SYNOPSYS®

Introduction to Design Verification
with VMM:
A Quickstart Guide

Version 1.0

Verification

Copyright Notice and Proprietary Information

Synopsys, Inc.
http://www.vmmcentral.org/
http://www.synopsys.com/SystemVerilog/
http://www.vmm-sv.com/
http://www.synopsys.com/Solutions/EndSolutions/DiscoveryVerification/Pages/default.aspx
http://www.synopsys.com/Tools/Verification/Pages/default.aspx

Revision History	Date	Description
Version 1.0	January 2008	Updated for vmmcentral.org, with VMM version 1.0

Release information:

.

Table of Contents

1.0 Introduction

The Verification Methodology Manual for SystemVerilog (VMM) describes the framework for developing re-usable verification components and testbench verification environment that provides for higher productivity and enables interoperability. In generic terminology, the VMM consists of coding guidelines and a set of base classes. The VMM book documents advanced functional verification techniques used by industry experts to validate complex SoCs.

In this document we will introduce the basic concepts of the VMM. This tutorial and quick guide is intended to provide design and verification engineer deeper understanding of the terminology, the methodology rules and use model. We will touch upon the highlights of VMM; a comprehensive methodology for design verification. This will be sufficient to jump start the novice users in learning the basics of VMM and immediately applying it to their verification at hand.

In the next chapters we will describe the basic libraries and utility classes through simple examples. We will build a testbench environment for a simple FIFO design in the last chapter which combines the basic and minimum recommendations of the Verification Methodology Manual (VMM).

This text accompanies examples and verification environment which can be downloaded and used for training and tutorial purposes.

It is assumed that the reader has some familiarity with SystemVerilog and its constructs. It is recommended that the quickstart guides to SystemVerilog with VCS be reviewed before studing this introduction. You can find the quickstart guides in the systemverilog nativetestbench directory of VCS document installation.

Design and Verification Flow

Verification is a major task in order to making sure the design meets the requirements set out by the architects and designers. Armed with product specification which leads into functional specification/architecture of the design, two parallel paths are taken, starting with verification planning and implementation planning.

In this handbook we will concentrate on the dynamic verification, the simulation-based verification tasks and process. The main area of concentration hence is on the testbench development. Additional aspects of verification such as assertions, semi-formal and formal verifications will be covered in future booklets.

Based on the plan and architecture the verification environment is drawn up, with appropriate components, such as transactors, data models in transactions, scoreboards, monitors and coverage gathering mechanisms.
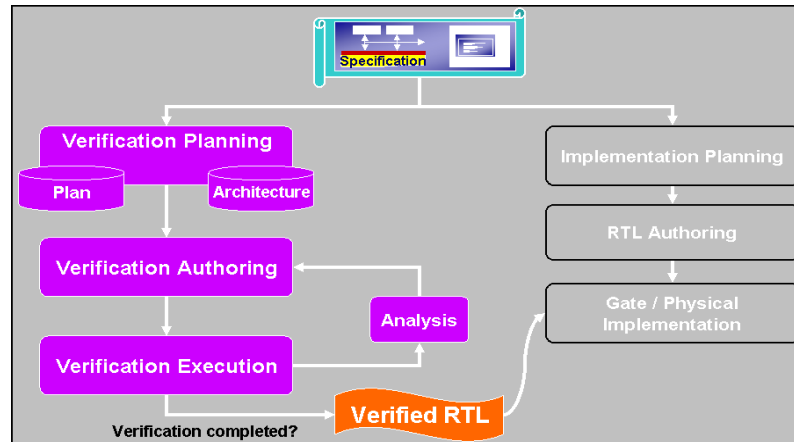
Figure 1: Design and Verification Flow abstraction

Of course this is an iterative process once the verification environment is authored, upon execution, the simulation results need to be thoroughly analyzed and checked. This process will continue until engineering team is satisfied that verification is complete. Functional coverage will help place a closure to this task.

2.0 Verification Process Overview

In this section we will review the basic elements of structured verification development.

Goals of Verification

In the recent past electronic designs have become more complex and have required more ready-made foundations of design blocks as well as the verification blocks. This complexity has translated itself into even more complex verification components and environments. Verification has become very critical segment of complex system and chip designs. The challenge is to raise productivity and quality of the design verification at the same time that you streamline the process and reduce the time it takes to functionally validate the design before fabrication.

One of the ways to help design verification engineers in this process involves modularized development of the verification environment. This is an important step to allow engineers to divide up the tasks in properly partitioned zones which can be connected together with minimal efforts. The fact is that today's systems require more time to develop the random test scenarios. This will require a robust infrastructure.

Framework of Verification

In any complex task undertaking be it design, verification, or take example of building constructions, starting with a well-defined and thought-through baseline and framework provide for higher productivity and improve the design and task efficiency. In this respect if verification engineers can build their environment on top of a well structured base the initial steps of development are done faster and the task shifts to generation of tests and scenarios that stimulates the design under test for unraveling the hard-to-find bugs. Once the baseline framework is better defined then the task of verification engineers will be shifted from development of the libraries and base classes to defining what types of test cases need to be created.

Today's designs also work in many levels of abstractions from high-level abstract models, transaction-level models to gate-netlist need to be verified for correctness. In this respect a well-defined framework needs to be combined with a well-defined methodology to produce the desired functionality. That is to say that the methodology and framework work hand-in-hand and in congruence to provide the full flexibility for the users. The methodology will use the baseline and the framework which contains the library elements and components necessary for the methodology.

SystemVerilog contains advanced constructs and features that form a solid foundation for creating constrained random verification environment.

Constrained Random and coverage guided Verification

Random test generation allows users to find bugs which they may not have thought about and which maybe hard to detect with directed and manual testing. Functional coverage capabilities built in to the SV environment allows the random simulation to quickly converge and identify areas that have not been tested.

The basis for the methodology to address the growing complexity of the verification in VMM is focused on coverage driven verification, where we would be concentrating and targeting uncovered areas, the main progress is measured using functional coverage metrics.

Verification environment: testbench overview and components

We will focus on testbench as the main vehicle for functional verification. The following diagram shows the overall architecture that a structured testbench environment would take.
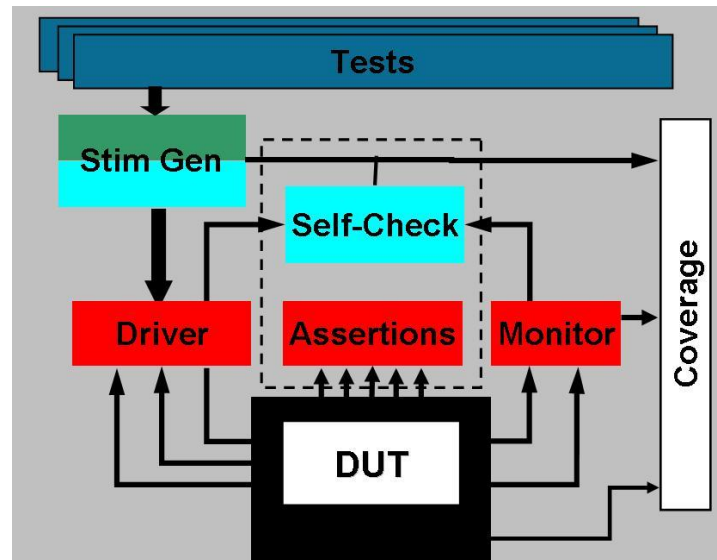
Figure 2: Tesbtench Architecture for Verification

One of the major items to keep in mind is that traditional directed testing suffered from lack of re-usable components, as well as lack of high-level structures which can really be possible with the encapsulation of modular functionality, for example; what ObjectOriented(1) technology in SystemVerilog brings, or the encapsulation of connection to the DUT, such as interface constructs. Refer to **Appendix A** for a review of ObjectOriented programming concepts. Refer to **Appendix B** for a review of SystemVerilog interface definition.

The modular approach and construction of the verification environment, testbench and its components based on the objects and classes will require well balanced and focused connection and communication levels. This can be further abstracted as each component segment is placed in a layer and built in a layered format and architecture. The major benefit is that the communication mechanisms or channels of communication can independently observe and manage the flow of traffic and data between layers and each modular layer can be designed and tested to its appropriate abstraction level.

In summary, the main components needed to build such a testbench environment that generates stimulus, transfers it to the design through drivers; and monitors the response and checks it through the self-checking components as well as provide coverage feedback.

The idea here is to build such a framework for the verification environment that makes it easy for the test writers to issue a few simple commands and create a test suite that runs many cycles of random stimulus. Also the output messages of the tests consistent and systematic so that it helps the engineers with debug of each and every test case. The data that is generated is collected in appropriate coverage blocks for proper analysis and feedback to the test creation process as well as verification closure.

Testbenches also encapsulate configuration object which then drive the stimulus generation at the top-level test control.

For example, let's say that we are trying to verify a color-pen plotter to see if it prints the correct colors with various pen sizes. Although writing directed test cases for this simple example maybe easy, we want to be able to create a top-level environment that allows us to simply modify a set of constraints at the top-level testbench and simulate.

The format for a top-level test program with minimized command to start and run the test suite would then look like:

```
program test;
 // include common library files
 // decalare an object of verif_env type.
 // print a formatted message at the start of the test

  verif_env    env;
  initial
   begin
     env = new();
     env.run();
     // print a formatted message at the end of the test
   end
endprogram
```

In the above program the environment will have been created in such a framework to contain the stimulus generation mechanism, the checking mechanism, the coverage block and configuration of the test and the desired configuration of the design under test.   The design connects to the testbench through a signal interface component which encapsulates all connectivity to the design.

3.0 Introduction to building blocks of VMM

The main idea behind using concepts and base libraries in VMM as discussed previously is to create a consistent and unified style and mechanism for verification project so that it reduces the time required to develop the verification infrastructure, and allows more time for verifying the design.

Layered architecture for testbench and VMM

The architecture that Reference Verification Methodology is based on is a layered architecture for testbench and verification development. Testcases and test suites are implemented on top of a verification environment which helps minimize the number of details of testcase that need to be written.

The basic data and transaction abstraction is provided with class data types. Classes as self-contained components form the foundation of Object-Oriented programming structure. Object-oriented programming is different from the procedural programming. There are three principles behind objects; Encapsulation, Inheritance and Polymorphism. Refer to the appendix C for a basic introduction to object oriented programming concepts.

The object-oriented (OO) class features in SystemVerilog supported in VCS provide the inheritance mechanism to build re-usable and modular verification environments to model complex systems. Using the OO elements of VCS provides the foundation for a layered architecture approach to implementing the verification environment.

Data abstraction in classes and objects benefit users in various ways; above all they encourage the reuse of software components that are developed for specific task. Users will also reduce development time and risk because systematic and modularized development using objects provides more resilience and reduces code size.

Using the features and constructs embedded in SystemVerilog as a basis, the Verification Methodology Manual (VMM) represents a methodology supported by a standard library, consisting of base classes and rules for setting up a consistent and cohesive layered verification environment. The main benefit for users are unification of construction of components and result/information reporting, speed-up in creating layered and re-usable testbenches, allows easy plug-and-play of verification components, and high-level test suites and sequence generations using constrained random mechanisms and functional coverage gathering for design/verification completeness.

In this section we will describe the main components of VMM base classes through examples as well as the concepts and rules embodied in the transaction-level methodology in VMM.
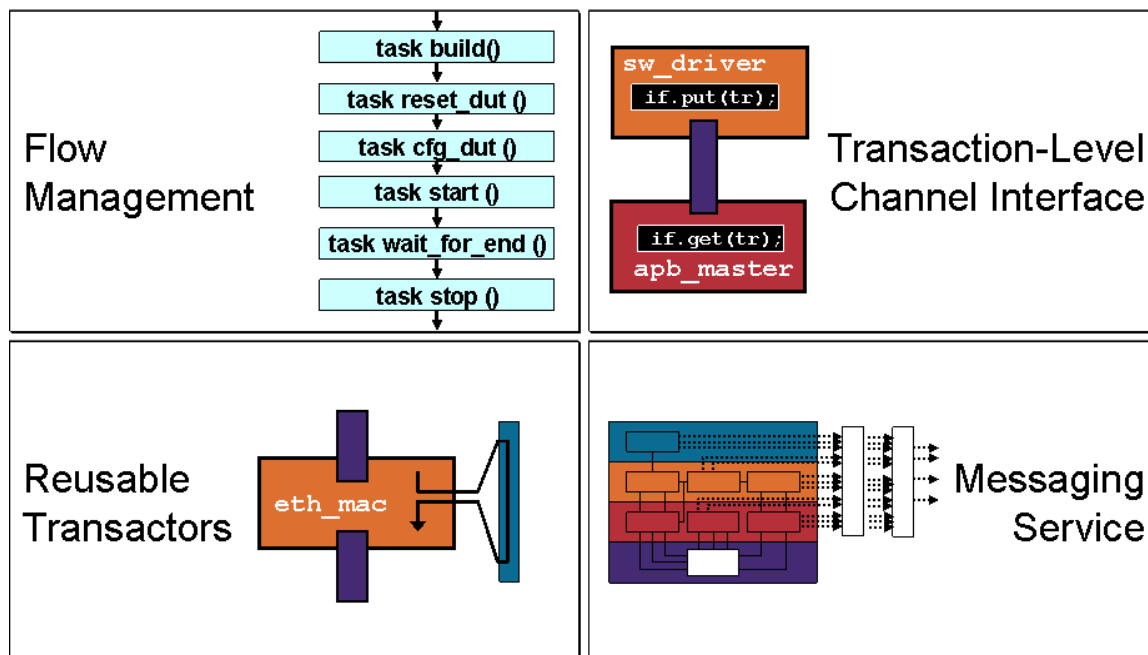
Figure 3: VMM base framework and libraries for verification

The data-transactions as the container for stimulus patterns and transactor components for acting and reacting on those transaction components. The base methodology will set forth a prescribed flow management for the verification environment which unifies flow mechanisms for all components, a transaction level channel interface for means of communications between reusable transactor and set of messaging services to unify the reporting mechanisms.

One Environment, Multiple testcases sets

The VMM will help in designing a flexible testbench infrastructure which allows users to create many more tests without changing the underlying testbench. While you can verify a design using a simple testbench, you would have to create many elaborate tests and continually update the testbench. This latter approach yields more code and reduces the readability and maintainability of your code.

Now, let us introduce the base elements and classes of VMM:
`vmm_log, vmm_data, vmm_xactor, vmm_env,` base classes and callbacks and macros: `vmm_xactor_callbacks` and VMM macros such as `` `vmm_channel `` and `` `vmm_atomic_gen. ``

Messaging and Report formats `vmm_log`

One of the difficulties in debugging traditional testbench results stems from the fact that each segment of the testbench produces its own style of reports and making sense of where and how these were related is usually difficult task. Many types of messages are

produced during a single test simulation run. The **vmm_log** class lets you control the displayed messages and their format. The base class also has capability to allow one to promote and demote the messages as it becomes necessary per simulation run, which is very useful for error testing.

The **vmm_log** is usually instantiated inside a testbench object such as a generator or checker, or in a data object:

```
vmm_log   log;  // declare an instance of vmm_log
log = new("name", "instance");  // instantiate log
```

The **name** string is the name of the class that contains the log, such as "USB Host", or "MAC Frame".   The **instance** string is the name of this instance of the object such as "Generator 1", or "Left side". If there is only a single instance, one can just use the string "class".

In order to create a message you can use the text arrays to place text or formatted text for log reports. The **start_msg**, **text** and **end_msg** methods are used to create from simple to complex messages. For example:

```
  if (log.start_msg(vmm_log::DEBUG_TYP))
    begin
      void'(log.text("Starting test, using text array in log"));
      log.end_msg();
    end
```

However, the easiest way to use a **vmm_log** object is with the macros:

```
        `vmm_fatal   (vmm_log log, string msg);
        `vmm_error   (vmm_log log, string msg);
        `vmm_warning(vmm_log log, string msg);
        `vmm_trace   (vmm_log log, string msg);
        `vmm_debug   (vmm_log log, string msg);
        `vmm_verbose(vmm_log log, string msg);
        `vmm_note    (vmm_log log, string msg);
```

Here are two examples of using the above messages. The first displays a simple string. The second needs to print variable arguments, so it uses **$sformat**, which returns a formatted string:

```
        `vmm_verbose(log, "Checking rcvd byte");
        if (byte != expect)
          begin
           $sformat(msg,"Bad data: 0x%h vs. 0x%h",byte, expect);
            `vmm_error(log, msg);
        end
```

Note that these macros expand to several lines, so surround them with begin-end when used in an if-statement.
Using the `**vmm_debug** macro in this example:

```
`vmm_debug(this.log, ("Buffering TX Frame");
```

Message severity and verbosity:

The `vmm_log` controlled messages allow appropriate simulation handling based on the message types and severity. The various message types and message severerities are summarized in VMM book tables 4-1 and 4-2.

Typical message severities comprise of the enumerated types in `vmm_log` object such as: `FATAL_SEV, ERROR_SEV, WARNING_SEV, NORMAL_SEV, TRACE_SEV, DEBUG_SEV, VERBOSE_SEV.`

Note that using enumerated types will require identification with the `vmm_log`, i.e, `vmm_log::VERBOSE_SEV` to guarantee the behavior matches with the VMM specification.

Similarly message types let the simulation produce and save the relevant messages for the particular simulation run. VMM identifies the following as enumerated message types: `FAILURE _TYP, NOTE_TYP, DEBUG_TYP, TIMING_TYP, XHANDLING_TYP` with several additional message types that can be used by transactors.

The simulation will take appropriate actions depending on the message severity.

Modifying severity and verbosity at run time:

Once the testbench is compiled one can change the severity and verbosity of the message services at run time by using "`+vmm_log_default=<sev>`" runtime command-line option, where "`<sev>`" is the desired minimum severity, one of the following: "error", "warning", "normal", "trace", "debug" or "verbose".

With 'verbose' setting all messages will be reported if triggered. One can also globally force the minimum severity level by using "`+vmm_force_verbosity=<sev>`" runtime command-line option. Note that the equivalent method for the `vmm_log` object is defineds as `set_verbosity` function in the `vmm_log` class.

The following example illustrates the use of the macros and verbosity control.

