what)

‘vmm_xactor_member_vmm_data_array(member-name,
                           vmm_xactor::do_what_e do_what)

‘vmm_xactor_member_vmm_data_aa_scalar(member-name,
                           vmm_xactor::do_what_e do_what)

‘vmm_xactor_member_vmm_data_aa_string(member-name,
                           vmm_xactor::do_what_e do_what)
```

OpenVera

Not supported.

Description

Add the specified vmm_data-type, array of vmm_datas, scalar-indexed associative array of vmm_datas or string-indexed associative array of vmm_datas data member to the default implementation of the methods specified by the `’do_what’` argument.

The shorthand implementation must be located in a section started by a “`‘vmm_xactor_member_begin()`” .

Example

Example A-265

```
class eth_frame extends vmm_data;
   . . .
endclass

class eth_frame_gen extends vmm_xactor;
   eth_frame eth_frame_packet;
   . . .
   `vmm_xactor_member_begin(eth_frame_gen);
     `vmm_xactor_member_vmm_data (eth_frame_packet, DO_ALL)
      . . .
   `vmm_xactor_member_end(eth_frame_gen)
   . . .
endclass
```

‘vmm_xactor_member_channel*()

Shorthand implementation for a channel data member.

SystemVerilog

```
‘vmm_xactor_member_channel(member-name,
                           vmm_xactor::do_what_e do_what)

‘vmm_xactor_member_channel_array(member-name,
                            vmm_xactor::do_what_e do_what)

‘vmm_xactor_member_channel_aa_scalar(member-name,
                            vmm_xactor::do_what_e do_what)

‘vmm_xactor_member_channel_aa_string(member-name,
                            vmm_xactor::do_what_e do_what)
```

OpenVera

Not supported.

Description

Add the specified channel-type, array of channels, dynamic array of channels, scalar-indexed associative array of channels or string-indexed associative array of channels data member to the default implementation of the methods specified by the `’do_what’` argument.

The shorthand implementation must be located in a section started by a “`‘vmm_xactor_member_begin()`” .

Example

Example A-266

```
class eth_frame_gen extends vmm_xactor;
   eth_frame_channel in_chan
   . . .
   `vmm_xactor_member_begin(eth_frame_gen);
      `vmm_xactor_member_channel (in_chan, DO_ALL)
      . . .
   `vmm_xactor_member_end(eth_frame_gen)
   . . .
endclass
```

‘vmm_xactor_member_xactor*()

Shorthand implementation for a transactor data member.

SystemVerilog

```
‘vmm_xactor_member_xactor(member-name,
                          vmm_xactor::do_what_e do_what)

‘vmm_xactor_member_xactor_array(member-name,
                            vmm_xactor::do_what_e do_what)

‘vmm_xactor_member_xactor_aa_scalar(member-name,
                            vmm_xactor::do_what_e do_what)

‘vmm_xactor_member_xactor_aa_string(member-name,
                            vmm_xactor::do_what_e do_what)
```

OpenVera

Not supported.

Description

Add the specified transactor-type, array of transactors, dynamic array of transactors, scalar-indexed associative array of transactors or string-indexed associative array of transactors data member to the default implementation of the methods specified by the `’do_what’` argument.

The shorthand implementation must be located in a section started by a “`‘vmm_xactor_member_begin()`” .

Example

Example A-267

```
class custom_gen extends vmm_xactor;
   . . .
endclass

class eth_frame_gen extends vmm_xactor;
  custom_gen custom_gen_inst;
   . . .
   `vmm_xactor_member_begin(eth_frame_gen);
      `vmm_xactor_member_xactor (custom_gen_inst, DO_ALL)
      . . .
   `vmm_xactor_member_end(eth_frame_gen)
   . . .
endclass
```

‘vmm_xactor_member_user_defined()

User-defined shorthand implementation data member.

SystemVerilog

```
‘vmm_xactor_member_user_defined(member-name)
```

OpenVera

Not supported.

Description

Add the specified user-defined default implementation of the methods specified by the `’do_what’` argument.

Refer to “User-defined vmm_xactor Member Default Implementation” on page 2-11 for details on how to specify the shorthand implementation for a data member.

The shorthand implementation must be located in a section started by a `“‘vmm_xactor_member_begin()”` .

Example

Example A-268

```
class eth_frame_gen extends vmm_xactor;
   integer fr_no;
   . . .
   `vmm_xactor_member_begin(eth_frame_gen);
      `vmm_xactor_member_user_defined (fr_no, DO_ALL)
      . . .
   `vmm_xactor_member_end(eth_frame_gen)
   . . .
```

```
  function bit do_fr_no ( input vmm_data::do_what_e do_what)

      do_fr_no = 1;  // Success, abort by returning 0

      case (do_what)
      . . .
      endcase
   endfunction
   . . .
endclass
```

vmm_xactor_callbacks

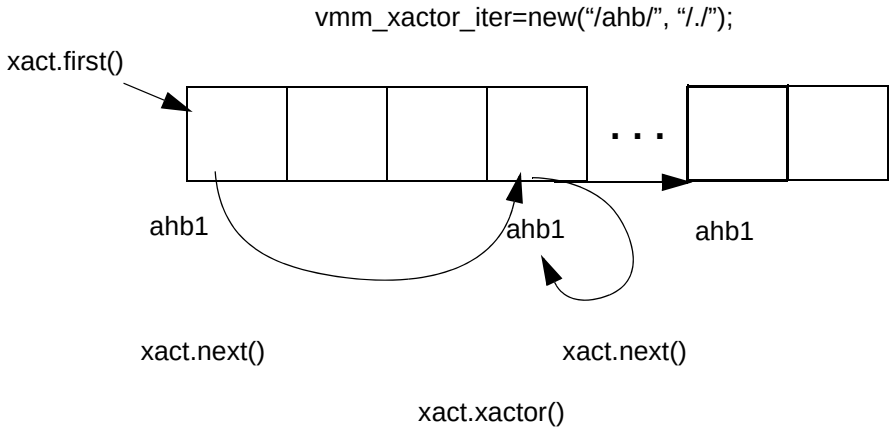
This class implements a pure virtual base class for callback containments. See the documentation for the “vmm_xactor::append_callback()” on page A-672.

vmm_xactor_iter

This class can iterate over all known `vmm_xactor` instances, based on the names and instance names, regardless of their location in the class hierarchy.

VMM adds this class to traverse list a registered transactors that match a regular expression. This feature is very useful to register specific transactor callbacks, connect specific transactors to a scoreboard object, and re-allocate transactor by killing its channels and reassigning some new ones.

```
class driver_typed #(type T = vmm_data) extends vmm_xactor;
  function new(string instance);
    super.new("driver", instance);
  endfunction
  virtual protected task main();
     vmm_channel chans[$];
     super.main();
     get_input_channels(chans);
     foreach (chans[i]) begin
       vmm_channel_typed #(T) chan;
       $cast(chan, chans[i]);
       start_drive(chan, i);
     end
  endtask
  virtual task start_drive(vmm_channel_typed #(T) chan);
    T tr;
    fork
     forever begin
       chan.get(tr);
       `vmm_note(log, tr.psdisplay(“Executing.."));
       wait_if_stopped();
     end
   join_none
  endtask
endclass
```



VMM provides a method to access all transactors available in the environment using the **vmm_xactor_iter**. The VMM transactor iterator iterates over the transactors based on name/instances using regexp. There is no need to know the hierarchical references to the **vmm_xactor** instance. The **vmm_xactor_iter** maintains a single queue of all transactors matching the regular expression provided.

The VMM transactor iterator can be used by either creating a new iterator object using **vmm_xactor_iter::new()** or by using the shorthand macro **`foreach_vmm_xactor()**. The two methods are explained below.

Using the vmm_xactor_iter Class

```
vmm_xactor_iter iter = new("/./" , "/./");
```

Uses regexp style name and instance matching. "/./" will return all the **vmm_xactor** objects present in the environment.

The methods available with **vmm_xactor_iter** are:

- **`vmm_xactor_iter::first()`**

  Resets the iterator to the first transactor , that is to the start of the queue.

- **`vmm_xactor_iter::xactor()`**

  Returns a reference to the current transactor iterated on.

- **`vmm_xactor_iter::next()`**

  Moves the iterator to the next transactor.

The example below shows how to start all transactors extended from the `ahb_transactor` class. The `ahb_transactor` class is extended from **`vmm_xactor`** class

```
vmm_xactor_iter iter = new("/./", "/./");
   // Returns a list of all vmm_xactor objects
while( iter.xactor() != null) begin
   ahb_transactor ahb;
   if($cast(ahb, iter.xactor()) begin
       //get abh_transactor extended objects
     ahb.start_xactor();
   end
   iter.next();
end
```

Using the Shorthand Macro `foreach_vmm_xactor()

The macro **`foreach_vmm_xactor`**`(ahb_transactor,"/./","/./")` requires three arguments. This call returns all objects of `ahb_transactor` and its derived classes. For returning all

vmm_xactor objects, use **vmm_xactor** as the first argument. The second and third arguments are string name and instance, respectively.

The variable name `xact` of the type specified as the first argument is implicitly declared.

The example below achieves the same functionality as above using the shorthand macro.

The macro must be used in the declarative portion of the code or immediately followed by a **begin** keyword.

```
begin
   `foreach_vmm_xactor(ahb_transactor, "/./" , "/./")
   begin
      xact.start_xactor();
   end
end
```

Summary

vmm_xactor_iter::new()

Create a new transactor iterator.

SystemVerilog

```
function void new(string name    = "/./",
    string inst    = "/./");
```

OpenVera

Not supported.

Description

Create a new transactor iterator and initialize it using the specified name and instance name. If the specified name or instance name is enclosed between `//` characters, their are interpreted as regular expressions. Otherwise, they are interpreted as the full name or instance name to match.

"vmm_xactor_iter::first()" is implicitly called. So once created, the first transactor matching the specified name and instance name patterns is available using the "vmm_xactor_iter::xactor()" method. The subsequent transactors can be iterated on, one at a time, using the "vmm_xactor_iter::next()" method.

Example

Example A-269

```
vmm_xactor_iter iter = new("/AHB/");
while (iter.xactor() != null) begin
   ahb_master ahb;
```