Note that the VMM library has to be included in the test using: `` `include "vmm.sv" `` in order to be able to use the base classes and features of vmm.

```
program SIMSPLE_TEST;
`include "vmm.sv"
  vmm_log log = new("Test", "FIFO_DRIVER");

initial begin

  if (log.start_msg(vmm_log::DEBUG_TYP)) begin
    void'(log.text("Starting test, using text array in log"));
    void'(log.text("      second line in text array in log"));

    log.end_msg();
  end

 `vmm_report(log, "This is a macro otuptut text vmm_report");
 `vmm_note(log, "This is a macro output text  vmm_note");
```

```
  `vmm_verbose(log, "This is a macro output text  vmm_verbose");
  `vmm_debug(log, "This is a macro output text  vmm_debug");
  `vmm_trace(log, "This is a macro output text  vmm_trace");
  `vmm_warning(log, "This is a macro output text  vmm_warning");
  `vmm_error(log, "This is a macro output text  vmm_error");
  `vmm_fatal(log, "This is a macro output text  vmm_fatal");
  end
endprogram : SIMPLE_TEST
```

Once we compile the test case we can run the simulation with the following command:

```
./simv +vmm_log_default="VERBOSE_SEV"
```

The output result will look like the following:

```
Debug[DEBUG] on Test(FIFO_DRIVER) at                        0:
    Starting test, using text array in log
          second line in text array in log
Debug[REPORT] on Test(FIFO_DRIVER) at                       0:
    This is a macro otuptut text vmm_report
Normal[NOTE] on Test(FIFO_DRIVER) at                        0:
    This is a macro output text  vmm_note
Verbose[DEBUG] on Test(FIFO_DRIVER) at                       0:
    This is a macro output text  vmm_verbose
Debug[DEBUG] on Test(FIFO_DRIVER) at                        0:
    This is a macro output text  vmm_debug
Trace[DEBUG] on Test(FIFO_DRIVER) at                        0:
    This is a macro output text  vmm_trace
WARNING[FAILURE] on Test(FIFO_DRIVER) at                     0:
    This is a macro output text  vmm_warning
!ERROR![FAILURE] on Test(FIFO_DRIVER) at                     0:
    This is a macro output text  vmm_error
*FATAL*[FAILURE] on Test(FIFO_DRIVER) at                     0:
    This is a macro output text  vmm_fatal
$finish at simulation time                     0
          V C S   S i m u l a t i o n   R e p o r t
Time: 0
CPU Time:      0.220 seconds;      Data structure size:   0.0Mb
```

It is fairly easy to search for ERROR or FATAL messages in the log files during post-processing, hence reducing the time to debug of tests and design.

Data and Transaction `vmm_data`

Traditionally the verification and testbench methods were developed as procedures, one per transaction. The result was that code would not be self-contained and would not allow data types to be extended. In addition with procedural code one can not add the new features such as constraints to the data type values.

The transactions should be modeled as objects. This allows data values to exist in a transaction class that can be randomized and further manipulated by the methods that are part and parcel of the same container. For example you can copy, pack or unpack data fields.

Let's start by considering a very trivial example showing the usage of the base class **vmm_data**. The data transactions are extended from **vmm_data** class.

First we declare a class called **pkt_trans** which is a transaction descriptor class, and describes exactly what data we need to send through our system.

Let's assume our packet data consists of two different pieces of information: a 4-bit number "**pen**" and an enumerated type that describes the color. The color can be one of 6 colors described below.

Within the class **pkt_trans** , we have four properties, and four methods.

The first property is **log** which is an instance of the VMM logging base class, **vmm_log** and is used to customize formatting of and filtering the messages coming from this class. The next two properties are specific data information (pen and color). The last property is a constraint named "vanilla" which specifies that the value of "pen" should be less than 8.

The first method is the constructor for the object; all we need here is a call to its parent class constructor **new** method (this is a requirement of Object Oriented features in SystemVerilog).

The second method is the display method that is customized for this specific descriptor class.

There is also **vmm_data::allocate()** method which is used in conjunction with factory pattern usage, discussed in later sections. In this method a new instance of the data transaction is created.

The next two methods are for the purpose of **copying** the data from one object descriptor to another. The only custom parts of these methods are these two lines:

```
cpy.pen = this.pen;
cpy.color = this.color;
```

These are done after a call to the base class **copy_data()** method which takes care of all the base member copying. The copy method creates a duplicate of the main transaction object which is then sent through channels to appropriate transactors. Often when a transaction is generated by means of VMM generic transaction generator or user defined one would need to modify some parameters and send the transaction object but keep the original transaction for checking purposes.

Here is a look at some member properties and methods of **vmm_data** base class. The virtual functions and methods in the base class would be defined to perform the required tasks based on the functionality that the extended classes would have

```
   //Unique identifiers for data model or transaction object instance
   // Note: The following properties will be set by the transactor that
   // operates and handles the data as part of the overall test sequence
   int stream_id;
   int scenario_id;
   int data_id;
   vmm_notify notify;        // notification object

   function new(vmm_log log);

   virtual function string psdiplay(string prefix = "");
   virtual function vmm_data allocate();
   virtual function compare (input vmm_data to, output string diff,
                            input int kind = -1);
   virtual function vmm_data copy(vmm_data to = null);
   // above function will perform a shallow copy, for a full copy
   // of all members and variables copy_data is used
   virtual protected function void copy_data(vmm_data to);
```

Figure 4: Members of vmm_data base class

Now that we have a view of the members of base class `vmm_data`, let's take a look at a simple data transaction.

Here is the code for the data transaction descriptor `pkt_trans`:

```
program test();
`include "vmm.sv"

class pkt_trans extends vmm_data;
  static vmm_log log = new("PKT_TRANS","PKT_TRANS");
  typedef enum { RED , BLUE, GREEN, ORANGE, PINK, YELLOW } color_t;

  //constrainable data members
  rand logic  [3:0]  pen;
  rand color_t color;
  constraint vanilla {
    pen < 8;
  }
// constructor for the pkt_trans class
  function new();
    super.new(log);
  endfunction
// display method for pkt_trans class
  function string psdisplay(string prefix );
     $sformat(psdisplay, "%s : [%d:%s] ", prefix,
this.pen,this.color.name);
  endfunction

// method for allocation in the pkt_trans class
  function vmm_data allocate(vmm_data to = null);
     pkt_trans tr = new;
```

```systemverilog
      return tr;
  endfunction: allocate


// copy method for the pkt_trans class
// Note: the copy_data  method is called within this function to copy
// the actual data values.
  function vmm_data copy(vmm_data to = null);
     pkt_trans cpy;
     if (to == null)
       cpy = new();
     else if (!$cast(cpy, to)) begin
      `vmm_error(this.log, "Cannot copy to non-packet_processor_trans instance");
       copy = null;
       return;
     end
    super.copy_data(cpy);
    cpy.pen = this.pen;
    cpy.color = this.color;
    copy = cpy;
  endfunction: copy


// compare method compares two data objects.
function bit compare(input vmm_data to,
            output string diff, input int kind = -1);

   pkt_trans pkt;
   if ( to == null)
     begin
       `vmm_fatal(log, "Cannot compare to a NULL reference");
       return 0;
    end
   else if (!$cast(pkt, to))
     begin
       `vmm_fatal(log, "Attempting to compare to a non pkt_trans instance");
       return 0;
     end
  if (this.pen != tr.pen) begin
       $sformat(diff, "Pen %0d != %0d", this.pen, pkt.pen);
       return 0;
     end
  if (this.color != tr.color) begin
       $sformat(diff, "Color %0s != %0s", this.color, pkt.color);
       return 0;
     end
  // here the two compare so return a success 1
    return 1;
  endfunction: compare
endclass : pkt_trans


//// Global code segments in the main program file
pkt_trans    tr,tr2;

initial begin
    tr = new();
    tr.randomize();
    `vmm_note(tr.log,(tr.psdisplay("New Random Descriptor") ));
```

```
      tr2 = new();
      tr.copy(tr2);
     `vmm_note(tr2.log,(tr2.psdisplay("Copy of Random Descriptor") ));
   $finish;
end
endprogram : test
```

Once we define a data transaction class (**pkt_trans**) and declare a variable of that type
(`tr` and `tr2`) we need to instantiate those objects before they are used. This process uses
a call to the constructor, `new()` method to actually create storage for the data transaction
object. Once the object is allocated we can call the built-in randomization routines.

In order to print the content of the object we use `` `vmm_note `` macro for message log class,
which prints out the header information as well as the customize version of its contents.

The output looks like:

```
Normal[NOTE] on PKT_TRANS(PKT_TRANS) at                        0:
    New Random Descriptor : [ 1:PINK]
Normal[NOTE] on PKT_TRANS(PKT_TRANS) at                        0:
    Copy of Random Descriptor : [ 1:PINK]
```

Communication means: Channels `vmm_channel`

The channel is used to exchange transactions between its components within the layers of
the testbench structure. For example, transactions can flow from the main generator to
the driver or from a monitor to the checker and scoreboard. The channels are one of
several mechanisms to move the data. The connection between these components is the
vmm_channel. One side produces the transaction, places the transactions into the
channel. The other side which consumes, retrieves the transactions out of the channel,
and acts upon them.

Although a SystemVerilog **mailbox** which acts as a FIFO is used as a common
connection mechanism for many designs, the **vmm_channel** has several advantages over
it, and uses a dynamic queue construct. The **vmm_channel** reduces coding errors since it
will automatically allow type-checking. The flow control mechanism is universal and
does allow for blocking and non-blocking as well as peeking and gathering information
on the content of the channel. It provides for level marking as well to allow more checks
and replication.

Macro for creating vmm_channel objects: `` `vmm_channel ``

The channel can be simply created using the `` `vmm_channel `` macro with the appropriate
data transaction as its argument. The data transaction is extended from `vmm_data`. Once
the data transaction is defined as a class extension of `vmm_data` then it can be used as an
argument for `` `vmm_channel ``. This macro will create channel objects for the data
transaction type.
For example, for the `pkt_trans` transaction, executing the following statement
`` `vmm_channel ``(pkt_trans)

a vmm channel of the type `pkt_trans_channel` class will be automatically created. The channel components of these types must be instantiated and connected between transactors to pass `pkt_trans` data descriptors between them.

The vmm_channel object implementation will provide for appropriate methods to handle placement and removal of transactions in the channel. The flow in the channel is from a producer to consumer.

This macro is expanded and creates 11 methods automatically as shown below.
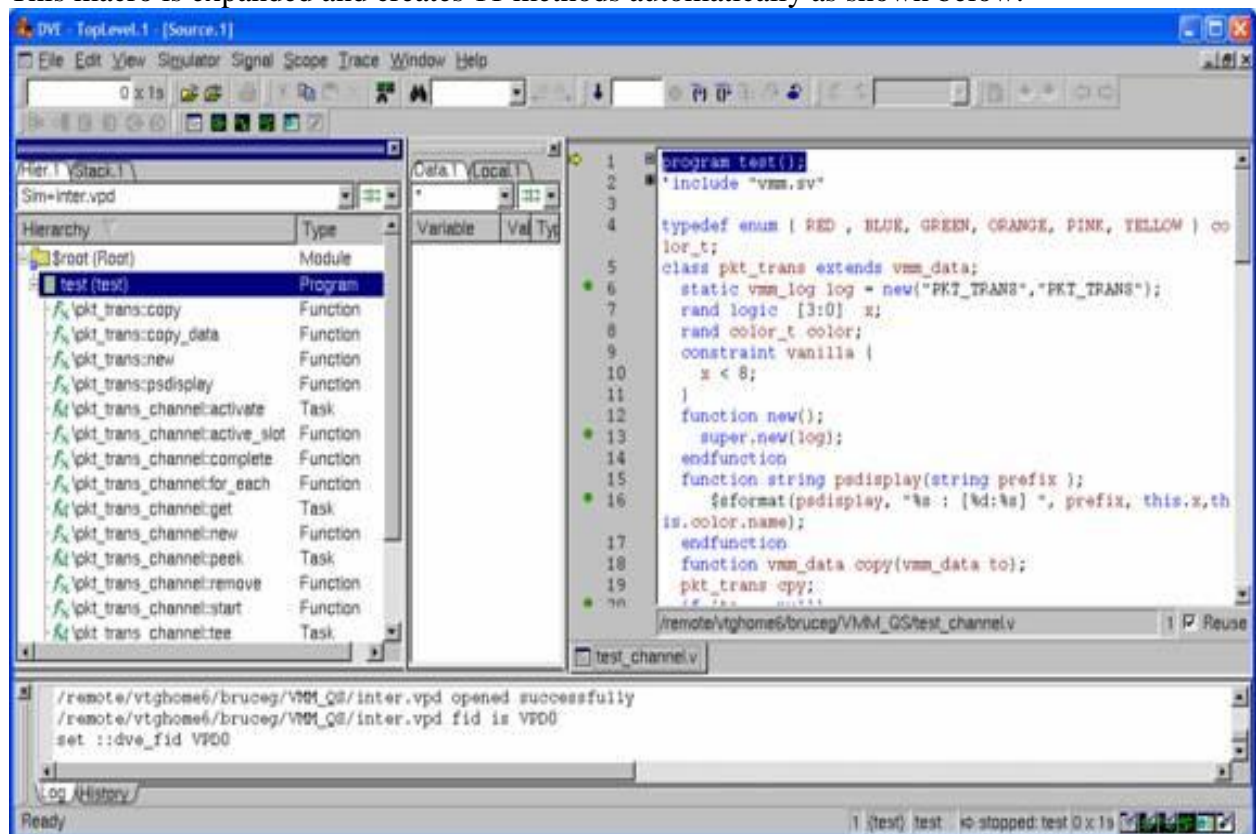
Figure 5: vmm_channel macro expansion result

The main methods are `vmm_channell::new()` constructor for the channel, `vmm_channell::put` and `vmm_channel::get` which allow placing and removing the objects. Note that the get() method is a blocking method and will wait until there is an element in the channel.

The channel instantiation will require a name and an instance for the channel in the call to the constructor.

Let's take a look at the pkt_trans example illustrating the usage of the base class for a channel, `vmm_channel`:

```
program test();
   `include "vmm.sv"
```

```
  class pkt_trans extends vmm_data;
  // Same code as above
  endclass : pkt_trans
//
// macro for creating the channels for pkt_trans transactions
// will create a class description for  transaction_channel
// where transaction is the name of descriptor passed
`vmm_channel(pkt_trans)

// will create and define a base class type:  pkt_trans_channel

//// Global segments
///  declare channel object of the type  pkt_trans_channel
pkt_trans_channel    chan;
pkt_trans            tr,tr2;

initial begin
// instantiate the channel object
// 1000 is the fill-level, number of transaction descriptors
  chan = new("pkt_transfer","first instance",1000);
end

// two blocks running in parallel, one produces the pkt_trans data
// the other consumes, the flow is through the pkt_trans_channel
// producer block
initial begin
  `vmm_note(tr.log, "Randomize transaction sequence  vmm_note");
  repeat(10) begin
   #10;
    tr = new();
    tr.randomize();
    $display(tr.psdisplay("GENERATOR put transaction data") );
    chan.put(tr);
  end
  $finish;
end
// consumer block
initial forever #30 begin
      tr2 = new();
      chan.get(tr2);
    $display(tr2.psdisplay("..CHANNEL get transaction data"));
end
endprogram : test
```

Once the channel class is auto-created, we declare an instance called **chan**. The main usage model is for producer or generator to create a transaction and use the channel `put` method to place the tranasaciton in the channel and the consumer or the channel driver to use the `get` method to retrieve the data.

In the consumer the data transaction object is instantiated (namely tr2) and content of the channel is copied to this object by the channel get method. Here is a result with the following command

```
    simv +vmm_log_default="NORMAL_SEV"
```

The following is the results.

```
 Normal[NOTE] on PKT_TRANS(PKT_TRANS) at                          0:
      Randomize transaction sequence  vmm_note
 GENERATOR put transaction data : [ 1:PINK]
 GENERATOR put transaction data : [ 2:YELLOW]
 ..CHANNEL get transaction data : [ 1:PINK]
 GENERATOR put transaction data : [ 6:YELLOW]
 GENERATOR put transaction data : [ 1:YELLOW]
 GENERATOR put transaction data : [ 4:YELLOW]
 ..CHANNEL get transaction data : [ 2:YELLOW]
 GENERATOR put transaction data : [ 3:GREEN]
 GENERATOR put transaction data : [ 6:BLUE]
 GENERATOR put transaction data : [ 6:GREEN]
 ..CHANNEL get transaction data : [ 6:YELLOW]
 GENERATOR put transaction data : [ 7:RED]
 GENERATOR put transaction data : [ 5:PINK]
 $finish at simulation time                 100
           V C S   S i m u l a t i o n   R e p o r t
```

As it is shown the data transaction is placed into the channel and retrieved in the sequence they were placed.

The vmm_channel provides various method to operate the channel object efficiently, method prototypes are described in VMM book in standard library specification, pages 387 -397. For example if one needs to change the fill level of the channel, reconfigure method will modify the fill level. One has to be careful,the reconfiguration lowers the fill level threads currently blocking on vmm_channel:: put() will unblock.

```
// reconfigure the channel to 500 transactions.
  chan.reconfigure(500);
```

Note that the **put** method is blocking method, i.e, if the channel has reached its full level any put will wait until there is open slot available in the channel.

The vmm_channel::sneak() is similar to put however it is non-blocking and adds transaction descriptors to the tail of the channel. We can also use vmm_channel::peek() method to get a refrence to the transaction descriptor in the channel for next retreival withouth physically removing it from the channel. This is useful in case one requires a test to see if this transaction object belongs to the channel or not.

Transactors `vmm_xactor`

Transactor is a term used to define and identify component of verification that acts upon or executes and observes transactions over various paths and cycle time in dynamic verificaiton environments. Transactors are implemented as class objects which allow definition of randomization and constraints. Transactors perform on differnt kinds of transaction generation and handling. Transactors can be generally categorized into Active, Reactive and Passive transactors. Master/Slave devices are active/re-active transactors, where monitors are usually defined as Passive transactor models. The base transactor in VMM is **vmm_xactor** and emobdies the methods and mechanisms needed to allow intimate interaction with the overall execution steps in the testbench from top test layer to the signal DUT connection. The **vmm_xactor** comprises the main notification services needed for observability and control of sequences of activities within a transactor

or all transactors. The logging services comes in as part of vmm_log embedded in each transactor messaging. In the next sections we will also learn more about callback mechanisms which allow insertion or modification to the regular tranactor actions.

Basic transactor methods

Here is a basic transactor, with the minimum set of pre-defined methods which must be defined in the extended class for appropriate handling of data or transaction object at each layer of testbench.

`vmm_xactor::start_xactor(),`
`vmm_xactor::stop_xactor()  and`
`vmm_xactor::main()`

The `vmm_xactor::main()` is spawned when `vmm_xactor::start_xactor()` task is called from upper layers. It is also stopped when the `vmm_xactor::stop_xactor()` is called. There are other methods that can be defined for further control of activity in the transactor: `vmm_xactor::reset_xactor()`.

The transactor stop mechanism can be further fine-grained by using the following `vmm_xactor` methods: `wait_if_stopped()` and `wait_if_stopped_or_empty();` We shall return to these methods later.

For example, `vmm_xactor::start_xactor()` starts the virtual method `main()` − there is no need to add this in your implementation. Lets take a look at a transactor `pkt_driver` which is extended from `vmm_xactor` for use with `pkt_trans` transaction.

```
class pkt_driver extends vmm_xactor;
   // declare property members of the pkt_driver
   // such as interfaces, channels, etc.
   pkt_trans_channel        chan_in;
   function new( string inst,
              int stream_id = -1,
              pkt_trans_channel     chan_in);
   // The chan_in will be used by up-stream transactor or
   // generator to pass pkt_trans data objects to pkt_driver
   // transactor.
   // first call the constructor of vmm_xactor base class.
   // pass the name and instance name used in vmm_log
      super.new("PKT_driver xactor", inst, stream_id);
      // check for channel
      if (chan_in == null)
        this.chan_in = new("pkt_driver_channel", "channel");
       else this.chan_in = chan_in;
    endfunction: new

   // Declre tasks and functions (methods)
   //start_xactor starts the execution threads and calls the main
   // task in the transactor
   virtual function void start_xactor();
      super.start_xactor();
      // any specific code related to pkt_driver
   endtask : start_xactor
```

```
 //stops execution threads after currently executing
 //transaction had completed. Takes effect at next call
 //to ::wait_if_stopped()
virtual function void stop_xactor();
    super.stop_xactor();
    // any specific code related to pkt_driver
endtask : stop_xactor

//resets the xactor's state and execution threads
virtual function void reset_xactor(reset_e rst_typ = SOFT_RST);
    super.reset_xactor(rst_typ);
    // specific reset type can be placed here.
    // channels have to be flushed as well, for example
    this.chan_in.flush();
 endtask : reset_xactor

// Remember that upper layer, the enviornment calling
// start_xactor, each vmm_xactor::start_xactor gets called
// which then calls the main task.
virtual protected task main();
 // the first executible statement in main base main task.
    super.main();
    // appropriate code for main functionality of the pkt_driver
    // transactor such as check the channel for pkt_trans and
    //
    forever begin : main_task_loop
      pkt_trans tr;
      // per rule 4-121, we can get the handle to the next
      // data transaction descriptor in the channel, remember
      // peek blocks until one is available
      this.chan_in.peek(tr);
      `vmm_trace(this.log, "Starting transaction for pkt_drive");
      // now process the transaction object,
      // code goes here
     `vmm_trace(this.log, {"This is Pen and its Color", tr.psdisplay("")});
     // now we can unblock the channel
      this.chan_in.get(tr);
    endtask : main
  endclass : pkt_driver
```

As noted transactors act upon data transactions which move about the testbench through channels from layer to layer. The command layer transactors process the data and stimulate the pins and signals of the design as well as monitor the output of the device. The upper layer transactors form the data transactions formats that represent the abstraction of test stimuli at each layer. When all placed in the proper area of verification environmnet embodied in extension of **vmm_env** and connected with appropriate channels they provide the high level of controlability and observability required to manage the testbench and test sequences.

Generators: VMM Atomic Generator `vmm_atomic_gen`

Once the data patterns are defined and declared as transactions the testbench environment would need to generate a set or series of various transaction descriptors or objects which would need to be propagated and applied to the design under test. This process, the stimulus generation process is usually done at the higher layers; it will take direction from the test layer and start generating stimulus accordingly.

The simplest generator would autonomously act on transaction object and create them in sequence. These individual transaction objects are randomized and sent through the channel in the same way they were created, one by one, hence forming an atomic generation mechanism.

The VMM environment provides a macro `vmm_atomic_gen` to automatically create a class transactor which is extended from `vmm_xactor` for atomic generation of the specified transactions. The macro will also create a callback class which is extended from `vmm_xactor_callbacks`. For a discussion on the callbacks and how they work refer to the section on Callbacks within this chapter.

The `` `vmm_atomic_gen(class name, "Calls Description") `` defines an atomic generator class named `<class_name>_atomic_gen`.

For our `pkt_trans` data transaction the **vmm_xactor** extended generator class is named **pkt_trans_atomic_gen** and will contain output channel, handle for randomized transaction, a stop_count value **stop_after_n_insts** for generator to stop creating transaction objects.

The callbacks class for atomic generator is named **pkt_trans_atomic_gen_callbacks** which allows callback methods to be integrated with the atomic generator main and inject tasks.
Note that the data transaction class for the generator must be derived from `vmm_data` class and the **<class_name>_channel** class must exist. Therefore one would have the directives: `` `vmm_channel_<class_name> `` and atomic generation macro following each other to guarantee the above is done properly

Example of atomic generator

The macros are used to define the channel and the atomic generator.

```
`vmm_channel(pkt_trans)

`vmm_atomic_gen(pkt_trans, "packet_transaction_genertaor")
```

Just like the channel class, a variable of this type is declared and then instantiated, in this case `gen`.

```
    // Global segments
    //  declare channel object of the type  pkt_trans_channel
    //  driver xactor, atomic generator, test config
     pkt_trans_channel       chan;
     pkt_driver              driver;
     pkt_trans_atomic_gen    gen;
```

The generator will be part of the environment class which will be described in the next segment. The transactors, generators, configuration objects and other components will be instantiated in the main environment object and used.

Here is a look at the basic properties and methods that the generator contains after the macro is executed. The VMM_ATOMIC_GEN is described in Appendix A of VMM book. You can also create individual generator by extending a `vmm_xactor` and incorporating the properties and methods that are required by VMM.

```
class pkt_trans_atomic_gen extends vmm_xactor;
   integer stop_after_n_insts;
   integer GENERATED;
   integer DONE;

   pkt_trans            randomized_obj; //used for factory
   pkt_trans_channel    out_chan;       //gets the copy of trans

   local integer        scenario_count;
   local integer        obj_count;

   extern funcion new(string        instance,
                   integer      stream_id = -1,
                   pkt_trans_channel      out_chan  = null);
   extern virtual task inject(pkt_trans obj, var bit    dropped);
   extern virtual function void reset_xactor(integer rst_type = 0);
   extern virtual protected task main();
 endclass : pkt_trans_atomic_gen
```

We will not go into details of atomic_generator callbacks in this introductory section but suffice it to say that this will be used in the `inject` task for the generator. Here is the definition for the **pkt_trans_atomic_gen_callbacks** class for atomic generator which allows callback methods to be integrated with the atomic generator main and inject tasks.

```
 class pkt_trans_atomic_gen_callbacks extends vmm_xactor_callbacks;
    virtual task post_inst_gen(pkt_trans_atomic_gen gen,
                            pkt_trans        obj,
                            var bit          drop);
    endtask
 endclass: pkt_trans_atomic_gen_callbacks
```

Now let us look at one implementation for the generator methods that follows the VMM rules and recommendations.

```
 function pkt_trans_atomic_gen::new(string   instance,
                              integer  stream_id = -1,
                              pkt_trans_channel out_chan  = null);
   super.new("pkt_trans_atomic_gen", instance, stream_id);

   if (out_chan == null)
     out_chan = new("pkt_trans_atomic_gen output channel", instance);
   this.out_chan = out_chan;
   this.scenario_count = 0;
   this.obj_count = 0;
```

```
    this.stop_after_n_insts = 0;

    this.GENERATED = this.notify.configure(*, this.notify.ONE_SHOT);
    this.DONE      = this.notify.configure(*, this.notify.ON_OFF);

    this.randomized_obj = new;
endfunction: new
```

The constructor initializes the properties and instantiates the `randomized_obj` object which is used to generate `pkt_trans` data transactions in the main task. The two notification property variables, **GENERATED and DONE** are also configured for appropriate event notification behavior for the generator.

```
function void pkt_trans_atomic_gen::reset_xactor(integer rst_type = 0);
    super.reset_xactor(rst_type);
    this.obj_count = 0;
endfunction : reset_xactor

task pkt_trans_atomic_gen::main();
    bit dropped;
    fork
        super.main();
    join none
    this.obj_count = 0;
// will generate the transactions according to the integer specified
// for stop_after_n_insts
// updates the stream_id, and counts for each pkt_trans object
    while (this.stop_after_n_insts <= 0 ||
            this.obj_count < this.stop_after_n_insts) begin
        this.wait_if_stopped();
        this.randomized_obj.stream_id   = this.stream_id;
        this.randomized_obj.scenario_id = this.scenario_count;
        this.randomized_obj.object_id   = this.obj_count;

        if (!this.randomized_obj.randomize()) begin
            `vmm_error(this.log, "Cannot randomize atomic instance");
            continue;
        end
        // make a copy of the randomized_obj and send the copy to the
        // output channel
        begin
            pkt_trans    obj;
            cast_assign(obj, this.randomized_obj.copy());
            void = this.inject(obj, dropped);
        end
    end
    this.notify.indicate(this.DONE);
    this.scenario_count++;
endtask: main

  task pkt_trans_atomic_gen::inject(pkt_trans obj,
                                    var bit    dropped);
    dropped = 0;
    this.obj_count++;
```

```
    // before placing the copied object in the output channel
    // check for any callbacks to modify or drop the
    // transaction
       this.post_inst_gen_t(obj, dropped);
       `vmm_callback(pkt_trans_atomic_gen_callbacks,
                 post_inst_gen_t(this, obj, dropped));
        if (!dropped) begin
           this.notify.indicate(this.GENERATED, obj);
           this.out_chan.put_t(obj);
        end
  endtask: inject
```

The generators will create data transaction objects according to the test configuration and set of constraints specified by the test routine and pass them through channels to the driver transactors. The following shows an example of channel and generator instantiation for the pkt_trans example.

```
    chan =      new("pkt_transfer","first instance",1000);
    gen =       new("pkt_gen",0,chan);
    driver =    new("pkt_driver",0,chan);
```

As mentioned these will be part of the build task of the verification environment object.

Verification Environment `vmm_env`

The testbench traverses many phases of execution steps, from initialization and reset to stimulus generation and result reporting. The base class **vmm_env** helps manage these steps and ensures that all steps execute in the proper order that were described.
The vmm_env class divides a simulation into the following steps, with corresponding methods:

- **gen_cfg()** – Randomize test configuration descriptor
- **build()** – Allocate and connect test environment components
- **reset_dut()** – Reset the DUT
- **cfg_dut()** – Download test configuration into the DUT
- **start()** – Start components
- **wait_for_end()** – End of test detection
- **stop()** – Stop data generators and wait for DUT to complete its tasks
- **cleanup()** – Check recorded statistics and sweep for lost data
- **report()** – Print final report

The testbench environment extends vmm_env class. For the environment object the top level method is **run()** which keeps track of executed steps, and, when called, runs the remaining ones. For example, the following program runs all steps automatically:

```
    program test;
     `include "vmm.sv"
      class verif_env extends vmm_env;
       // include transactor declaration, channel declaration,
       // configuration descriptor declaration and define the
```

```
     // methods for our specific DUT protocol
   endclass: verif_env
    // decalare an object of verif_env type.
   verif_env   env;
   vmm_log log = new("toptest", "main_log");
   initial begin
      `vmm_note(log, "Test program is starting");
      env = new();
      env.run();
      `vmm_note(log, "Test program ends");
   end
 endprogram : test
```

In this example the class **verif_env** extends **vmm_env**. Once the method **run()** is called it will call all the steps which have not yet been run.
Now let's take a closer look at each of the main segments of the verification environment methods.

Test and Device Configuration `gen_cfg()`
The device under test has a varied functionality which results in multitude of device configurations and formats. Each of these configurations needs testing and validation. Also test conditions which satisfy the environment that the design operates in are many and varied. In order to manage various test and device configuration vmm_env requires a test configuration descriptor object which handles test conditions. These test conditions can identify number of transactions that are to be generated and targeted to the device under test, gaps between transactions, or number of different scenarios each particular device configuration should be tested with.
The vmm_env::gen_cfg() step allows for setting and randomizing the configuration parameters per each test and device.
The following code segment runs the first step, makes a modification to the configuration after it is randomized (calling gen_cfg() will do the randomization, you will code this in the gen_cfg of verif_env class) and then completes the test:

```
   program test;
     // verif_env is defined as extension to vmm_env as above
     initial begin
       verif_env my_env;
       env = new();
       env.gen_cfg();              // Create rand config
       env.cfg.trans_cnt = 1;    // Only run once
       env.run();                  // Perform all other steps
     end
   endprogram : test
```

Note that in he verif_env:: gen_cfg() method there is a call to super.gen_cfg() as vmm_env base class needs to update the base parameters.

First we would need to define a class as configuration descriptor, in this case we call it **t_cfg** class.

```
   class t_cfg;
       // How many transactions to generate before test ends?
       rand int trans_cnt;
```

```
      constraint basic {
        trans_cnt > 9;
        trans_cnt < 10000000;
        trans_cnt == 10;
      }
    endclass: t_cfg
```

The following is the sample code that `gen_cfg()` could contain in the `verif_env` class:

```
///////////////////////////////////////////////////////
// gen_cfg() - Generate a randomized testbench configuration
///////////////////////////////////////////////////////
function void verif_env::gen_cfg() ;
  super.gen_cfg() ;
  if (cfg.randomize() == 0)
    `vmm_fatal(log, "Failed to randomize testbench configuration");
  $sformat(msg, ("cfg.trans_cnt = %d", cfg.trans_cnt));
  `vmm_note(log, msg);
endfunction : gen_cfg
```

Important notes: The default testcase configuration descriptor instance, for example `cfg` above, is the only object that is instantiated in the environment constructor. All other components are instantiated in the build method discussed below. These follow the rules set in VMM: rule 4-34 and 4-35.

The implementation of verification methods should call their base implementation first. These would be accomplished by calling `super.method_name()` as the first executable statement inside the body of the method. (VMM rule 4-31).

The build phase: `build()`

Once the configuration is setup the next step is to build the verification environment. The transactors, channels and other components declared within the environment need instantiation.

```
function void verif_env::build() ;
  super.build() ;

  // instantiate channels, transactors, generators, etc.
  // monitors, scoreboards
   chan =     new("pkt_transfer","first instance",1000);
   gen =      new("pkt_gen",0,chan);
   driver =   new("pkt_driver",0,chan);
   // set any number of transactions in the generator to stop
   gen.stop_after_n_insts = cfg.trans_cnt;

endfunction: build
```

So far we have not delved into the signal connection between testbench components, i.e, command transactors and the device under test. The actual connection is done through the device interface. Interface in SystemVerilog can bundle signals and wires together for connection of modules and make it easier to pass them to the testbench and design.

Reset and Configure DUT `reset_dut(),cfg_dut()`

The next logical steps are to reset the DUT and set the appropriate device configuration, such as register sizes, input output transmit receive mode of the device.

Once again, the `reset_dut()` method would make a call to `super.reset_dut()`, as does the `cfg_dut()` method to the `super.cfg_dut()`. With these two methods the design will be set in the desired configuration, this should match what the test configuration expects.

Start the components phase: `start()`

The `start()` task once called will start the components embedded in the verification environment. Note that the transactors should not start before the design is reset and configured properly. If started earlier in this case these transactors would provide false stimulus to the design as well as monitor inconsistent behavior since the design is not set in its proper test form.

```
virtual task verif_env::start();
  super.start();
  // call all transactors  start_xactor() methods in this task
  // drivers, monitors, scoreboards, generators.
endtask: start
```

Waiting for end phase: `wait_for_end()`

Now that activity has started in the testbench we need to have a mechanism to wait for the end of the test. The actual time tests end varies according to the random conditions and setup that is part of each test configuration and generation methods. Hence `wait_for_end()` waits for an indication that the test has reached its completion stage. At this point it can notify that wait_for is done and the environment is ready to stop all transactors and gather results. In the generator the notify object is used to indicate that the wait_for_end is complete. The following is an example for wait_for_end task:

```
virtual task verify_env::wait_for_end();
  super.wait_for_end();
  // wait for the generator ending
  // it will be added when the generator is defined
  gen.notify.wait_for(pkt_trans_atomic_gen::DONE);
  // wait for some cycles to let dut settle
endtask: wait_for_end
```

Stop and cleanup phase: `stop(),cleanup()`

This method stops all the components of the verification environment and hence would want to terminate the simulation gracefully and with no residual effect. The `cleanup()` method should perform necessary tasks for the graceful end of the simulation such, for example letting the DUT flush out all buffered data. Also in this cleanup stage one would flush all channels.

Report phase: `report()`

Finally in the logical flow set forth by the `vmm_env` the last phase would report the final success or failure of the test and close all opened files.