```
      if ($cast(ahb, iter.xactor()) begin
         ...
      end
      iter.next();
   end
```

vmm_xactor_iter::first()

Reset the iterator to the first transactor.

SystemVerilog

```
function vmm_xactor first();
```

OpenVera

Not supported.

Description

Reset the iterator to the first transactor matching the name and instance name patterns specified when the iterator was created using `vmm_xactor_iter::new()` and return a reference to it, if found.

Returns *`NULL`* if no transactors match.

The order in which transactors are iterated on is unspecified.

Example

Example A-270

```
int i = 0;
vmm_xactor_iter iter = new("/AHB/", "");
vmm_xactor xa;
for (xa = iter.first(); xa != null; xa= iter.next()) i++;
`vmm_note (log, $psprintf("No. of AHB transactors = %0d ",i))
                                                  i));
```

vmm_xactor_iter::xactor()

Return the current transactor iterated on.

SystemVerilog

```
function vmm_xactor xactor();
```

OpenVera

Not supported.

Description

Return a reference to a transactor matching the name and instance name patterns specified when the iterator was created using `vmm_xactor_iter::new()`.

Returns *`NULL`* if no transactors match.

Example

Example A-271

```
vmm_xactor_iter iter = new("/AHB/");
while (iter.xactor() != null) begin
   ahb_master ahb;
   if ($cast(ahb, iter.xactor()) begin
      ...
   end
   iter.next();
end
```

vmm_xactor_iter::next()

Move the iterator to the next transactor.

SystemVerilog

```
function vmm_xactor next();
```

OpenVera

Not supported.

Description

Move the iterator to the next transactor matching the name and instance name patterns specified when the iterator was created using `vmm_xactor_iter::new()` and return a reference to it, if found.

Returns *`NULL`* if no transactors match.

The order in which transactors are iterated on is unspecified.

Example

Example A-272

```
int i = 0;
vmm_xactor_iter iter = new("/AHB/", "");
vmm_xactor xa;
for (xa = iter.first(); xa != null; xa= iter.next()) i++;
`vmm_note (log, $psprintf("No. of AHB transactors = %0d ",i))
```

‘foreach_vmm_xactor()

Shorthand transactor iterator macro.

SystemVerilog

```
‘foreach_vmm_xactor(type, name, inst) begin
   xact...
end
```

OpenVera

Not supported.

Description

Shorthand macro to simplify the creation and operation of a transactor iterator instance, looking for transactors of a specific type, matching a specific name and instance name. The subsequent statement is executed for each transactor iterated on.

A variable named `"xact"` of the type specified as the first argument to the macro is implicitly declared and iteratively set to each transactor of the specified type that matches the specified name and instance name.

The macro must be located immediately after a `"begin"` keyword.

Example

Example A-273 Iterating over all transactors of type "ahb_master"

```
begin
   ‘foreach_vmm_xactor(ahb_master, "/./", "/./") begin
      xact.register_callback(...);
   end
```

```
end
```

B

Command-line Options

This appendix provides detailed documentation on VMM-related command-line options.

The OpenVera and SystemVerilog standard libraries have identical functionality and features. The command-line arguments affecting their operation are thus documented together. How command-line arguments are specified may differ between tools.

The command-line arguments are documented in alphabetical order. A summary of all available command-line arguments, with cross references to the page where their detailed documentation can be found, is provided at the beginning of this Appendix.

Argument Summary

+vmm_channel_shared_log

Use a single `vmm_log` instance for all `vmm_channel` instances.

Description

By default, all `vmm_channel` instances have a unique and separate `vmm_log` instance in their `vmm_channel::log property`. When this command-line option is used, all `vmm_channel::log` properties of all `vmm_channel` instances will refer to the same `vmm_log` instance.

This run-time command-line option can be used when a large number of channel instances are created in a verification environment, usually triggering a warning message after 200 `vmm_log` instances have been created.

Side Effects

The name and instance names of channels are assumed to be the name and instance name of the `vmm_log` within it. Because all channel instances will share the same `vmm_log` instance, they will have the same name and instance name.

It will no longer be possible to control error and debug messages produced by channels on a per-instance basis.

Furthermore, it will not be possible to identify the source of a message produced by a channel, as the reported name will be "*VMM Channel([shared])*" in all instances.

C

Class Customization Macros

This appendix provides detailed documentation component customization macros available in the VMM Standard Library.

The OpenVera and SystemVerilog standard libraries have identical functionality and features. The macros used to customize the components they contain are documented together. The syntax to define and refer to a macro is different in each language. The name of the macro is the same, except for the `RVM_` prefix which is used in OpenVera, and the `VMM_` prefix which is used in SystemVerilog.

The macros are documented for each class they affect. The affected classes are listed in alphabetical order. A summary of all available customizable classes, with cross references to the page where their detailed documentation can be found, is provided at the beginning of this Appendix.

Customizable Class Summary

vmm_atomic_gen
vmm_scenario_gen
vmm_broadcast
vmm_scheduler
vmm_watchdog

Structure