In the step by step flow, if at any point the above methods are not explicitly called, they would be implicitly called by each method that follows it. In this case calling `run()` will start to call the previous methods which will in turn call its previous method, starting at `gen_cfg()` method and ending with `report()` method.

```
program test();
/////////////////////////////////////////////
//  the header file for vmm base class library
/////////////////////////////////////////////
 `include "vmm.sv"
typedef enum { RED , BLUE, GREEN, ORANGE, PINK, YELLOW } color_t;

/////////////////////////////////////////////////
// pkt_trans  A transaction class for data
/////////////////////////////////////////////////
class pkt_trans extends vmm_data;
  //as previously defined
endclass: pkt_trans

/////////////////////////////////////////////////
// macro for creating the channels for pkt_trans transactions
/////////////////////////////////////////////////
 `vmm_channel(pkt_trans)
 `vmm_atomic_gen(pkt_trans,"pkt transaction generator")

/////////////////////////////////////////////////
//  pkt_driver   A transactor class for pkt_trans data transaction
/////////////////////////////////////////////////
class pkt_driver extends vmm_xactor;
 // as defined in previous segment
 endclass : pkt_driver

/////////////////////////////////////////////////
// test configuration class with basic constraints
/////////////////////////////////////////////////
class test_cfg;
 // How many transactions to generate before test ends?
  rand int trans_cnt;

  constraint basic {
    trans_cnt > 9;
    trans_cnt < 10000000;
    trans_cnt == 10;
  }
endclass: test_cfg

/////////////////////////////////////////////////
// verification environment class definition
/////////////////////////////////////////////////
class verif_env extends vmm_env;
 //  declare channel object of the type  pkt_trans_channel
 //  driver xactor, atomic generator, test config
 pkt_trans_channel       chan;
 pkt_driver              driver;
 pkt_trans_atomic_gen    gen;
 test_cfg                cfg;
```

```
  pkt_trans                tr,tr2;

    function new();
       super.new();
       this.cfg = new();
    endfunction: new

    virtual function void gen_cfg();
       super.gen_cfg();
       if (cfg.randomize() ==0)
         `vmm_fatal(log, "Failed to randomize testbench configuration");
         `vmm_note(this.log, "cfg, with trans_cnt ");
    endfunction: gen_cfg

    virtual task run();
       super.run();
    endtask: run

    virtual function void build();
       super.build();
       chan =     new("pkt_transfer","first instance",1000);
       gen =      new("pkt_gen",0,chan);
       driver =   new("pkt_driver",0,chan);
       gen.stop_after_n_insts = cfg.trans_cnt;
    endfunction: build

    virtual task start();
       super.start();
       driver.start_xactor();
       gen.start_xactor();
    endtask: start

    virtual task stop();
       super.stop();
       gen.stop_xactor();
       driver.stop_xactor();
    endtask: stop

    virtual task wait_for_end();
       super.wait_for_end();
      // wait for the generator ending
      // it will be added when the generator is defined
      gen.notify.wait_for(pkt_trans_atomic_gen::DONE);
      // for now we can wait for #150.
      #150;
    endtask: wait_for_end

   virtual task report();
       super.report();
   endtask: report

    virtual task reset_dut();
       super.reset_dut();
    endtask: reset_dut

 endclass: verif_env
```

```
 //declare a variable of the type verification environment
 verif_env     env;
 pkt_trans     tr,tr2;

 initial begin
   env = new();
   env.build();
   env.run();
 end

 endprogram: test
```

The environment setup plays an important role in the structured testbench. It provides a controllable and observable path for all major components in the testbench thereby a high performance verification system can be properly built.

Factory pattern

Various random stimuli is created through the generator and applied to the DUT. It is very desirable to have control over this random stimulus from the environment and test level. In the atomic generator the data transaction which is named randomized_obj is treated as factory pattern, and it is a public member of the generator object. Hence it can be assigned from the upper layers.

This factory pattern data transaction is allocated or copied using `allocate()` or `copy()` methods in the generator transactor rather than being instantiated in the generator's constructor routine. This allows control from outside of the generator, i.e, one can extend the base data transaction and apply it to the factory pattern, hence creating new sets of constraints for the test case.

```
 program test();
  //same as in the previous sections

  // define an extension to the base data transaction
  //////////////////////////////////////////////////////
  // my_pkt_trans  extends the base pkt_trans
  //////////////////////////////////////////////////////
  class my_pkt_trans extends pkt_trans;
     constraint test_02 {
       pen inside { 4,5,6};
       color inside { RED, BLUE, GREEN};
     }
  endclass: my_pkt_trans

  verif_env      env;
  my_pkt_trans   tr;

  initial begin
   env = new();
   env.build();
   //  set the generator factory pattern, the randomized_obj
   //  to this extended transaction object.
```

```
    //  We are replacing the factory pattern instance with
    //  this extension
    begin
      tr = new();
      env.gen.randomized_obj = tr;
    end

    env.run();
  end
endprogram : test
```

One can also replace for example test configuration class in the environment.

Self-checking and Functional Coverage

Many random transactions get generated for each test case the result of which must be checked for validity against expected behavior of the DUT. A self-checking component is encapsulated in a class which contains the specific routine to check the data output from the DUT. On the other hand protocol checking can verify the compliance to the design protocol specification.

The functional coverage routines and methods will allow users to gather information about the stimulus as well as the standard protocols. These coverage definitions can be encapsulated in a coverage class as shown below.   However, including data coverage recording in the response checkers also allows for automatic "scoreboarding" to be done, which ensures that for all of the data combinations in the input, the appropriate output combinations were received. It also allows the user to analyze the coverage data and evaluate whether the right input stimulus combinations were generated to verify all possible output conditions.

The scorebaording techniques will use data structure that holds the expected response. Usually a queue can be used to store the ordered responses from the DUT. If there are independent streams of output then multiple queues are needed to store the values to be checked against expected results.

In order to integrate self-checking and functional coverage components with the stimulus generation and monitor transactors the callback methods need be employed which we will cover in the next segment.

Callbacks

It becomes difficult to incorporate new mechanisms and routines once an environment has been designed and its components have been developed unless entry points were reserved for such purposes.

There are design patterns which form the basis of these additions, i.e., they get called within the main routine of the software flow, in this case within the main flow of the verification components. Take for example the driver transactor that needs to perform a write or read. If we needed to inject an error condition to the device under test how would one go about inserting that error? The driver routine only takes care of the 'correct' behavior. Hence you would need to allow changes to be made before the driver takes the

final action or at any time along the flow. Also you may want to have other components take different actions depending on the outcome of the driver actions. For example after the write is complete one needs to inform the checker and the coverage block to gather the information based on the data transaction stimulus that was just driven to the design.

It is this access mechanism that is provided by callback routines, simply as design patterns that get registered in the main routines to be called back at certain designated points.

The transactor callback routine or what is known as callback façade is implemented as an extension to the **vmm_xactor_callbacks** class. Once a callback class is defined, a declaration is usually made in the environment. Once the object is instantiated it is ready to be connected to any transactor, the callback object is said to be registered to the transactor. The registration is like letting each particular transactor know that there are some extra pieces of code that can be called in the transactor flow. The callbacks are usually registered by the virtual append method associated with the **vmm_xactor_callbacks** class. Multiple callback routines can be appended to a particular transactor object.

Callbacks, just like other objects in the testbench require **definition** (process of extending from vmm_xactor_callbacks), **declaration** and **instantiation**. Once an instance of callback object is created then it needs to be **registered** to the specific transactor object. The actual calls are made by **invoking** the callbacks in the transactor.

The following VMM base classes have callbacks associated with them:

vmm_xactor, vmm_log, vmm_atomic_gen, vmm_scenario_gen.

We will concentrate on the vmm_xactor callbacks in these segments. Each base class callbacks class for the above is defined as:

vmm_log_callbacks, vmm_xactor_callbacks, vmm_atomic_gen_callbacks and **vmm_scenario_gen_callbacks**.

The callbacks can be used for functional coverage definition and gathering, incorporation into scoreboards and data transaction injection as discussed above.

Transactor callbacks invocation with macro `vmm_callback()

The transactor callback class will contain methods (tasks or void functions, refer to 4-160) which will be executed once the registered callbacks are called with appropriate macro for callbacks: `vmm_callback()`. This is stated in rule VMM 4-163.

```
//////////////////////////////////////////////////
// transactor callback class for pkt_driver
//////////////////////////////////////////////////
virtual class pkt_driver_callbacks extends vmm_xactor_callbacks;
   // Callbacks before a transaction is started
   virtual task master_pre_tx(pkt_driver       xactor,
                              ref pkt_trans    trans,
                              ref bit          drop);
   endtask : master_pre_tx

   // Callback after a transaction is completed
   virtual task master_post_tx(pkt_driver    xactor,
```

```
                                           pkt_trans      trans);
    endtask : master_post_tx

endclass: pkt_driver_callbacks
```

The above transactor callback façade for `pkt_driver` is referenced in the transactor `pkt_driver`. In order to create varied application that uses the callbacks, one needs to extend this callback and create set of specifically targeted callbacks. For example `pkt_driver_callbacks` class can be extended further for functional coverage callbacks and scoreboarding.

The prototype call for `vmm_callback macro is as follows:

```
`vmm_callback(callback_class_name, methods(arguments))
```

Note this macro is shorthand for the following code that gets inserted in the main transactor code:

```
  foreach (this.callbacks[i]) begin
     fifo_master_callbacks cb;
     if ($cast_assign(cb, this.callbacks[i])) continue;
     cb.ptr_tr(this, tr, drop);
  end
```

The `vmm_callback` macro will process whatever `pkt_driver_callbacks` or its extension that have been registered to the `pkt_driver_xactor`. For example, we can extend the above to create a callbacks class which contains the callback methods as well as the covergroups for coverage data gathering.

```
//////////////////////////////////////////////////////
// This is for coverage of pkt_driver transactor  callback class
//////////////////////////////////////////////////////
class pkt_driver_cov_callbacks extends pkt_driver_callbacks;
  local pkt_trans tr ;

  covergroup pkt_trans_cov;
     PEN: coverpoint tr.pen {
           bins LOW = { [0:2]};
           bins MED = { [3:5]};
           bins HIGH = {[6:7]};
         }
     COLOR: coverpoint tr.color {
           bins LOW = { [RED:BLUE]};
           bins MED = { [GREEN:ORANGE]};
           bins HIGH = { [PINK:YELLOW]};
         }
     PENxCOLOR: cross PEN,COLOR ;
  endgroup

  // Callbacks before a transaction is started
  virtual task master_pre_tx(pkt_driver     xactor,
                             ref pkt_trans trans,
                             ref bit         drop);
   // Empty
  endtask
```

```
  // Callback after a transaction is completed
  virtual task master_post_tx(pkt_driver xactor,
                              pkt_trans  trans);

    tr = trans ;                    // Save a handle to the transaction
    pkt_trans_cov.sample();         // Sample Coverage

  endtask

  function new();
    pkt_trans_cov = new();
  endfunction: new

endclass: pkt_driver_cov_callbacks
```

Now that the callback façade classes have been defined we can look at the pkt_driver transactor and locate the precise entry point for the callbacks invocation. In the main task code we can identify locations for pre and post transaction invocation of callback methods.

```
  virtual protected task pkt_driver::main();
    bit drop;
    super.main();
    // appropriate code for main functionality of the pkt_driver
    // transactor such as check the channel for pkt_trans and
    forever begin : main_task_loop
      pkt_trans    tr;
      // per rule 4-121, we can get the handle to the next
      // data transaction descriptor in the channel, remember
      // peek blocks until one is available

      this.chan_in.peek(tr);

      // Now the transaction has been received, we can check to
      // see if there is any task that we need to do via any callbacks
      // that could be appended for this instance of transactor
      // Pre-Tx callback
   `vmm_callback(pkt_driver_callbacks, master_pre_tx(this, tr, drop));
         if (drop == 1) begin
             `vmm_note(log, tr.psdisplay("Dropped"));
           continue;
         end
      //now process the transaction object,
    $display(tr.psdisplay(" -- TRANSACTOR got the transaction data"));

     // now we can unblock the channel
      this.chan_in.get(tr);
       // Now the the transaction has been processed, we can check to
       // see if there is any task that we need to do via any callbacks
       // that could be appended for this instance of transactor
        `vmm_callback(pkt_driver_callbacks, master_post_tx(this, tr));
      #10;
     end
    endtask: main
```

The callback class name in the `` `vmm_callback `` code above refers to the base name of the transactor callbacks and not any of the extended callback classes. The second argument is the method(s) that will be executed per each call invocation.

At this point the definition process is complete, the instantiation and registration is done at the verification environment build process:

```
virtual function void verif_env::build();
   super.build();
   chan =        new("pkt_transfer","first instance",1000);
   gen =         new("pkt_gen",0,chan);
   driver =      new("pkt_driver",0,chan);

   //// instantiate the callbacks and register the callback to
   //// the driver transactor
   begin
     pkt_driver_cov_callbacks      cov_callb = new();
     driver.append_callback(cov_callb);
   end
   gen.stop_after_n_insts = cfg.trans_cnt;
endfunction: build
```

Either `prepend_callback` or `append_callback` methods of `vmm_xactor` can be used to register the callback façade instance with the instance of transactor. Callback methods will be invoked in the order in which they were registered. The following shows the html coverage report for our pen-plotter example.
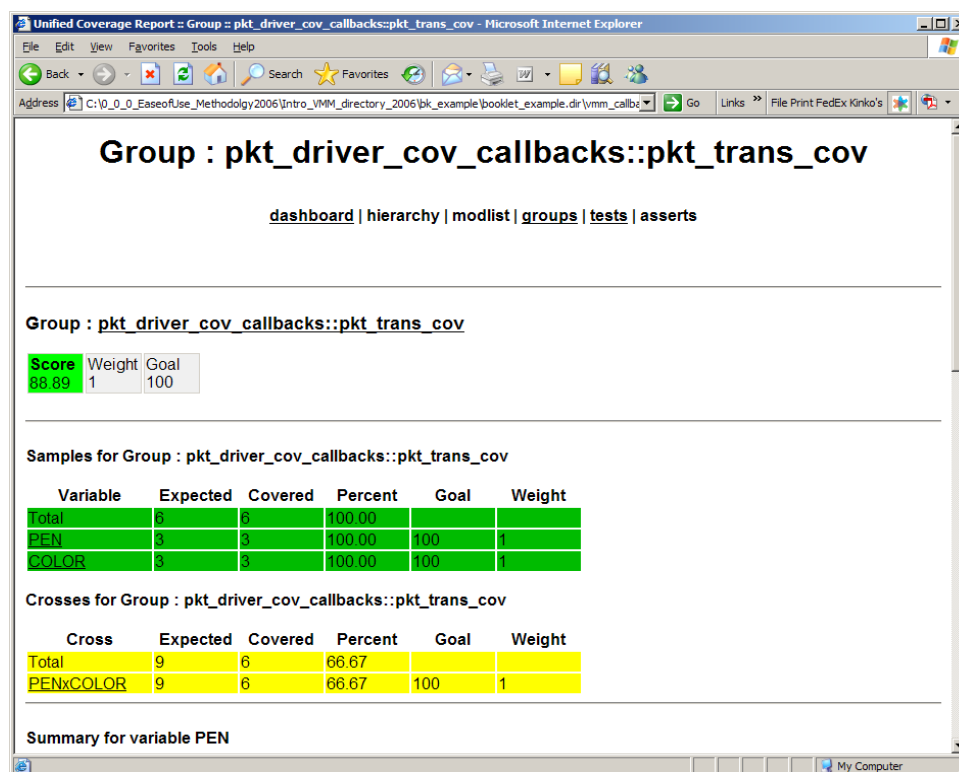
Figure 6: Coverage result for pen plotter