```
class vmm_broadcast extends ‘VMM_XACTOR;
   ...
   extern function new(string name,
                       string inst,
                       int    stream_id = -1,
                       ...
                       ‘VMM_XACTOR_NEW_ARGS);
endclass: vmm_broadcast


function vmm_broadcast::new(string name,
                            string inst,
                            int    stream_id,
                            ...
                            ‘VMM_XACTOR_NEW_ARGS);
   super.new(name, inst, stream_id ‘VMM_XACTOR_NEW_CALL);
      ...
endfunction: new
```

Macros

The following macros are available to retarget the pre-defined transactors in the VMM Standard Library to a base class other than `vmm_xactor`.

`VMM_XACTOR`
Base class of the transactor. If this macro is not defined (the default), it is defined to `vmm_xactor`.

VMM_XACTOR_NEW_ARGS
Additional argument declarations required by the base class constructor. Must start with a comma. All arguments must have a default value. Must be defined if VMM_XACTOR is defined.

VMM_XACTOR_NEW_CALL
Values of additional arguments required by the base class constructor. Must start with a comma. Must be defined if VMM_XACTOR is defined.

See Also

See “vmm_xactor” for customizing the vmm_xactor class.

Example C-1

```
. . .
`define VMM_XACTOR my_xactor
`define VMM_XACTOR_NEW_ARGS ,int i = 0,int j = 0
`define VMM_XACTOR_NEW_EXTERN_ARGS ,int i,int j
`define VMM_XACTOR_NEW_CALL 10,10

class my_xactor;
   . . .
   function new();
      . . .
      `vmm_note(log,$psprintf({"vmm_broadcast is extended
      from my_xactor with ", "%0d and %0d arguments."},i,j));
      . . .
   endfunction
   . . .
endclass
. . .
```

vmm_channel

Structure

```
‘define vmm_channel(class_name) \
   class class_name‘‘_channel extends ‘VMM_CHANNEL; \
      ‘VMM_LOG    log; \
      ‘VMM_NOTIFY notify; \
      ... \
   endclass: class_name‘‘_channel

class vmm_channel extends ‘VMM_CHANNEL_BASE;
   ‘VMM_LOG       log;
   ‘VMM_NOTIFY    notify;
   ...
   function new(...);
      super.new(‘VMM_CHANNEL_BASE_NEW_CALL);
      ...
   endfunction: new

   ‘VMM_CHANNEL_BASE_METHODS
   ...
endclass: vmm_channel
```

Macros

The following macros are available to customize the `vmm_channel` class. It may be necessary to redefine a group of macros together if one of them is redefined.

`VMM_CHANNEL`
Name of the transaction-level interface base class. The user-defined class must ultimately be based on the `vmm_channel` class. If this macro is not defined (the default), the class named `vmm_channel` is used.

VMM_CHANNEL_BASE
Base class of the vmm_channel utility class. If this macro is not defined (the default), the vmm_channel class is not based on any other class.

VMM_LOG
Name of the message service interface class. The user-defined class must ultimately be based on the vmm_log class. If this macro is not defined (the default), the class named vmm_log is used.

VMM_NOTIFY
Name of the notification service interface class. The user-defined class must ultimately be based on the vmm_notify class. If this macro is not defined (the default), the class named vmm_notify is used.

VMM_CHANNEL_BASE_NEW_CALL
Values of arguments required by the base class constructor. Requires that the VMM_CHANNEL_BASE macro be defined.

VMM_CHANNEL_BASE_METHODS
Implementation of virtual methods declared in the base class. Requires that the VMM_CHANNEL_BASE macro be defined.

Example C-2

```
. . .
`define VMM_CHANNEL my_chan

//Now user-defined my_chan will be used instead of
// vmm_channel
class my_chan;
. . .
endclass

`define VMM_CHANNEL_BASE my_channel
//If you don't define VMM_CHANNEL_BASE_NEW_CALL then also
//it will call super.new() because you have defined
//VMM_CHANNEL_BASE
`define VMM_CHANNEL_BASE_NEW_CALL
```

```
`define VMM_CHANNEL_BASE_METHODS extern function void \
    my_function1(); extern function void my_function2();

`define VMM_LOG my_log
`define VMM_NOTIFY my_notify

class my_channel;
   . . .
   vmm_log log;
   function new();
      log = new("my_channel","log");
      `vmm_note(log,"vmm_channel is extended from
         my_channel");
   endfunction
   . . .
endclass
. . .
class my_log extends vmm_log;
  // You can overwrite all the vmm_log method here and those
  // methods will be used everywhere if you have defined
  // VMM_LOG
   . . .
end
. . .
class my_notify extends vmm_notify;
   // You can overwrite all the vmm_notify method here and
   // those methods will be
   //used everywhere if you have defined VMM_NOTIFY
   . . .
end
. . .
//Use of VMM_CHANNEL_BASE_METHODS
//These functions will be used as a vmm_channel class
//functions
function void vmm_channel::my_function1();
   . . .
endfunction

function void vmm_channel::my_function2();
  . . .
endfunction
. . .
```

vmm_consensus

Structure

```
class vmm_consensus extends ‘VMM_CONSENSUS_BASE;
   ‘VMM_LOG log;
   ...
   function new(...);
      super.new(‘VMM_CONSENSUS_BASE_NEW_CALL);
      ...
   endfunction: new

   ‘VMM_CONSENSUS_BASE_METHODS
   ...
endclass: vmm_consensus
```

Macros

The following macros are available to customize the vmm_env class. It may be necessary to redefine a group of macros together if one of them is redefined.

VMM_CONSENSUS_BASE
Base class of the vmm_consensus utility class. If this macro is not defined (the default), the vmm_consensus class is not based on any other class.

VMM_LOG
Name of the message service interface class. The user-defined class must ultimately be based on the vmm_log class. If this macro is not defined (the default), the class named vmm_log is used.

VMM_CONSENSUS_BASE_NEW_CALL
Values of arguments required by the base class constructor. Requires that the VMM_CONSENSUS_BASE macro be defined.

VMM_CONSENSUS_BASE_METHODS

Implementation of virtual methods declared in the base class. Requires that the VMM_CONSENSUS_BASE macro be defined.

Example C-3

```
. . .
//Refer Base example of vmm_channel Example C-2
`define VMM_CONSENSUS_BASE my_consensus

//If you don't define VMM_CONSENSUS_BASE_NEW_CALL then also
//it will call
//super.new() because you have defined VMM_CONSENSUS_BASE
`define VMM_CONSENSUS_BASE_NEW_CALL
`define VMM_CONSENSUS_BASE_METHODS extern function void
    my_function1();

`define VMM_LOG my_log

function void vmm_consensus::my_function1();
   . . .
endfunction
. . .
```

vmm_data

Structure

```
class vmm_data extends ‘VMM_DATA_BASE;
   ‘VMM_NOTIFY notify;
   ...
   function new(vmm_log log
                ‘VMM_DATA_BASE_NEW_ARGS);
      super.new(‘VMM_DATA_BASE_NEW_CALL);
      ...
   endfunction: new

   ‘VMM_DATA_BASE_METHODS
   ...
endclass: vmm_data
```

Macros

The following macros are available to customize the `vmm_data` class. It may be necessary to redefine a group of macros together if one of them is redefined.

`VMM_DATA_BASE`
Base class of the `vmm_data` class. If this macro is not defined (the default), the `vmm_data` is not based on any other class.

`VMM_DATA_BASE_NEW_ARGS`
Additional argument declarations required by the base class constructor. Must start with a comma. All arguments must have a default value. Requires that the `VMM_DATA_BASE` macro be defined.

`VMM_DATA_BASE_NEW_CALL`
Values of arguments required by the base class constructor. Requires that the `VMM_DATA_BASE` macro be defined.

VMM_DATA_BASE_METHODS

Implementation of virtual methods declared in the base class. Requires that the VMM_DATA_BASE macro be defined.

Example C-4

```
. . .
//Refer Base example of vmm_channel Example C-2
`define VMM_DATA_BASE my_data
`define VMM_DATA_BASE_NEW_CALL 10,10
`define VMM_DATA_BASE_METHODS extern function void
    my_function1();
`define VMM_DATA_BASE_NEW_ARGS ,int i = 0,int j = 0
`define VMM_DATA_BASE_NEW_EXTERN_ARGS ,int i,int j
. . .
class my_data;
   . . .
   vmm_log log;
   function new(int i, int j);
      `vmm_note(log,$psprintf({"vmm_data is extended from
          my_data with %0d ", "and %0d arguments,",i,j);
   endfunction
   . . .
endclass
. . .
function void vmm_data::my_function1();
   . . .
endfunction
. . .
```

vmm_env

Structure

```
class vmm_env extends ‘VMM_ENV_BASE;
   ‘VMM_LOG       log;
   ‘VMM_NOTIFY    notify;
   ‘VMM_CONSENSUS end_vote;
   ...
   function new(string name
                ‘VMM_ENV_BASE_NEW_ARGS);
      super.new(‘VMM_ENV_BASE_NEW_CALL);
      ...
   endfunction: new

   ‘VMM_ENV_BASE_METHODS
   ...
endclass: vmm_env
```

Macros

The following macros are available to customize the vmm_env class. It may be necessary to redefine a group of macros together if one of them is redefined.

VMM_ENV_BASE
Base class of the vmm_env class. If this macro is not defined (the default), the vmm_env is not based on any other class.

VMM_LOG
Name of the message service interface class. The user-defined class must ultimately be based on the vmm_log class. If this macro is not defined (the default), the class named vmm_log is used.

VMM_NOTIFY
Name of the notification service interface class. The user-defined class must ultimately be based on the vmm_notify class. If this macro is not defined (the default), the class named vmm_notify is used.

VMM_CONSENSUS
Name of the end-of-test decision-making class. The user-defined class must ultimately be based on the vmm_consensus class. If this macro is not defined (the default), the class named vmm_consensus is used.

VMM_ENV_BASE_NEW_ARGS
Additional argument declarations required by the base class constructor. Must start with a comma. All arguments must have a default value. Requires that the VMM_ENV_BASE macro be defined.

VMM_ENV_BASE_NEW_CALL
Values of arguments required by the base class constructor. Requires that the VMM_ENV_BASE macro be defined.

VMM_ENV_BASE_METHODS
Implementation of virtual methods declared in the base class. Requires that the VMM_ENV_BASE macro be defined.

Example C-5