Here is the complete example:

```
program test();
  `include "vmm.sv"
  typedef enum { RED , BLUE, GREEN, ORANGE, PINK, YELLOW } color_t;
  typedef  class   pkt_trans;
  typedef  class   pkt_driver;

///////////////////////////////////////////////
// This is for pkt_driver transactor  callback class
///////////////////////////////////////////////
virtual class pkt_driver_callbacks extends vmm_xactor_callbacks;

   // Callbacks before a transaction is started
   virtual task master_pre_tx(pkt_driver      xactor,
                              ref pkt_trans  trans,
                              ref bit        drop);

   endtask

   // Callback after a transaction is completed
   virtual task master_post_tx(pkt_driver xactor,
                               pkt_trans  trans);
   endtask

endclass: pkt_driver_callbacks

///////////////////////////////////////////////
// This is for coverage of pkt_driver transactor  callback class
// extended from  pkt_driver_callbacks
///////////////////////////////////////////////
class pkt_driver_cov_callbacks extends pkt_driver_callbacks;
  local pkt_trans tr ;

  covergroup pkt_trans_cov;
     PEN: coverpoint tr.pen {
           bins LOW = { [0:2]};
           bins MED = { [3:5]};
           bins HIGH = {[6:7]};
         }
     COLOR: coverpoint tr.color {
           bins LOW = { [RED:BLUE]};
           bins MED = { [GREEN:ORANGE]};
           bins HIGH = { [PINK:YELLOW]};
         }
     PENxCOLOR: cross PEN,COLOR ;
   endgroup

  // Callbacks before a transaction is started
  virtual task master_pre_tx(pkt_driver      xactor,
                             ref pkt_trans trans,
                             ref bit         drop);
   // Empty
  endtask : master_pre_tx
  // Callback after a transaction is completed
  virtual task master_post_tx(pkt_driver xactor,
                              pkt_trans  trans);
```

```
     tr = trans ;                            // Save a handle to the transaction
     pkt_trans_cov.sample();           // Sample Coverage
   endtask : master_post_tx
   function new();
     pkt_trans_cov = new();
   endfunction: new

endclass: pkt_driver_cov_callbacks

//////////////////////////////////////////////////
// pkt_trans  A transaction class for data
//////////////////////////////////////////////////
class pkt_trans extends vmm_data;
// same as defined in previous segments
endclass: pkt_trans

//////////////////////////////////////////////////
// macro for creating the channels for pkt_trans transactions
//////////////////////////////////////////////////
`vmm_channel(pkt_trans)
`vmm_atomic_gen(pkt_trans,"pkt transaction generator")

//////////////////////////////////////////////////
//  pkt_driver   A transactor class for pkt_trans data transaction
//////////////////////////////////////////////////
class pkt_driver extends vmm_xactor;
   pkt_trans_channel      chan_in;

   function new(string               inst,
                int                  stream_id = -1,
                pkt_trans_channel  chan_in);
     super.new("PKT_driver xactor", inst, stream_id);
     // check for channel
           if (chan_in == null)
             this.chan_in = new("pkt_driver_channel", "channel");
           else this.chan_in = chan_in;
   endfunction: new

   virtual function void start_xactor();
     super.start_xactor();
     `vmm_note(this.log, "Starting the transactor for pkt_driver");
     // any specific code related to pkt_driver
   endfunction: start_xactor

   virtual function void stop_xactor();
     super.stop_xactor();
     // any specific code related to pkt_driver
       `vmm_note(this.log, "Stopping the transactor for pkt_driver");
   endfunction: stop_xactor

   virtual function void reset_xactor(reset_e rst_typ = SOFT_RST);
     super.reset_xactor(rst_typ);
     // specific reset type can be placed here.
     // channels have to be flushed as well, for example
     this.chan_in.flush();
   endfunction: reset_xactor
```

```
    // Remember that upper layer, the enviornment calling
    // start_xactor, each vmm_xactor::start_xactor gets called
    // which then calls the main task.
    //////////////////////////////////////////////////////
    //  The  main()  method  definition
    //////////////////////////////////////////////////////
  virtual protected task main();
     bit drop;
     super.main();
     forever begin : main_task_loop
       pkt_trans tr;
       // per rule 4-121, we can get the handle to the next
       // data transaction descriptor in the channel, remember
       // peek blocks until one is available

       this.chan_in.peek(tr);

       // Now the transaction has been received, we can check to
       // see if there is any task that we need to do via any callbacks
       // that could be appended for this instance of transactor
       // Pre-Tx callback
     `vmm_callback(pkt_driver_callbacks, master_pre_tx(this, tr, drop));
          if (drop == 1) begin
                `vmm_note(log, tr.psdisplay("Dropped"));
            continue;
          end

     //now process the transaction object,
     $display(tr.psdisplay(" -- TRANSACTOR got the transaction data"));
     // now we can unblock the channel
      this.chan_in.get(tr);

       // Now the the transaction has been processed, we can check to
       // see if there is any task that we need to do via any callbacks
       // that could be appended for this instance of transactor
     `vmm_callback(pkt_driver_callbacks, master_post_tx(this, tr));

      #10;
     end
    endtask: main
 endclass : pkt_driver

//////////////////////////////////////////////////////
// test configuration class with basic constraints
//////////////////////////////////////////////////////
class test_cfg;
 // How many transactions to generate before test ends?
  rand int trans_cnt;
  constraint basic {
    trans_cnt > 9;
    trans_cnt < 10000000;
    trans_cnt == 10;
  }
endclass: test_cfg

//////////////////////////////////////////////////////
// verification environment class definition
```

```systemverilog
/////////////////////////////////////////////////
class verif_env extends vmm_env;

 pkt_trans_channel       chan;
 pkt_driver              driver;
 pkt_trans_atomic_gen    gen;
 test_cfg                cfg;
 pkt_trans               tr,tr2;

   function new();
      super.new();
      this.cfg = new();
   endfunction: new

   virtual function void gen_cfg();
      super.gen_cfg();
      if (cfg.randomize() ==0)
        `vmm_fatal(log, "Failed to randomize testbench configuration");
        `vmm_note(this.log, "cfg, with trans_cnt ");
   endfunction: gen_cfg

  virtual task run();
      super.run();
  endtask: run

 /////////////////////////////////////////////////
 //   the build method
 //    testbench, transactors, and callbacks
 /////////////////////////////////////////////////
  virtual function void build();
      super.build();
      chan =      new("pkt_transfer","first instance",1000);
      gen =       new("pkt_gen",0,chan);
      driver =    new("pkt_driver",0,chan);

      //// instantiate the callbacks and integrate
      begin
        pkt_driver_cov_callbacks        cov_callb = new();
        driver.append_callback(cov_callb);
      end

      gen.stop_after_n_insts = cfg.trans_cnt;
   endfunction: build

   virtual task start();
      super.start();
      driver.start_xactor();
      gen.start_xactor();
   endtask: start

   virtual task stop();
      super.stop();
      gen.stop_xactor();
      driver.stop_xactor();
   endtask: stop

   virtual task wait_for_end();
```

```
      super.wait_for_end();
      gen.notify.wait_for(pkt_trans_atomic_gen::DONE);
      #150;
   endtask: wait_for_end

  virtual task report();
      super.report();
      `vmm_note(this.log, " ----- TEST REPORT ----");
   endtask: report

   virtual task reset_dut();
      super.reset_dut();
   endtask: reset_dut
 endclass: verif_env

  verif_env       env;
  pkt_trans       tr,tr2;

initial begin
//////////////////////////////////////////////////
// instantiate the channel object and generator object
//////////////////////////////////////////////////
  env = new();
  env.build();
  env.run();
end

endprogram: test
```

Summary: VMM basics

We have briefly looked at the basic constructs and rules of Verification Methogolody Manual (VMM). We will use a simple FIFO design and build the verification testbench structure and test cases for it to further illustrate these concepts.

.

4.0 Creating a testbench using VMM

In this chapter we will apply the guidelines described in previous chapters to a simple FIFO design. This FIFO design is also the DUT used in the VCS quickstart guides mentioned in the introduction section of this booklet.

The FIFO Design Block

The FIFO block is a synchronous (Single-Clock) FIFO with static flags. It is fully parameterized and has single-cycle push and pop operations. It also has empty, half-full, and full flags, as well as parameterized almost full and almost empty flag thresholds with error flags. A block diagram for the FIFO (DW_fifo_s1_sf) design is as follows:
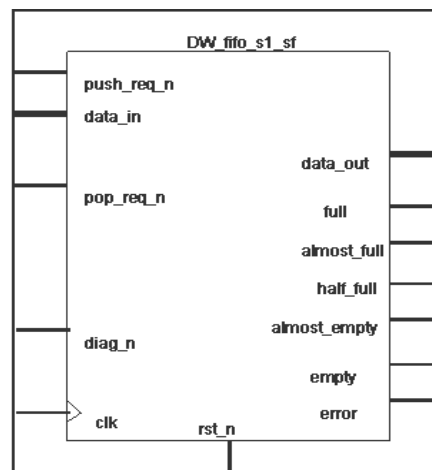
Figure 7: FIFO design block diagram

The following sample timing diagram illustrates how the FIFO block works.
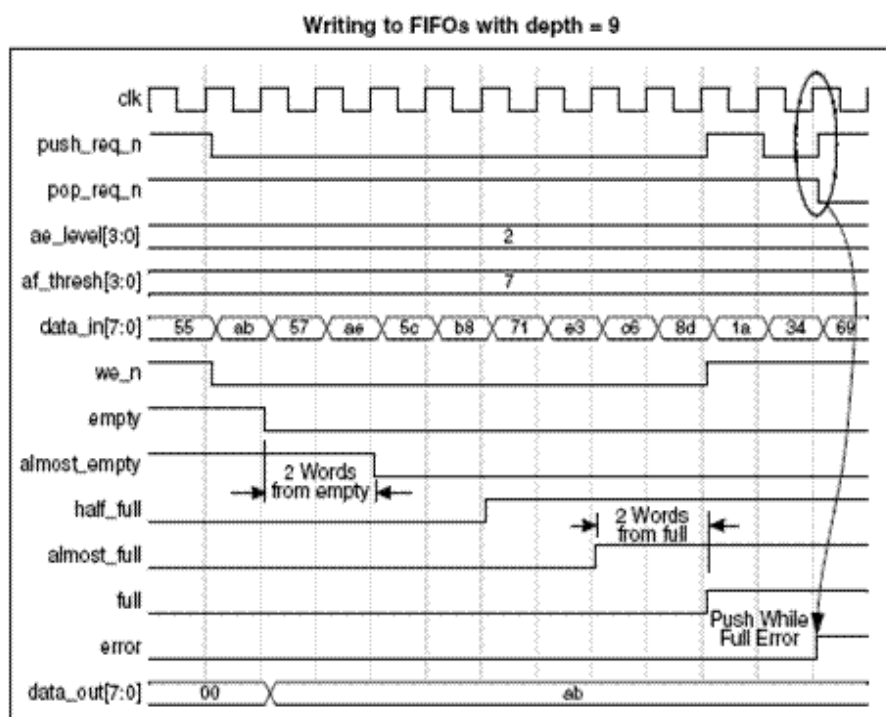
Figure 8: FIFO timing diagram

The verification code will provide you with the steps to develop the testbench and each underlying component and simulate designs. You can use this structure and the code with your current design environment to increase the number of test cases as well as create more complete and complex routines for testing the DUT.

Testbench Files and Structure for FIFO Example

The following is the file structure of the verification components.

```
+ fifo_example/
    +hdl/
    +env/
    +fifo/
    +tests/
```

hdl/   directory contains all Verilog files for Calc1 design

DW_fifo_s1_sf.v   Verilog FIFO model

DW_fifoctl_s1_sf.v   Verilog FIFO controller

DW_ram_r_w_s_dff.v   Verilog memory

top.sv   top level module instantiation of fifo and test

env/   directory contains verification environment, callbacks and scoreboard

dut_env.sv   verification environment class for FIFO (from vmm_env)

dut_sb.sv   scoreboard class for FIFO

cov_callbacks.sv callbacks for coverage collection
sb_callbacks.sv   callbacks for scoreboard integration

fifo/   directory contains transaction class, transactors, configuration and interface

fifo_cfg.sv   test configuration descriptor class

fifo_if.sv   systemverilog interface definition

fifo_master.sv   master driver transactor (from vmm_xactor)
fifo_monitor.sv   monitor transactor (from vmm_xactor)
fifo_trans.sv   base transaction descriptor class (from vmm_data)

tests/   directory contains test case files

test_01.sv   simple test case with all defaults

test_02.sv   test case with factor pattern replacement

Verification Architecture for FIFO

The program block is a SystemVerilog module that is specialized for verification purposes. Verification objects like constraint blocks, functional coverage, random signals, etc, which comprise the test stimulus will be referenced within program blocks.

The following figure is a high level view of the verification architecture. The architecture uses the vmm base classes to build a structured testbench. The top level program will contain an environment object, which will contain generator driver and checker objects.
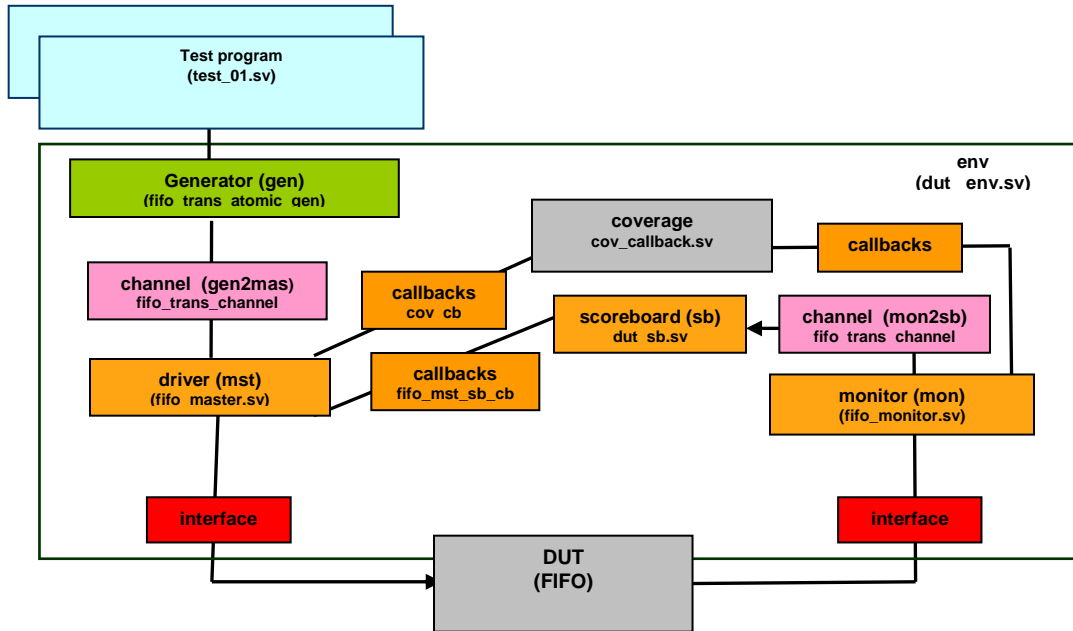
Figure 9: verification architecture overview

If we take a look at the first test suite, we see the desired structure for the environment and testbench using vmm in the main program: test_01.sv

```
`include "fifo_if.sv"    // interface file
program test(intf intf);
  `include "vmm.sv"
   // including all components definition
  `include "fifo_cfg.sv"
  `include "fifo_trans.sv"
  `include "fifo_master.sv"
  `include "fifo_monitor.sv"
  `include "dut_sb.sv"
  `include "sb_callbacks.sv"
  `include "cov_callbacks.sv"
  `include "dut_env.sv"

  dut_env   env;                        // DUT Environment
  initial begin
    env = new(intf);                    // Create the environment
    env.build();                        // Build the environment
    env.run();                          // Run all steps
  end
endprogram : test
```

FIFO Data transaction

The first component to consider is the data transaction descriptor class for FIFO design verification. The FIFO word length,WIDTH and depth, DPETH are parametrizable at the top. We have chosen the other fields for push (write) or pop (read) data rates.

Data	operation/data_rate

Figure 10: FIFO data format

The "data" field is 16-bits wide. The data_rates for write and read are 16-bit wide as well. The code for the base constraints contains the default constraint set. Here is the code for `fifo_trans` class.

```
//------------------------------------------------------------------
// Filename  : fifo_trans.sv
//  Data transaction class extended from vmm_data for fifo example.
//           Introduction to design verification with VMM booklet.
//           (c) Synopsys, Inc. 2006, 2007
//------------------------------------------------------------------
class fifo_trans extends vmm_data;

  static vmm_log log = new ("fifo_trans", "class") ;

  // Local Data Members
   rand logic [`WIDTH-1:0] data [`DEPTH];
   rand logic [15:0]  wr_data_rate;
   rand logic [15:0]  rd_data_rate;
   constraint reasonable {
     wr_data_rate > 0; wr_data_rate < 10;
     rd_data_rate > 0; rd_data_rate < 10;
   }
  // Constructor
  extern function new();

  // VMM Standard Methods
  extern virtual function string   psdisplay(string prefix = "");

  extern virtual function vmm_data allocate ();
  extern virtual function vmm_data copy     (vmm_data to = null);
  extern virtual function void     copy_data(vmm_data to = null);
  extern virtual function bit       compare  (vmm_data to,
                                         output string diff,
                                         input int kind = -1);

  extern virtual function bit       is_valid (bit silent = 1,
                                         int kind = -1);

  extern virtual function int unsigned byte_size  (int kind = -1);
  extern virtual function int unsigned byte_pack  (ref logic [7:0] bytes[],
                                         input int unsigned offset = 0,
                                          input int   kind = -1);
  extern 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);
endclass: fifo_trans
//------------------------------------------------------------
// VMM Macros - Channel and Atomic Generator
//------------------------------------------------------------
`vmm_channel(fifo_trans)
`vmm_atomic_gen(fifo_trans, "FIFO Atomic Gen")
```

Note that the method prototypes are declared above with `extern` keyword, the methods will need definition to complete the class definition. For example the following is the code for allocate function:

```
function vmm_data fifo_trans::allocate();
  // Allocate a new object of this type, and return a handle to it
  fifo_trans i = new();
  allocate = i;
endfunction: allocate
```

The macros for channel and atomic_gen class definition are shown. As mentioned these macros automatically create the two class descriptors, namely `fifo_trans_channel` and `fifo_trans_atomic_gen`.
The full code example is placed in the **appendix D**.

FIFO transactors and base callbacks:

The next component to consider is the master and monitor transactor class for FIFO design. The transactors are extended from `vmm_xactor`. We will also define base callbacks for these transactors, they are extended from `vmm_xactor_callbacks`. The following is the description for `fifo_master` transactor class:

```
class fifo_master extends vmm_xactor;
    // FIFO Interface (Master side)
    virtual  intf.Master  fifo_master_if;
    // FIFO Transaction channels
    fifo_trans_channel      in_chan ;

    extern function new (string instance,
                         integer stream_id = -1,
                         virtual intf.Master fifo_master_if,
                         fifo_trans_channel in_chan = null);

    extern virtual task main() ;
    extern virtual task reset() ;
    // fifo specific methods
    extern protected virtual task fifo_mwrite(ref fifo_trans tr) ;
    extern protected virtual task fifo_mread(fifo_trans tr) ;
    extern protected virtual task do_idle();

endclass: fifo_master
```

The FIFO monitor transactor, `fifo_monitor` class is:
```
//--------------------------------------------------------------
// FIFO Monitor Transactor Class
//--------------------------------------------------------------
class fifo_monitor extends vmm_xactor;
  // Factory Object for creating fifo_trans
  fifo_trans    randomized_obj;
  // FIFO Interface (Monitor side)
  virtual intf.Monitor fifo_monitor_if;
  // Output Channel
```

```
   fifo_trans_channel        out_chan;

   extern function new(string instance,
                       int stream_id = -1,
                       virtual intf.Monitor fifo_monitor_if,
                       fifo_trans_channel   out_chan = null);

   extern virtual task main() ;
   // fifo specific method
   extern virtual task sample_fifo(ref fifo_trans tr);

endclass: fifo_monitor
```

In these transactors we also declare a fifo interface type which connects the master and monitor to the FIFO design at the top level. For a brief review of interface construct see appendix D. Note that the master transactor will use the master modport and the monitor will use the monitor modport.

Refer to the appendix C for full definition of methods and transactor classes.

Pure virtual callback base for fifo_master transactor:
We define a pure base callback class for master transactor which is derived from the `vmm_xactor_callbacks`. This base class is a virtual class with virtual methods. The actual callback class that will get registered to the master transactor is an extension to this callbacks class and will be discussed when we introduce the scoreboard and coverage callbacks classes.

```
//----------------------------------------------------------------
// FIFO Master Callback base Class
//----------------------------------------------------------------
virtual class fifo_master_callbacks extends vmm_xactor_callbacks;

   // Callbacks before a transaction is started
   virtual task master_pre_tx(fifo_master    xactor,
                              ref fifo_trans trans,
                              ref bit        drop);
   endtask
   // Callback after a transaction is completed
   virtual task master_post_tx(fifo_master xactor,
                               fifo_trans  trans);
   endtask
endclass: fifo_master_callbacks
```

Now that the base callbacks class for `fifo_master` is defined we can incorporate the invocation calls in the `fifo_master` main task. Here is the code for `main()` :

```
   task fifo_master::main();
     fifo_trans    tr;
     bit           drop;
     // Fork off the super.main() to perform any base-class tasks
     fork
       super.main();
     join_none

     // Main loop to drive the FIFO Bus
```

```
      while (1) begin
        // Wait if the xactor is stopped on the in_chan is empty
        // Get a transaction from the input channel
        this.wait_if_stopped_or_empty(this.in_chan) ;
        in_chan.get(tr);

        // Pre-Tx callback
        // invoke the callbacks that are derived from fifo_master_callbacks and
        // are registered to this instance of the fifo_master transactor.
        // execute the master_pre_tx methods in callbacks
        `vmm_callback(fifo_master_callbacks, master_pre_tx(this, tr, drop));
        if (drop == 1) begin
          `vmm_note(log, tr.psdisplay("Dropped"));
          continue;
        end

        // Process the transaction
          fifo_mwrite(tr);
          fifo_mread(tr);
        // invoke the callbacks that are derived from fifo_master_callbacks and
        // are registered to this instance of the fifo_master transactor.
        // execute the master_post_tx methods in callbacks
        `vmm_callback(fifo_master_callbacks, master_post_tx(this, tr));
      end
  endtask: main
```

The `vmm_callback` macro invokes the callbacks to ensure that the callbacks are called in proper registration sequence. It also removes the burden of knowing the details of callbacks from the transactor implementation.
The coverage and scoreboard callbacks are derived from fifo_master_callbacks, and implement the **master_post_tx** and **master_pre_tx** methods.
Again, the actual callbacks will be extended from this pure virtual fifo_master_callbacks class. Here we take a look at these extensions.

Scoreboard fifo master callbacks: sb_callbacks.sv
In order to allow integration of scoreboard callbacks to master transactor a callbacks class is derived from fifo_master_callbacks base class. This class will instantiate a scoreboard object and uses its methods to inquire the master transactor. The code below shows the fifo_master_sb_callbacks class:

```
//----------------------------------------------------------
// Scoreback Connection via FIFO Master Callback Class
//----------------------------------------------------------
typedef class    fifo_master;
typedef class    dut_sb;

class fifo_master_sb_callbacks extends fifo_master_callbacks;
  dut_sb      sb;     // scoreboard object
  // Constructor
  function new(dut_sb sb);
    this.sb = sb;
  endfunction: new

  // Callbacks before a transaction is started
  virtual task master_pre_tx(fifo_master    xactor,
```

```
                              ref fifo_trans trans,
                              ref bit         drop);
       // Empty
    endtask: master_pre_tx

    // Callback after a transaction is completed
    virtual task master_post_tx(fifo_master xactor,
                                fifo_trans  trans);
      sb.from_master(trans);
    endtask: master_post_tx

  endclass: fifo_master_sb_callbacks
```

The scoreboard class contains `from_master` method which places the incoming
transaction on the incoming queue to be compared with the output.

FIFO Test Configuration descriptor: `fifo_cfg`
The fifo_cfg class defines a test configuration class for the fifo example. Per rule 4-33 in
VMM, the vmm_env::gen_cfg() method is used to randomize the testcase configuration
descriptor. This will be discussed in fifo verification environment section.

```
    class fifo_cfg;
      // How many transactions to generate before test ends?
      rand int trans_cnt;
      constraint basic {
        trans_cnt > 9;
        trans_cnt < 10000000;
        trans_cnt == 10;
      }
    endclass: fifo_cfg
```

FIFO verification environment: `dut_env.sv`
The testbench verification environment for the FIFO example is derived from vmm_env
class and contains all other components, master and monitor transactors, scoreboard and
coverage, channels. Here is the code for the `dut_env` class:

```
  //---------------------------------------------------------
  // dut_env class
  //---------------------------------------------------------
  class dut_env extends vmm_env ;

    // FIFO Master/Monitor Virtual Interface
    virtual  intf   ifc;

    vmm_log      log;
    fifo_cfg     cfg;
    // channel for the output generator
    fifo_trans_channel     gen2mas;

    // channel for the output monitor mon2scb
    fifo_trans_channel     mon2scb;
    //   Generator, atomic, class is created with the `vmm_atomic_gen
    //   fifo_trans_atomic_gen
```

```
    fifo_trans_atomic_gen        gen;

    fifo_master                  mst;
    fifo_monitor                 mon;
    dut_sb                       scb;

    // Constructor
    extern function new(virtual intf ifc);

    // VMM Environment Steps
    extern virtual function void gen_cfg();
    extern virtual function void build();
    extern virtual task reset_dut();
    extern virtual task cfg_dut();
    extern virtual task start();
    extern virtual task wait_for_end();
    extern virtual task stop();
    extern virtual task cleanup();
    extern virtual task report();

  endclass: dut_env
```

The constructor method is shown below:

```
  function dut_env::new(virtual intf ifc);
    // Pass in the name of the environment to the VMM-Env logger class
    super.new("DUT_ENV");
    // Save a copy of the virtual interfaces
    this.ifc = ifc;
    // Allocate/new() the log using new("dut", "env")
    log = new("dut", "env");
    // Allocate/new() the cfg object
    this.cfg = new() ;
  endfunction: new
```

The configuration testcase is randomized by gen_cfg() method as follows:

```
  function void dut_env::gen_cfg() ;
    super.gen_cfg() ;
    //  Randomize the cfg object
    if (cfg.randomize() == 0)
      `vmm_fatal(log, "Failed to randomize testbench configuration");

    `vmm_note(log, "cfg.trans_cnt");
  endfunction: gen_cfg
```

Next we will take a look at build method which instantiates all the component objects as well as callbacks. It will also take care of registering the callbacks to appropriate transactors.

```
  function void dut_env::build() ;
    super.build() ;
   // instantiate the channel for connecting the atomic generator to the master
    gen2mas = new ("FIFO Trans Channel", "gen2mas");

    // instantiate the channel for connecting the monitor to scoreboard
```

```
    mon2scb = new ("FIFO Trans Channel", "mon2scb") ;
    // instantiate the generator, this is the atomic_gen
    gen = new ("FIFO Atomic Gen", 1, gen2mas) ;
    // instantiate the master object
    mst = new ("FIFO trans master", 1, ifc, gen2mas );
    // instantiate the monitor object
    mon = new ("FIFO trans monitor", 1, ifc, mon2scb);
    // instantiate the scoreboard transactor instance
    scb = new(cfg.trans_cnt, mon2scb) ;

    //  Integrating the scoreboard using callbacks
    //  Create a new fifo_master_sb_callbacks object fifo_mst_sb_cb
    //  Append this using mst.append_callback(fifo_mst_sb_cb)
    begin
       fifo_master_sb_callbacks   fifo_mst_sb_cb  = new(scb);
       mst.append_callback(fifo_mst_sb_cb);
    end

    // Integrating the functional coverage using a callback object
    begin
       fifo_master_cov_callbacks    cov_cb = new();
       mst.append_callback(cov_cb);
    end

    // Configure the generator to stop after cfg.trans_cnt instances
    gen.stop_after_n_insts = cfg.trans_cnt ;

 endfunction: build
```

The start and stop methods in dut_env will call the appropriate start and stop methods of each of instantiated transactors. Note, the base class start method should be called first.

```
    task dut_env::start();
      super.start();
      gen.start_xactor();
      mst.start_xactor();
      mon.start_xactor();
      scb.start_xactor();
    endtask: start

    task dut_env::stop();
      super.stop() ;
      gen.stop_xactor();
      mst.stop_xactor();
      mon.stop_xactor();
      scb.stop_xactor();
    endtask: stop
```

Last method we will look at in this segment for verification environment is wait_for_end. We will use the notification objects (vmm_notify class properties of vmm_env and vmm_xactor) from the generator and scoreboard to indicate that the test is done, i.e, there is no more transactions to be sent by the generator and that the scoreboard check has been completed.

```
task dut_env::wait_for_end();
  super.wait_for_end();

  fork
    gen.notify.wait_for(fifo_trans_atomic_gen::DONE);
    scb.notify.wait_for(scb.DONE);
  join
  #100000;  // let everything settle in the dut
endtask: wait_for_end
```

Refer to the discussion on **vmm_notify** class property which implements an interface to the notification service.

FIFO Scoreboard class: `dut_sb.sv`

The scoreboard class is defined as an extension to the vmm_xactor. It contains a fifo_trans channel for connection to the monitor transactor (actual data) and a queue to hold data from the master transactor (expected data). The notify object within scoreboard will use DONE to indicate its status to the verification environment.

```
class dut_sb extends vmm_xactor;
  vmm_log log;                    // For log messages

  int    max_trans_cnt;       // Max # of transactions
  local  int match;           // Number of good matches

  local fifo_trans_channel mon2scb;     // Transactions from monitor
  local fifo_trans from_master_q[$] ;   // queue of data from the master

  integer DONE;                           // DONE notification

  extern function new(int max_trans_cnt, fifo_trans_channel mon2scb);
  extern task main();
  extern task report();
  extern task cleanup();

  // method specific to fifo scoreboard class, used in the
  // callbacks extensions
  extern function void from_master(fifo_trans tr);

endclass : dut_sb
```

The constructor instantiate the log and assigns the channel to the channel being passed to the scoreboard at initialization. Also the notify object configures the DONE flag to be level-sensitive, that is notifications will remain active (notified) until explicitly reset.

```
function dut_sb::new(int max_trans_cnt,
                     fifo_trans_channel mon2scb);

  super.new("DUT_SB", "class", 0);

  this.log = new("Scoreboard", "Scoreboard");
  this.max_trans_cnt = max_trans_cnt;
```

```
   this.mon2scb        = mon2scb;
   match = 0;
   // Configure DONE notification to be ON/OFF
   this.DONE = notify.configure(-1, vmm_notify::ON_OFF);

 endfunction: new
```

The `from_master` method pushes the transaction on the queue.

```
 function void dut_sb::from_master(fifo_trans tr) ;
   from_master_q.push_back(tr) ;
 endfunction: from_master
```

The main method in the scoreboard starts a forever thread that performs the comparison between the expected output and actual output received the DUT.

```
 task dut_sb::main();
    int i;
    logic check;
    fifo_trans      mas_tr, mon_tr;
    fork
       super.main();
   join_none
 `vmm_note(this.log, "Starting scoreboard",
            $time,  max_trans_cnt)) ;
   while(1) begin
     // Since this device operates as a transfer function, the self-checking
     // mechanism is quite simple. The scoreboard first waits for a
     // transaction to be generated then waits for the monitor to notify that
     // this transaction occurred.  In order to determine the transaction
     // correctness the following rules are applied:
     //   - Each generated WRITE transactions are stored to a register file
     //     (which acts as a reference model in this case).
     //   - Each generated READ transactions get their data field filled from
     //      the register file (so to provide an expected result).
     //   - each transactions is then compared on a first-come first-serve basis.
     mon2scb.get(mon_tr);
     mas_tr = from_master_q.pop_front();

     // Perform the comparison of master vs mon vs memory
     check = 1;
     for (i=0;i<`DEPTH;i++)
       check = check & ( mas_tr.data[i] == mon_tr.data[i] ) ;
     if (check==0) begin
       `vmm_note(log, "CHECK FAILED ==>");
     end
     else
     match++;
     // Determine if the end of test has been reached
     if(match >= max_trans_cnt) begin
       `vmm_note(this.log, "Done scorboarding");
        this.notify.indicate(this.DONE);
     end
   end // while(1)
 endtask: main
```

FIFO Coverage callbacks class: `cov_callbacks.sv`
The functional coverage model is implemented using SystemVerilog coverage group construct: `covergroup`. Coverage groups can be modeled in a self-checking structure such as scoreboard object. They should in general be encapsulated in an coverage object. In the case of FIFO we have chosen to define the covergroups directly in the callbacks class.

The stimulus coverage would be sampled after it has been through the design, i.e, it is known to be valid for coverage information gathering. This usually means that the stimulus coverage is sampled through a passive transactor, such as a monitor. In this FIFO example since our `fifo_master` transactor has implemented a read method it allows us to sample coverage after the data read. Therefore the callbacks for the coverage gathering mechanism are derived from the `fifo_master_xactor_callbacks` in this example.

```
typedef class fifo_trans;
class fifo_master_cov_callbacks extends fifo_master_callbacks;
  local   fifo_trans      tr ;

  covergroup fifo_trans_cov;
     RD: coverpoint tr.rd_data_rate {
           bins LOW = { [1:127]};
           bins MED = { [128:511]};
           bins HIGH = { [512:1023]};
         }
     WD: coverpoint tr.wr_data_rate {
           bins LOW = { [1:127]};
           bins MED = { [128:511]};
           bins HIGH = { [512:1023]};
         }
     RDxWD: cross RD,WD ;
   endgroup

  // Callback method before a transaction is started
  virtual task master_pre_tx(fifo_master    xactor,
                             ref fifo_trans trans,
                             ref bit        drop);
   // Empty
  Endtask : master_pre_tx

  // Callback method after a transaction is completed
  virtual task master_post_tx(fifo_master xactor,
                             fifo_trans  trans);

    tr = trans ;                      // Save a handle to the transaction
    fifo_trans_cov.sample();       // Sample Coverage
  endtask : master_post_tx

  function new();
    fifo_trans_cov = new();
  endfunction: new
 endclass: fifo_master_cov_callbacks
```

FIFO testcase and factory pattern use

Once the verification environment is created for the FIFO and the individual components have been written it becomes fairly easy to create a testcase. We showed this at the beginning of this chapter with the default testcase, `test_01.sv` which uses all default constraints for the base transactions and transactors.

The atomic generator always randomizes the same instance, `randomized_obj` and copies it before sending it to the channel for consumption by other transactors. It is possible to replace this with an instance of a derived class, a factory pattern. The additional or overridden constraint blocks in the derived class will be used since the `randomize()` method is a virtual method. Of course the constraints in the derived data transaction patterns should not conflict with the basic constraints defined in the base transaction descriptor class.

The following code shows the derived `fifo_trans` with modified constraints being defined in the program block. This instance is randomized and is copied to the `randomized_obj` of the generator within the environment object.

```
 `include "fifo_if.sv"
program test(intf intf);
 `include "vmm.sv"

 `include "fifo_cfg.sv"
 `include "fifo_trans.sv"
 `include "fifo_master.sv"
 `include "fifo_monitor.sv"
 `include "dut_sb.sv"
 `include "sb_callbacks.sv"
 `include "cov_callbacks.sv"
 `include "dut_env.sv"

   class my_fifo_trans extends fifo_trans;
     constraint test_02 {
      wr_data_rate inside { 4,5,6 };
      rd_data_rate inside { 4,5,6 };
     }
   endclass: my_fifo_trans

   dut_env   env;                        // DUT Environment
   initial begin
     env = new(intf);               // Create the environment
     env.build();                   // Build the environment

   //create a randomized object and set it to the object
   // in generator
     begin
       my_fifo_trans trans = new();
       env.gen.randomized_obj = trans;
     end
     env.run();                     // Run all steps
   end
 endprogram : test
```

And this is the verilog top module file which instantiates both the program block, interface and the design:

```
module top;

    bit clk;
    intf        intf1(clk);
    always #5 clk = !clk;
    test test1(intf1);
    DW_fifo_s1_sf_wrap fifo_inst(intf1);

endmodule
```

5.0 Summary

We have shown through example the basic concepts embodied in the Verification Methodology Manual. This introduction allows novice users to jump start their verification development using the VMM base class libraries. We have discussed the main ideas of modularized and layered architecture for testbench development and have shown how to step by step create the components that are necessary to build a robust and reusable verification infrastructure.

In the future we will describe more advanced concepts such as advanced scenario generation and assertions with other examples as well as new and additional base classes in VMM. Readers are encouraged to visit the VMM central page to get more information as well as Synopsys verification product pages.

Appendix A: Basics of SystemVerilog Object Oriented Programming

VCS provides data abstraction with class data types. Classes as self-contained components form the foundation of Object-Oriented programming structure. Object-oriented programming is different from the procedural programming with which people are familiar. There are three principles behind objects; Encapsulation, Inheritance and Polymorphism.

Objects, Declaration and Instantiation

An object is an instance of a defined and declared class.

```
Sensor TempSense;  //declare a variable of class Sensor
TempSense = new;  // initialize variable, object of Sensor
```

Instantiation is done by calling the constructor *new* to the object. Objects are dynamic data storage; hence memory is allocated at the time of instantiation for that object and not at the time of definition and declaration.

Encapuslation and Data hiding:

The class data type encapsulates all attributes of coherent data in a system—the members that store data items and the member functions and tasks that manipulate the data and communicate with the rest of the system. Classes efficiently manage the complexity of large programs through modularization. Once declared classes can be instantiated and dynamically created as objects. Hence an object is a container of variables and methods operating on them. We operate on the object by calling its methods.

Classes can contain public part and private parts.   Members—both variable and methods—declared in the private part of class are hidden from the rest of the system; using *protected* or *local* attributes. The public segment is the section that connects the objects to the rest of the environment. This lets designers and verification engineers solve the details of their verification environment piece by piece, in a modular format. As each section is completed, it can be set aside with all its details of implementation from the rest of the system, so that only the public section can affect – or be affected by other segments. The following shows a simple data class for a sensor.

```
class Sensor;
  string model;
  integer address;
  bit state;  // 0 off, 1 on
  integer value;
     // constructor, initialization sub-routine
  function new();
    model = "Normal";
    value = 0;
    state = 0;
  endfunction
  //  methods, accessing the variable members
  task start_sensor();
     state = 1;
```

```
  endtask
  function bit current_state();
     current_state = state;
  endtask
endclass: Sensor
```

In this example we have created a constructor function, new, and one task that starts the sensor and a function which reports the state of the sensor. You can add other methods that operate on the sensor in this class.

Inheritance and Polymorphism:

In order to be able to build re-usable components, the object-oriented technology provides a mechanism for creating hierarchies of classes through *derived* classes. The process of deriving classes is called *inheritance*. Inheritance is the mechanism that allows a class B to inherit properties of a class A. We say ``B extends A''. Objects of class B thus have access to attributes and methods of class B without the need to redefine them.