```
. . .
//Refer Base example of vmm_channel Example C-2
`define VMM_ENV_BASE my_env
`define VMM_ENV_BASE_NEW_ARGS ,int i = 0,int j = 0
`define VMM_ENV_BASE_NEW_EXTERN_ARGS ,int i,int j
`define VMM_ENV_BASE_NEW_CALL 10,10
`define VMM_ENV_BASE_METHODS extern function void
    my_function1();
`define VMM_LOG my_log
`define VMM_NOTIFY my_notify
`define VMM_CONSENSUS my_consensus

class my_env;
```

```
   . . .
   function new();
      . . .
      `vmm_note(log,$psprintf({"vmm_env is extended from
          my_env with %0d and ", "%0d arguments."},i,j));
      . . .
   endfunction
   . . .
endclass

class my_consensus extends vmm_consensus;
   //User can overwrite to inroduce extra functionality
   . . .
endclass
. . .
```

vmm_log

Structure

```
class vmm_log extends ‘VMM_LOG_BASE;
   ...
   function new(...);
      super.new(‘VMM_LOG_BASE_NEW_CALL);
      ...   endfunction: new

   ‘VMM_LOG_BASE_METHODS
   ...
endclass: vmm_log
```

Macros

The following macros are available to customize the vmm_env class. It may be necessary to redefine a group of macros together if one of them is redefined.

VMM_LOG_BASE
Base class of the vmm_log utility class. If this macro is not defined (the default), the vmm_log class is not based on any other class.

VMM_LOG_BASE_NEW_CALL
Values of arguments required by the base class constructor. Requires that the VMM_LOG_BASE macro be defined.

VMM_LOG_BASE_METHODS
Implementation of virtual methods declared in the base class. Requires that the VMM_LOG_BASE macro be defined.

Example C-6

```
//Refer Base example of vmm_channel Example C-2
`define VMM_LOG_BASE my_log
`define VMM_LOG_BASE_NEW_CALL
`define VMM_LOG_BASE_METHODS extern function void
```

```
my_function1();

class my_log;
   . . .
endclass
. . .
//Use of VMM_LOG_BASE_METHODS
//These functions will be used as a vmm_log class functions
function void vmm_log::my_function1();
   . . .
endfunction
. . .
```

vmm_notify

Structure

```
class vmm_notify extends ‘VMM_NOTIFY_BASE;
   ...
   function new(...);
      super.new(‘VMM_NOTIFY_BASE_NEW_CALL);
      ...
   endfunction: new

   ‘VMM_NOTIFY_BASE_METHODS
   ...
endclass: vmm_notify
```

Macros

The following macros are available to customize the vmm_env class. It may be necessary to redefine a group of macros together if one of them is redefined.

VMM_NOTIFY_BASE
Base class of the vmm_notify utility class. If this macro is not defined (the default), the vmm_notify class is not based on any other class.

VMM_NOTIFY_BASE_NEW_CALL
Values of arguments required by the base class constructor. Requires that the VMM_NOTIFY_BASE macro be defined.

VMM_NOTIFY_BASE_METHODS
Implementation of virtual methods declared in the base class. Requires that the VMM_NOTIFY_BASE macro be defined.

Example C-7

```
. . .
//Refer Base example of vmm_channel Example C-2
`define VMM_NOTIFY_BASE my_notify
```

```
`define VMM_NOTIFY_BASE_NEW_CALL
`define VMM_NOTIFY_BASE_METHODS extern function void
     my_function1();
. . .
class my_notify;
   . . .
endclass
. . .
```

vmm_object

Structure

```
class vmm_object extends ‘VMM_OBJECT_BASE;
   ...
   function new(vmm_parent parent = NULL
                ‘VMM_OBJECT_BASE_NEW_ARGS);
      super.new(‘VMM_DATA_BASE_NEW_CALL);
      ...
   endfunction: new

   ‘VMM_OBJECT_BASE_METHODS
   ...
endclass: vmm_object
```

Macros

The following macros are available to customize the vmm_object class. It may be necessary to redefined a group of macros together is one of them is redefined.

VMM_OBJECT_BASE
Base class of the vmm_object class. If this macro is not defined (the default), the vmm_object class is not based on any other class.

VMM_OBJECT_BASE_NEW_ARGS
Additional argument declarations required by the base class constructor. Must start with a comma. All arguments must have a default value. Requires that the VMM_OBJECT_BASE macro be defined.

VMM_OBJECT_BASE_NEW_EXTERN_ARGS
Additional argument declarations required by the base class constructor but without default values. Must start with a comma. Requires that the VMM_OBJECT_BASE macro be defined. Must be defined if the VMM_OBJECT_BASE_NEW_ARGS macro is defined.

VMM_OBJECT_BASE_NEW_CALL

Values of arguments required by the base class constructor. Requires that the VMM_OBJECT_BASE macro be defined.

VMM_OBJECT_BASE_METHODS

Implementation of virtual methods declared in the base class. Requires that the VMM_OBJECT_BASE macro be defined.

Example C-8

```
. . .
//Refer Base example of vmm_channel Example C-2
`define VMM_OBJECT_BASE my_object
`define VMM_OBJECT_BASE_NEW_CALL 10,10
`define VMM_OBJECT_BASE_METHODS extern function void
    my_function1();
`define VMM_OBJECT_BASE_NEW_ARGS ,int i = 0,int j = 0
`define VMM_OBJECT_BASE_NEW_EXTERN_ARGS ,int i,int j

class my_object;
   . . .
   function new(int i, int j);
     `vmm_note(log,$psprintf({"vmm_object is extended from
        my_object with %d", "and %0d arguments."},i,j));
   endfunction
   . . .
endclass
. . .
```

vmm_scenario

Structure

```
class vmm_scenario extends ‘VMM_SCENARIO_BASE;
   function new(‘VMM_LOG log
                ‘VMM_SCENARIO_BASE_NEW_ARGS);
      super.new(log, ‘VMM_SCENARIO_BASE_NEW_CALL);
      ...
   endfunction: new

   ‘VMM_SCENARIO_BASE_METHODS
   ...
endclass: vmm_data
```

Macros

The following macros are available to customize the vmm_scenario class. It may be necessary to redefine a group of macros together is one of them is redefined.

VMM_SCENARIO_BASE
Base class of the vmm_scenario class. If this macro is not defined (the default), the vmm_scenario class is based on the vmm_data class.

VMM_SCENARIO_BASE_NEW_ARGS
Additional argument declarations required by the base class constructor. Must start with a comma. All arguments must have a default value. Defined to VMM_DATA_BASE_NEW_ARGS by default.

VMM_SCENARIO_BASE_NEW_EXTERN_ARGS
Additional argument declarations required by the base class constructor but without default values. Must start with a comma. Must be defined if the VMM_SCENARIO_BASE_NEW_ARGS macro is defined. Defined to VMM_DATA_BASE_NEW_EXTERN_ARGS by default.

VMM_SCENARIO_BASE_NEW_CALL
Values of arguments required by the base class constructor. Defined to VMM_DATA_BASE_NEW_CALL by default.

VMM_SCENARIO_BASE_METHODS
Implementation of virtual methods declared in the base class. Defined to VMM_SCENARIO_BASE_METHODS by default.

Example C-9

```
. . .
//Refer Base example of vmm_channel Example C-2
`define VMM_SCENARIO_BASE my_scenario
`define VMM_SCENARIO_BASE_NEW_CALL 10,10
`define VMM_SCENARIO_BASE_METHODS extern function void
    my_function1();
`define VMM_SCENARIO_BASE_NEW_ARGS ,int i = 0,int j = 0
`define VMM_SCENARIO_BASE_NEW_EXTERN_ARGS ,int i,int j

class my_scenario;
   . . .
   function new(int i, int j);
      `vmm_note(log,$psprintf({"vmm_scenario is extended
         from my_scenario ", "with %0d and %0d
         arguments."},i,j));
   endfunction
   . . .
endclass
. . .
```

*usertype*_scenario

Structure

```
class user_type_scenario extends ‘VMM_SCENARIO;
   function new(‘VMM_SCENARIO_NEW_ARGS);
      super.new(log, ‘VMM_SCENARIO_NEW_CALL);
      ...
   endfunction: new
   ...
endclass: vmm_data
```

Macros

The following macros are available to customize the *usertype*_scenario base class, defined by the ‘vmm_scenario_gen macro. It may be necessary to redefine a group of macros together is one of them is redefined.

VMM_SCENARIO
Base class of the scenario class. If this macro is not defined (the default), the scenario class is based on the vmm_scenario class.

VMM_SCENARIO_NEW_ARGS
Additional argument declarations required by the base class constructor. Must start with a comma. All arguments must have a default value. Defined to VMM_DATA_NEW_ARGS by default.

VMM_SCENARIO_NEW_EXTERN_ARGS
Additional argument declarations required by the base class constructor but without default values. Must start with a comma. Must be defined if the VMM_SCENARIO_NEW_ARGS macro is defined. Defined to VMM_DATA_NEW_EXTERN_ARGS by default.

VMM_SCENARIO_NEW_CALL
Values of arguments required by the base class constructor. Defined to VMM_DATA_NEW_CALL by default.

Example C-10

```
//Refer Base example of vmm_channel Example C-2
`define VMM_SCENARIO my_scenario
`define VMM_SCENARIO_NEW_ARGS ,int i = 0,int j = 0
`define VMM_SCENARIO_NEW_EXTERN_ARGS ,int i,int j
`define VMM_SCENARIO_NEW_CALL 10,10

class my_scenario;
   . . .
   function new(int i, int j);
      `vmm_note(log,$psprintf({"user_type_scenario is
          extended from ", "my_scenario with %0d and %0d
          arguments."},i,j));
   endfunction
   . . .
endclass
. . .
```

vmm_xactor

Structure

```
class vmm_xactor extends ‘VMM_XACTOR_BASE;
   ‘VMM_LOG    log;
   ‘VMM_NOTIFY notify;
   ...
   function new(string name,
                string inst,
                int    stream_id = -1
                ‘VMM_XACTOR_BASE_NEW_ARGS);
      super.new(‘VMM_XACTOR_BASE_NEW_CALL);
      ...
   endfunction: new