If class B inherits from class A, then A is called a superclass of B. B is called a subclass of A. Objects of a subclass can be used where objects of the corresponding superclass are expected. This is due to the fact that objects of the subclass share the same behavior as objects of the superclass. Polymorphism, formed of Greek words, meaning "having multiple forms" is the characteristic of being able to assign different meaning or usage in different context to an entity. In a subclass, explicit calls to the methods of a superclass are done by using *super*. For example, *super.new()* in the subclass constructor calls the superclass constructor and has to be the first call in the subclass constructor definition. In the following example Adult class (subclass) extends from Person class (superclass).

```
  class Person;
    //data or class properties
    string name;
    integer age;
    string gender;
    //initialization
    function new();
      Name = "";
      Age = 0;
    endfunction;
    virtual task speak();
      $display("This is a person \n");
    endtask
  endclass

  class Adult extends Person;
    function new();
      super.new();
    endfunction;

    virtual task speak();
      $display(`` my name is %s \n'',name);
    endtask
  endclass: Adult
```

Note the usage of virtual attribute for the *task speak()*. Virtual methods are a basic polymorphic construct. Virtual methods provide prototypes for subroutines and override a method in all the base classes, where as a normal method only overrides a method in that class and its descendants. Subclasses override the virtual methods by following the prototype exactly; all versions of the virtual method look identical in all subclasses as far as number of arguments, return type, and encapsulation are concerned.

In our example if we were to define a generic Person class type, knowing that the class will be derived to define the characteristics of the functions such as speak(), the function has to be declared as virtual and then its behavior defined in derived classes.

```
class Person;
  //data or class properties

  virtual task speak();
  endtask
endclass: Person
```

Now we can define an adult person and a baby with extension of the Person, note in this case we are overriding the function in the base class.

```
class Adult extends Person;
  // properties
  virtual task speak();
    $display('' my name is %s \n'',name);
  endtask
endclass: Adult
```

We can also extend class Person to provide function for another entity, for example, Child:

```
class Child extends Person;
  virtual task speak();
    $display('' bah-bah-bah %s \n'',name);
  endtask
endclass: Child
```

In conjunction with its usage as a means of representing both the common features and specialized aspects of different data structures and components, class derivation is also a tool for modularization. For example, a verification engineer can define a derived class based on an existing class library that was produced specifically for verification tasks.

Appendix B: Interface construct and signal connectivity

SystemVerilog provides a new construct, **interface** as a container which acts as bundle of wires, encapsulating signal definitions and synchronization, and allows ease of connectivity between testbench and design components. Interface can be instantiated like a module. The interface construct provides flexibility through instantiation in a module or a program. The best approach to declare signals for connectivity is to declare interface signals as *wire* in SystemVerilog.

```
interface fifo_intf(input clk);
   parameter WIDTH = 16;
   wire rst_n;
   wire push_req_n;
   ...
   wire [WIDTH-1: 0] data_out;
endinterface : fifo_intf
```

Once an interface container has been defined, variable of that interface can be declared and instantiated.

```
fifo_intf intfFifo(clk);
 // declaration and instantiation of interface of type fifo_intf
```

modports

Directional information for module ports is provided by using the modport construct in SystemVerilog. In a verification environment there are various views and uses for interface signals: some are driven, such as in driver transactors, some are simply monitored, such as in monitor transactors. In order to compartmentalize these different views, modports are declared and defined for each of the transactor views.

```
interface fifo_intf(input clk);
    parameter WIDTH = 16;
    wire rst_n;
    wire push_req_n;
    ...
    wire [WIDTH-1: 0] data_out;
 modport Fifo_Driver(output rst_n;
               output push_req_n;
                ...
               input data_out);

 modport Fifo_Designer(input rst_n;
                  input push_req_n;
                   ...
                  output data_out);
 endinterface: fifo_intf
```

Now you can declare and instantiate the interface and pass the modport to the testbench:
```
fifo_intf intfFifo(clk);
 // declaration and instantiation of interface of type fifo_intf
 // pass modport to a program instantiation.
```

```
    fifo_test test(intfFifo.Fifo_Driver);
```
Note that in order to avoid duplication of effort and create a modular and flexible
verification environment that allows extensibility, the actual task and function code for
transactors is not defined inside the interface.

In the next sections we will describe the usage of clocking blocks to allow synchronous
sampling and driving of signals. This blocking removes any potential race conditions
between the design and verification testbench. Using such constructs in SystemVerilog
will eliminate unpredictability in timing behavior of testbench and design connection.

Virtual interface

Virtual interfaces provide a mechanism for separating abstract models and test programs
from the actual signals that make up the design further promoting code re-use. For
example, a network switch router has many identical ports that operate the same way, say
a 10Gbit MII for Ethernet connection. A single virtual interface declared inside a
class that deals with these Ethernet signals can represent different interface instances at
various times throughout the simulation. A virtual interface can be declared inside as a
class property that can be initialized procedurally or by an argument to the class
constructor.

In the following code example the class `fifo_driver` instantiates a virtual interface:
```
    class fifo_driver
      …
      virtual fifo_intf v_intf;  // full interface instance
      …
    endclass: fifo_driver
```

The instantiation of the interface can have several flavors in the Testbench area:
In a program block:
```
    v_intf = interface_passed_to_program;
```
Or through the class constructor:
```
    function new(virtual fifo_intf vIntf, ...);
            this.v_intf = vInft;
```

And with modport connections:
```
    virtual fifo_intf.Fifo_Driver fifo_driver_if;
```

Similarly the assignment through program block and class constructor is as follows:
    a) Assignment to virtual interfaces:
```
        fifo_wr_if = interface_passed_to_program;
```

    b) Through class constructor:
```
    function new(virtual fifo_intf.Fifo_Driver vIntfDR, ...);
            this.v_intf = vInftDR;
```

Referencing signals within an interface can be accessed using a relative hierarchical path
name. For example:
```
        ...
      initial
          intfFifo.rst_n <= 0;
```

```
        always @(posedge intfFifo.clk)
         $display("Cycle is %0 \n",cyc++);
```

Eliminating race-conditions in synchronous designs

Clocking blocks

One of the major contributions of SystemVerilog has been the introduction of a clocking block as a synchronization construct for signals between testbench and design. Usage of a clocking block inside interfaces will help remove race conditions between the two sides and handles signal delay differences between RTL and gate-level models. Clocking blocks used an in interface definition in conjunction with modports define proper directions for each modport and signals in it. The clocking block arranges signals that are synchronous to a particular clock and makes their timing explicit. The clocking block is key in allowing engineers to focus on test development rather than signals and transitions in time. Depending on the nature of the design and verification environment there can be more than one clocking block.

An example of interface and clocking blocks follows:

```
 interface fifo_intf(input clk,input clk2);
    ...
    parameter setup_t = 2ns;
    parameter hold_t = 3ns;
     default input #setup_t output #hold_t;
     clocking cb @(posedge clk);
        output rst_n;
        output push_req_n;
          ..
        input data_out;
     endclocking

     clocking cbDR @(posedge clk2);
        output rst_n;
        output push_req_n;
           ....
     endclocking
   endinterface: fifo_intf
```

Signals within each clocking block are sampled and driven with respect to the specified clock edge given appropriate setup and hold time. An interface can contain one or more clocking blocks within it.
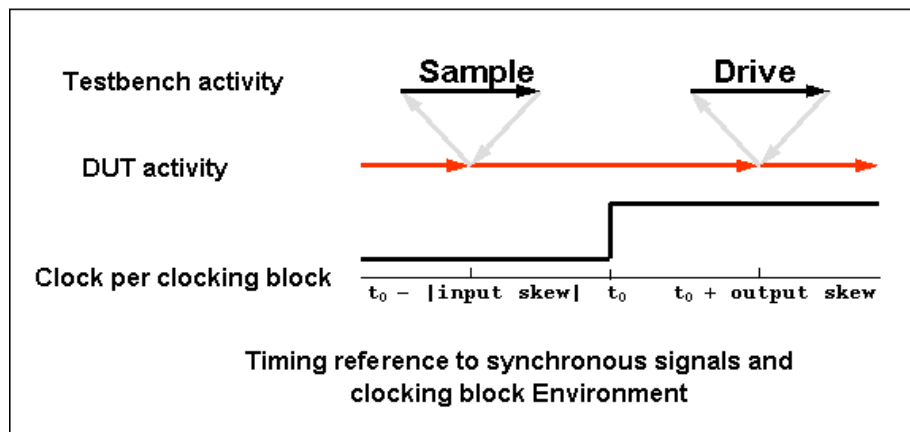
Figure 11: Timing reference for sample and drive of synchronous signals

Modport and Clocking blocks

In the interface definition each modport will have a reference to a clocking block. As discussed previously, the main transactors within the Testbench program block connect to the signals through virtual interface modports to allow extensibility.

```
interface fifo_intf;

    modport fifo_Driver(clocking cbDR);
    modport fifo_Checker(clocking cbCheck)

    ....
endinterface: fifo_intf
```

Signals are then referenced with respect to interface modport name, and are synchronized with the appropriate clocking block timing.

```
v_intf.cb.push_req_n <= 1'b1;    //direct clocking block

fifo_wr_if.cbDR.push_req_n <= 1'b1;  //modport usage
```

Asynchronous signals

The interface modport construct can be used to define and create asynchronous ports and signal connections. The signal is defined as an input or output in a modport declaration:

```
interface fifo_intf;

    modport fifo_Driver(clocking cbDR, output rst_n);
    modport fifo_Checker(clocking cbCheck)
    ....
endinterface: fifo_intf
```

In this case the signal rst_n is defined as asynchronous with respect to the clock edges.In this section we present a few advanced concepts.

Appendix C: Advanced concepts in the VMM based testbench

VMM has a rich set of advanced concepts and guidelines to help further in the generation of complex test scenarios. Similar to atomic generator there exist a scenario generator environment creation mechanism. Here is an overview of the scenario generation class, in the future versions of this handbook we will explore this feature in detail.

VMM_SCENARIO_GEN

The `vmm_scenario_gen(class_name, "Class Description")` is used to create scenario class for each `vmm_data` extended transaction. Similar to `vmm_atomic_gen`, this macro creates a scenario generator class which is an extension of `vmm_xactor` class. For the pen-plotter example in chapter 3, based on pkt_trans transaction class the scenario generator class will be: pkt_trans_scenario_gen.

The macro also defines the following classes named:

`pkt_trans_scenario`, `pkt_trans_scenario_election` and
`pkt_trans_scenario_gen_callbacks`.

Here is the prototype of the scenario_gen class.

```
class pkt_trans_scenario_gen extends vmm_xactor;
   integer stop_after_n_insts;
   integer stop_after_n_scenarios;

   integer GENERATED;
   integer DONE;
   pkt_trans_channel    out_chan;        //gets the copy of trans
   // scenario related members
   // queue set of scenarios which will get randomized
   pkt_trans_scenario    scenario_set[$]; //used for factory

   pkt_trans_scenario_election    select_scenario;

   extern funcion new(string       instance,
                   integer       stream_id = -1,
                   pkt_trans_channel        out_chan  = null);
   extern virtual task inject_obj(pkt_trans obj);
   extern virtual task inject(pkt_trans_scenario scenario);
 endclass : pkt_trans_atomic_gen
```