   ‘VMM_XACTOR_BASE_METHODS
   ...
endclass: vmm_xactor
```

Macros

The following macros are available to customize the `vmm_xactor` class. It may be necessary to redefine a group of macros together if one of them is redefined.

`VMM_XACTOR_BASE`
Base class of the `vmm_xactor` class. If this macro is not defined (the default), the `vmm_xactor` is not based on any other class.

`VMM_LOG`
Name of the message service interface class. The user-defined class must ultimately be based on the `vmm_log` class. If this macro is not defined (the default), the class named `vmm_log` is used.

`VMM_NOTIFY`
Name of the notification service interface class. The user-defined class must ultimately be based on the `vmm_notify` class. If this macro is not defined (the default), the class named `vmm_notify` is used.

`VMM_XACTOR_BASE_NEW_ARGS`
Additional argument declarations required by the base class constructor. Must start with a comma. All arguments must have a default value. Requires that the `VMM_XACTOR_BASE` macro be defined.

`VMM_XACTOR_BASE_NEW_CALL`
Values of arguments required by the base class constructor. Requires that the `VMM_XACTOR_BASE` macro be defined.

`VMM_XACTOR_BASE_METHODS`
Implementation of virtual methods declared in the base class. Requires that the `VMM_XACTOR_BASE` macro be defined.

See Also

See “vmm_atomic_gen” on page C-3 for customizing the transactors based on this class, that are pre-defined in the VMM Standard Library.

Example C-11

```
. . .
//Refer Base example of vmm_channel Example C-2
`define VMM_XACTOR_BASE my_xactor
`define VMM_XACTOR_BASE_NEW_ARGS ,int i = 0,int j = 0
`define VMM_XACTOR_BASE_NEW_EXTERN_ARGS ,int i,int j
`define VMM_XACTOR_BASE_NEW_CALL 10,10
`define VMM_XACTOR_BASE_METHODS extern function void
    my_function1();
`define VMM_LOG my_log
`define VMM_NOTIFY my_notify
```

```
class my_xactor;
   . . .
   function new();
      . . .
     `vmm_note(log,$psprintf({"vmm_xactor is extended from
         my_xactor with ", "%0d and %0d arguments."},i,j));
      . . .
   endfunction
   . . .
endclass
. . .
```

xvc_manager

Structure

```
class xvc_xvc_manager;
   ‘VMM_LOG    log;
   ‘VMM_LOG    trace;
   ‘VMM_NOTIFY notify;
   ...
endclass: xvc_manager
```

Macros

The following macros are available to customize the xvc_manager class. It may be necessary to redefined a group of macros together is one of them is redefined.

`VMM_LOG`
Name of the message service interface class. The user-defined class must ultimately be based on the `vmm_log` class. If this macro is not defined (the default), the class named `vmm_log` is used.

Example C-12

```
. . .
`define VMM_LOG my_log
. . .
class my_log extends vmm_log;
 //You can overwrite all the vmm_log method here and those
 //methods will be used everywhere if you have defined VMM_LOG
   . . .
end
. . .
```

xvc_xactor

Structure

```
class xvc_xactor extends ‘VMM_XACTOR;
   ‘VMM_LOG trace;
   ...
   extern function new(string name,
                       string inst,
                       int    stream_id = -1,
                       ...
                       ‘VMM_XACTOR_NEW_ARGS);
endclass: xvc_xactor


function xvc_xactor::new(string name,
                         string inst,
                         int    stream_id,
                         ...
                         ‘VMM_XACTOR_NEW_ARGS);
   super.new(name, inst, stream_id ‘VMM_XACTOR_NEW_CALL);
      ...
endfunction: new
```

Macros

The following macros are available to customize the `xvc_xactor` class. It may be necessary to redefine a group of macros together if one of them is redefined.

`VMM_XACTOR`
Base class of the transactor. If this macro is not defined (the default), it is defined to `vmm_xactor`.

`VMM_LOG`
Name of the message service interface class. The user-defined class must ultimately be based on the `vmm_log` class. If this macro is not defined (the default), the class named `vmm_log` is used.

VMM_XACTOR_NEW_ARGS
Additional argument declarations required by the base class constructor. Must start with a comma. All arguments must have a default value. Must be defined if VMM_XACTOR is defined.

VMM_XACTOR_NEW_CALL
Values of additional arguments required by the base class constructor. Must start with a comma. Must be defined if VMM_XACTOR is defined.

See Also

See "vmm_xactor" for customizing the vmm_xactor class itself.

Example C-13

```
. . .
//Refer Base example of vmm_channel Example C-2
`define VMM_XACTOR my_xactor
`define VMM_XACTOR_NEW_ARGS ,int i = 0,int j = 0
`define VMM_XACTOR_NEW_EXTERN_ARGS ,int i,int j
`define VMM_XACTOR_NEW_CALL 10,10
`define VMM_LOG my_log

class my_xactor;
   . . .
   function new();
      . . .
     `vmm_note(log,$psprintf({"xvc_xactor is extended from
        my_xactor with ", "%0d and %0d arguments."},i,j));
      . . .
   endfunction
   . . .
endclass
. . .
```