Appendix D: Example code

The following is the complete code for the FIFO verification example.

```
//-------------------------------------------------------------
//  SYNOPSYS CONFIDENTIAL - This is an unpublished, proprietary work of
// Synopsys, Inc., and is fully protected under copyright and trade secret
// laws. You may not view, use, disclose, copy, or distribute this file or
// any information contained herein except pursuant to a valid written
// license from Synopsys.
//           Introduction to design verification with VMM handbook.
//         (c) Synopsys, Inc. 2006, 2007
//-------------------------------------------------------------
// Filename  : fifo_trans.sv
//  This is a data transaction class extended from vmm_data for
//  fifo example
//
//-------------------------------------------------------------
class fifo_trans extends vmm_data;
  static vmm_log log = new ("fifo_trans", "class") ;

  // Local Data Members
   rand logic [`WIDTH-1:0] data [`DEPTH];
   rand logic [15:0]  wr_data_rate;
   rand logic [15:0]  rd_data_rate;
   constraint reasonable {
     wr_data_rate > 0; wr_data_rate < 10;
     rd_data_rate > 0; rd_data_rate < 10;
   }
  // Constructor
  extern function new();

  // VMM Standard Methods
  extern virtual function string   psdisplay(string prefix = "");
  extern virtual function vmm_data allocate ();
  extern virtual function vmm_data copy     (vmm_data to = null);
  extern virtual function void     copy_data(vmm_data to = null);
  extern virtual function bit       compare  (vmm_data to,
                                      output string diff,
                                      input int kind = -1);

extern virtual function bit      is_valid (bit silent = 1,
                                      int kind = -1);
extern virtual function int unsigned byte_size  (int kind = -1);
extern virtual function int unsigned byte_pack  (ref logic [7:0] bytes[],
                                      input int unsigned offset = 0,
                                        input int  kind = -1);
extern 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);

endclass: fifo_trans


//-------------------------------------------------------------
// VMM Macros - Channel and Atomic Generator
//-------------------------------------------------------------
`vmm_channel(fifo_trans)
`vmm_atomic_gen(fifo_trans, "FIFO Atomic Gen")
```

```
// methods definition for fifo_trans
function fifo_trans::new();
  super.new(this.log);
endfunction: new

function vmm_data fifo_trans::allocate();
  // Allocate a new object of this type, and return a handle to it
  fifo_trans i = new();
  allocate = i;
endfunction: allocate

function vmm_data fifo_trans::copy(vmm_data to);
  fifo_trans cpy;

  // Allocate a new object if needed, check the type if 'to' specified
  if (to == null)
    cpy = new();
  else if (!$cast(cpy, to)) begin
    `vmm_error(this.log, "Cannot copy to non-fifo_trans instance");
    copy = null;
    return;
  end

  // Copy the data fields into the 'to' object and return cpy
  copy_data(cpy);
  copy = cpy;
endfunction: copy

function void fifo_trans::copy_data(vmm_data to);
  fifo_trans cpy;

  // Copy all the VMM base class data
  super.copy_data(to);

  if (!$cast(cpy, to)) begin
    `vmm_error(this.log, "Cannot copy to non-fifo_trans instance");
    return;
  end

  cpy.data = this.data;
  cpy.wr_data_rate = this.wr_data_rate;
  cpy.rd_data_rate  = this.rd_data_rate;
endfunction: copy_data


function bit fifo_trans::compare(vmm_data   to,
                                 output string diff,
                                 input int  kind);
  fifo_trans pkt;
  compare = 1;

  // Check the type is correct
  if (to == null || !$cast(pkt, to)) begin
    `vmm_error(this.log, "Cannot compare to non-fifo_trans instance");
    compare = 0;
    return;
  end

  if (pkt.wr_data_rate !== this.rd_data_rate) begin
    $sformat(diff, "dir (%s !== %s)",  this.wr_data_rate, pkt.wr_data_rate);
    compare = 0;
  end
```

```
    if (pkt.rd_data_rate !== this.rd_data_rate) begin
      $sformat(diff, "addr (%1b !== %1b)", this.rd_data_rate, pkt.rd_data_rate);
      compare = 0;
    end
endfunction: compare


function string fifo_trans::psdisplay(string prefix);

      $sformat(psdisplay, "%s#%0d.%0d.%0d : wr_dr=%02d rd_dr=%02d data_0=%0d",
               prefix, this.stream_id, this.scenario_id, this.data_id,
               this.wr_data_rate, this.rd_data_rate,this.data[0]);

endfunction: psdisplay


function bit fifo_trans::is_valid(bit silent,
                                 int kind);
    is_valid = 1;
endfunction: is_valid


function int unsigned fifo_trans::byte_size(int kind);
    byte_size = 4+128*2;   // 260 bytes
endfunction: byte_size


function int unsigned fifo_trans::byte_pack(ref logic [7:0]    bytes[],
                                            input int unsigned offset,
                                            input int          kind);
  int i,j;
  logic [15:0] tmp;
  bytes = new[offset +  byte_size()];
  bytes[offset]   = wr_data_rate[7:0];
  bytes[offset+1] = wr_data_rate[15:8];
  bytes[offset+2] = rd_data_rate[7:0];
  bytes[offset+3] = rd_data_rate[15:8];
  for (i=0;i<128;i++) begin
    tmp = data[i];
    bytes[offset+4+i*2] = tmp[7:0];
    bytes[offset+5+i*2] = tmp[15:8];
  end
  byte_pack = byte_size();      // Return the number of bytes packed

endfunction: byte_pack


function int unsigned fifo_trans::byte_unpack(
           const ref logic [7:0] bytes[],
           input int unsigned    offset,
           input int             len,
           input int             kind);
   int i;
   wr_data_rate = { bytes[1], bytes[0] } ;
   rd_data_rate = { bytes[3], bytes[2] } ;
   for (i=0;i<128;i++) begin
     data[i] = { bytes[5+2*i] , bytes[4+2*i] } ;
   end
  byte_unpack = byte_size();   // Return the number of bytes unpacked
endfunction: byte_unpack


//-------------------------------------------------------------------
// Filename   : fifo_if.sv
//              This is a interface file for fifo example
//-------------------------------------------------------------------

`define WIDTH 16
```

```
`define DEPTH 128
//`define DEPTH 8
interface intf(input clk);
parameter WIDTH = 16;
  wire rst_n;
  wire push_req_n;
  wire pop_req_n;
  wire diag_n;
  wire [WIDTH-1 : 0] data_in;
  wire empty;
  wire almost_empty;
  wire half_full;
  wire almost_full;
  wire full;
  wire error;
  wire [WIDTH-1 : 0] data_out;
clocking Master_cb @(posedge clk);
  output push_req_n;
  output pop_req_n;
  output diag_n;
  output  data_in;
  output rst_n;
  input empty;
  input almost_empty;
  input half_full;
  input almost_full;
  input full;
  input error;
  input  data_out;
endclocking
clocking Monitor_cb @(posedge clk);
  input push_req_n;
  input pop_req_n;
  input data_in;
  input data_out;
endclocking
modport Master (clocking Master_cb);
modport Monitor (clocking Monitor_cb);
modport dut( input  clk,  rst_n,  push_req_n,  pop_req_n,  diag_n,data_in,
 output  empty,  almost_empty,  half_full,  almost_full,  full, error,
data_out);
endinterface


//-------------------------------------------------------------------
// Filename    : fifo_cfg.sv
//               This is a test configuration class for fifo example
//-------------------------------------------------------------------
class fifo_cfg;

  // How many transactions to generate before test ends?
  rand int trans_cnt;
  constraint basic {
    trans_cnt > 9;
    trans_cnt < 10000000;
    trans_cnt == 10;
  }
endclass: fifo_cfg


//-----------------------------------------------------------------
// Filename  : fifo_master.sv
//            This is a master transactor class for fifo example
//
```

```
//----------------------------------------------------------------
//    This BFM receives FIFO transactions from a channel, and drives
//    the pins via the SystemVerilog virtual interface.
//
//              |        |
//              | FIFO   |  VMM channel
//              | trans  |
//              |  VV    |
//              |        |
//      +-------------------+
//      |      FIFO-Master     |
//      +-------------------+
//              |||||||
//              |||||||
//              FIFO Bus
//
//----------------------------------------------------------------

`define FIFO_MASTER_IF     fifo_master_if.Master_cb

//----------------------------------------------------------------
// FIFO Master Xactor Class
//--------------------------------------------------------------------------

class fifo_master extends vmm_xactor;

    // FIFO Interface (Master side)
    virtual intf.Master fifo_master_if;

    // FIFO Transaction channels
    fifo_trans_channel  in_chan ;

    extern function new (string instance,
                         integer stream_id = -1,
                         virtual intf.Master fifo_master_if,
                         fifo_trans_channel in_chan = null);

    extern virtual task main() ;
    extern virtual task reset() ;

    extern protected virtual task fifo_mwrite(ref fifo_trans tr) ;
    extern protected virtual task fifo_mread(fifo_trans tr) ;
    extern protected virtual task do_idle();

endclass: fifo_master

//----------------------------------------------------------------
// FIFO Master Callback Class
//----------------------------------------------------------------

virtual class fifo_master_callbacks extends vmm_xactor_callbacks;

   // Callbacks before a transaction is started
   virtual task master_pre_tx(fifo_master    xactor,
                              ref fifo_trans trans,
                              ref bit        drop);
   endtask

   // Callback after a transaction is completed
   virtual task master_post_tx(fifo_master xactor,
                               fifo_trans  trans);
   endtask
```

```
  endclass: fifo_master_callbacks

function fifo_master::new(string instance,
                         integer stream_id,
                         virtual intf.Master fifo_master_if,
                         fifo_trans_channel in_chan);

   // Call the super task to initialize the xactor
   super.new("FIFO MASTER", instance, stream_id) ;

   // Save a refernce to the interface
   this.fifo_master_if = fifo_master_if;

   // Allocate an input channel if needed, save a reference to the channel
   if (in_chan == null) in_chan = new("FIFO MASTER INPUT CHANNEL", instance);
   this.in_chan       = in_chan;

endfunction: new

task fifo_master::main();
  fifo_trans      tr;
  bit             drop;

   // Fork off the super.main() to perform any base-class tasks
   fork
     super.main();
   join_none

   // Main loop to drive the FIFO Bus
   while (1) begin

     // Wait if the xactor is stopped on the in_chan is empty
     // Get a transaction from the input channel
     this.wait_if_stopped_or_empty(this.in_chan) ;
     in_chan.get(tr);

//     `vmm_note(log, tr.psdisplay("Master:"));
     // Pre-Tx callback
     `vmm_callback(fifo_master_callbacks, master_pre_tx(this, tr, drop));
     if (drop == 1) begin
       `vmm_note(log, tr.psdisplay("Dropped"));
       continue;
     end

     // Process the transaction
       fifo_mwrite(tr);
       fifo_mread(tr);
     `vmm_callback(fifo_master_callbacks, master_post_tx(this, tr));

   end
endtask: main


task fifo_master::reset();
  `FIFO_MASTER_IF.rst_n <= 0;
  `FIFO_MASTER_IF.pop_req_n <= 1;
  `FIFO_MASTER_IF.push_req_n <= 1;
  `FIFO_MASTER_IF.diag_n <= 1;
  do_idle();
  repeat (5) @(`FIFO_MASTER_IF);
  `FIFO_MASTER_IF.rst_n <= 1;
endtask: reset
```

```
task fifo_master::do_idle();
  `FIFO_MASTER_IF.push_req_n   <= 1;
  `FIFO_MASTER_IF.pop_req_n    <= 1;
   @(`FIFO_MASTER_IF);
endtask: do_idle

task fifo_master::fifo_mwrite(ref fifo_trans tr);
  for (int j = 0; j< `DEPTH; j++)
        begin
          repeat (tr.wr_data_rate) @(`FIFO_MASTER_IF);
          `FIFO_MASTER_IF.push_req_n <= 1'b0;
          `FIFO_MASTER_IF.data_in <= tr.data[j];
          @(`FIFO_MASTER_IF);
          `FIFO_MASTER_IF.push_req_n <= 1'b1;
        end
endtask: fifo_mwrite

task  fifo_master::fifo_mread(fifo_trans tr);
  logic [`WIDTH-1:0] temp;
  for (int j = 0; j< `DEPTH; j++)
        begin
          repeat(tr.rd_data_rate) @(`FIFO_MASTER_IF);
          while (`FIFO_MASTER_IF.empty) @(`FIFO_MASTER_IF);
          `FIFO_MASTER_IF.pop_req_n <= 1'b0;
          @(`FIFO_MASTER_IF);
          `FIFO_MASTER_IF.pop_req_n <= 1'b1;
        end
endtask: fifo_mread

//-----------------------------------------------------------------------------
// Filename  : fifo_monitor.sv
//           This is a monitor transactor class for fifo example
//-----------------------------------------------------------------------------

`define FIFO_MONITOR_IF    fifo_monitor_if.Monitor_cb

//-----------------------------------------------------------------------------
// FIFO Monitor Transactor Class
//-----------------------------------------------------------------------------

class fifo_monitor extends vmm_xactor;
  // Factory Object for creating fifo_trans
  fifo_trans randomized_obj;

  // FIFO Interface (Monitor side)
  virtual intf.Monitor fifo_monitor_if;

  // Output Channel
  fifo_trans_channel out_chan;

  extern function new(string instance,
                      int stream_id = -1,
                      virtual intf.Monitor fifo_monitor_if,
                      fifo_trans_channel out_chan = null);

  extern virtual task main() ;

  extern virtual task sample_fifo(ref fifo_trans tr);

endclass: fifo_monitor

//-----------------------------------------------------------------------------
// FIFO Monitor Callback Class
```

```
//----------------------------------------------------------------------------

typedef class fifo_monitor;

virtual class fifo_monitor_callbacks extends vmm_xactor_callbacks;

   // Callbacks before a transaction is started
   virtual task monitor_pre_rx(fifo_monitor     xactor,
                                  ref fifo_trans trans);
   endtask

   // Callback after a transaction is completed
   virtual task monitor_post_rx(fifo_monitor xactor,
                                  fifo_trans  trans);
   endtask

endclass: fifo_monitor_callbacks


function fifo_monitor::new(string instance,
                           int stream_id,
                           virtual intf.Monitor fifo_monitor_if,
                           fifo_trans_channel out_chan);

   super.new("FIFO TRANS monitor", instance, stream_id) ;

   // Allocate an output channel if needed, save a reference to the channel
   if (out_chan == null) out_chan = new("FIFO MASTER OUTPUT CHANNEL", instance);
   this.out_chan        = out_chan;

   // Create the default factory object
   randomized_obj = new();

   // Save the inteface into a local data member
   this.fifo_monitor_if = fifo_monitor_if;

endfunction: new

task fifo_monitor::main();

  fifo_trans tr;

  // Fork super.main to perform any base-class actions
  fork
    super.main();
  join_none

  // Main Monitor Loop
  while(1) begin

    $cast(tr, randomized_obj.copy());

    // Pre-Rx Callback
    `vmm_callback(fifo_monitor_callbacks ,monitor_pre_rx(this, tr));

    // Sample the bus using the fifo_sample() task
    sample_fifo(tr);
    // Put the trans into the output channel using sneak so it can't block
    out_chan.sneak(tr);

    `vmm_callback(fifo_monitor_callbacks ,monitor_post_rx(this, tr));

    // Printthe transaction in debug mode
```

```
//       `vmm_note(log, tr.psdisplay("Monitor ==>"));

   end

endtask: main

task fifo_monitor::sample_fifo(ref fifo_trans tr);
    int i;
    tr.rd_data_rate  =  0;
    tr.wr_data_rate =   0 ;
    for (i=0;i<`DEPTH;i++)  begin
      while(`FIFO_MONITOR_IF.pop_req_n) @(`FIFO_MONITOR_IF);
        tr.data[i] = `FIFO_MONITOR_IF.data_out;
        @(`FIFO_MONITOR_IF);
    end

endtask: sample_fifo

//--------------------------------------------------------------------------------
// Filename  : dut_env.sv
//           This is FIFO testsbench environment extending vmm_env class
//--------------------------------------------------------------------------------
//
// This class instantiates all the permanent testbench top-level components
//
// After all components are instantiated  this will include:
//   * FIFO atomic generator
//   * FIFO master
//   * FIFO monitor
//   * scoreboard
//
//--------------------------------------------------------------------------------

//--------------------------------------------------------------------------------
// dut_env class
//--------------------------------------------------------------------------------

class dut_env extends vmm_env ;
  // FIFO Master/Monitor Virtual Interface
  virtual intf  ifc;

  vmm_log log;
  fifo_cfg cfg;
  // channel for the output generator
  fifo_trans_channel gen2mas;

  // channel for the output monitor mon2scb
  fifo_trans_channel mon2scb;

  // Generator, atomic, class is created with the macro
  // fifo_trans_atomic_gen(.)
  fifo_trans_atomic_gen gen;

  fifo_master            mst;
  fifo_monitor           mon;
  dut_sb                 scb;

  // Constructor
  extern function new(virtual intf ifc);

  // VMM Environment Steps
  extern virtual function void gen_cfg();
  extern virtual function void build();
```

```
    extern virtual task reset_dut();
    extern virtual task cfg_dut();
    extern virtual task start();
    extern virtual task wait_for_end();
    extern virtual task stop();
    extern virtual task cleanup();
    extern virtual task report();

endclass: dut_env

  function dut_env::new(virtual intf ifc);
    // Pass in the name of the environment to the VMM-Env logger class
    super.new("DUT_ENV");

    // Save a copy of the virtual interfaces
    this.ifc = ifc;

    // Allocate/new() the log using new("dut", "env")
    log = new("dut", "env");

    // Allocate/new() the cfg object
    this.cfg = new() ;
endfunction: new


function void dut_env::gen_cfg() ;
  super.gen_cfg() ;
  //  Randomize the cfg object
  if (cfg.randomize() == 0)
    `vmm_fatal(log, "Failed to randomize testbench configuration");

  `vmm_note(log, "cfg.trans_cnt");

endfunction: gen_cfg

function void dut_env::build() ;
  super.build() ;
  // instantiate the channel for connecting the atomic generator to the master
  gen2mas = new ("FIFO Trans Channel", "gen2mas") ;

  // instantiate the channel for connecting the monitor to scoreboard
  mon2scb = new ("FIFO Trans Channel", "mon2scb") ;

  // instantiate the generator, this is the atomic_gen
  gen = new ("FIFO Atomic Gen", 1, gen2mas) ;

  // instantiate the master object
  mst = new ("FIFO trans master", 1, ifc, gen2mas ) ;

  // instantiate the monitor object
  mon = new ("FIFO trans monitor", 1, ifc, mon2scb);

  // instantiate the scoreboard transactor instance
  scb = new(cfg.trans_cnt, mon2scb) ;

  //  Integrating the scoreboard using callbacks
  //  Create a new fifo_master_sb_callbacks object fifo_mst_sb_cb
  //  Append this using mst.append_callback(fifo_mst_sb_cb)
  begin
    fifo_master_sb_callbacks  fifo_mst_sb_cb  = new(scb);
    mst.append_callback(fifo_mst_sb_cb);
  end
```

```
  // Integrating the functional coverage using a callback object
  begin
    fifo_master_cov_callbacks cov_cb = new();
    mst.append_callback(cov_cb);
  end

  // Configure the generator to stop after cfg.trans_cnt instances
  gen.stop_after_n_insts = cfg.trans_cnt ;

endfunction: build

task dut_env::reset_dut();
  super.reset_dut();
  mst.reset();
endtask:reset_dut

task dut_env::cfg_dut();
  super.cfg_dut()  ;
endtask: cfg_dut


task dut_env::start();
  super.start();

  gen.start_xactor();
  mst.start_xactor();
  mon.start_xactor();
  scb.start_xactor();
endtask: start

task dut_env::wait_for_end();
  super.wait_for_end();

  fork
    gen.notify.wait_for(fifo_trans_atomic_gen::DONE);
    scb.notify.wait_for(scb.DONE);
  join
  #100000;
endtask: wait_for_end

task dut_env::stop();
  super.stop() ;
  gen.stop_xactor();
  mst.stop_xactor();
  mon.stop_xactor();
  scb.stop_xactor();
endtask: stop

task dut_env::report() ;
  super.report() ;
  scb.report();
endtask: report


task dut_env::cleanup();
  super.cleanup() ;
  scb.cleanup();
endtask: cleanup

//----------------------------------------------------------------
// Filename  : dut_sb.sv
//            This is a scoreboard transactor class for fifo example
//
```

```
//---------------------------------------------------------------

class dut_sb extends vmm_xactor;
  vmm_log log;              // For log messages

  int max_trans_cnt;       // Max # of transactions
  local int match;         // Number of good matches

  local fifo_trans_channel mon2scb;    // Transactions from monitor
  local fifo_trans from_master_q[$] ;  // queue of data from the master

  integer DONE;                        // DONE notification

  extern function new(int max_trans_cnt, fifo_trans_channel mon2scb);

  extern task main();
  extern task report();
  extern task cleanup();

  extern function void from_master(fifo_trans tr);
endclass

function dut_sb::new(int max_trans_cnt,
                    fifo_trans_channel mon2scb);
  super.new("DUT_SB", "class", 0);
  this.log = new("Scoreboard", "Scoreboard");
  this.max_trans_cnt = max_trans_cnt;
  this.mon2scb       = mon2scb;
  match = 0;
  // Configure DONE notification to be ON/OFF
  DONE = notify.configure(-1, vmm_notify::ON_OFF);

endfunction: new

function void dut_sb::from_master(fifo_trans tr) ;
  from_master_q.push_back(tr) ;
endfunction: from_master


task dut_sb::main();
   int i;
   logic check;
  fifo_trans mas_tr, mon_tr;

  // Fork off the super.main() to perform any base-class tasks
  fork
    super.main();
  join_none

   `vmm_note(this.log, "Starting scoreboard ") ;

  while(1) begin

    // Since this device operates as a transfer function, the self-checking
    // mechanism is quite simple. The scoreboard first waits for a
    // transaction to be generated then waits for the monitor to notify that
    // this transaction occurred.  In order to determine the transaction
    // correctness the following rules are applied:
    //   - Each generated WRITE transactions are stored to a register file
    //     (which acts as a reference model in this case).
    //   - Each generated READ transactions get their data field filled from
    //      the register file (so to provide an expected result).
```

```
    //   - each transactions is then compared on a first-come first-serve
basis.
    mon2scb.get(mon_tr);
    mas_tr = from_master_q.pop_front();


    // Perform the comparison of master vs mon vs memory
    `vmm_debug(this.log, "Scorebaord check");
    check = 1;
    for (i=0;i<`DEPTH;i++)
      check = check & ( mas_tr.data[i] == mon_tr.data[i] ) ;
    if (check==0) begin
        `vmm_note(log, "CHECK FAILED ==>");
    end
    else
    match++;
    // Determine if the end of test has been reached
    if(match >= max_trans_cnt) begin
      `vmm_note(this.log, "Done scorboarding);
      this.notify.indicate(this.DONE);
    end
  end // while(1)
endtask: main

task dut_sb::report();
endtask: report

task dut_sb::cleanup();
endtask: cleanup


//----------------------------------------------------------------
// Filename  : sb_callbacks.sv
//            This is the scoreboard integration callback class
//
//            Introduction to design verification with VMM booklet.
//          (c) Synopsys, Inc. 2006, 2007
//----------------------------------------------------------------

//----------------------------------------------------------------
// Scoreback Connection via FIFO Master Callback Class
//----------------------------------------------------------------

typedef class fifo_master;
typedef class dut_sb;

class fifo_master_sb_callbacks extends fifo_master_callbacks;
  dut_sb sb;
  // Constructor
  function new(dut_sb sb);
    this.sb = sb;
  endfunction: new

  // Callbacks before a transaction is started
  virtual task master_pre_tx(fifo_master    xactor,
                             ref fifo_trans trans,
                             ref bit        drop);
   // Empty
  endtask: master_pre_tx

  // Callback after a transaction is completed
  virtual task master_post_tx(fifo_master xactor,
                              fifo_trans  trans);
```

```
      sb.from_master(trans);
  endtask: master_post_tx
endclass: fifo_master_sb_callbacks

typedef class fifo_monitor;
class fifo_monitor_sb_callbacks extends fifo_master_callbacks;
  dut_sb sb;
   // Constructor
  function new(dut_sb sb);
    this.sb = sb;
  endfunction: new

  // Callbacks before a transaction is started
  virtual task monitor_pre_rx(fifo_monitor    xactor,
                              ref fifo_trans trans);
  endtask: monitor_pre_rx

  // Callback after a transaction is completed
  virtual task monitor_post_rx(fifo_monitor xactor,
                               fifo_trans  trans);
  endtask: monitor_post_rx
endclass: fifo_monitor_sb_callbacks


//-----------------------------------------------------------------------
// Filename  : cov_callbacks.sv
//           This is specific to the fifo DUT, and collects coverage
//-----------------------------------------------------------------------
//-----------------------------------------------------------------------
// coverage  FIFO Master Callback Class
//-----------------------------------------------------------------------
typedef class fifo_trans;

class fifo_master_cov_callbacks extends fifo_master_callbacks;
  local fifo_trans tr ;
  covergroup fifo_trans_cov;
      RD: coverpoint tr.rd_data_rate {
            bins LOW = { [1:127]};
            bins MED = { [128:511]};
            bins HIGH = { [512:1023]};
          }
      WD: coverpoint tr.wr_data_rate {
            bins LOW = { [1:127]};
            bins MED = { [128:511]};
            bins HIGH = { [512:1023]};
          }
      RDxWD: cross RD,WD ;
   endgroup

  // Callbacks before a transaction is started
  virtual task master_pre_tx(fifo_master    xactor,
                             ref fifo_trans trans,
                             ref bit        drop);
   // Empty
  endtask

  // Callback after a transaction is completed
  virtual task master_post_tx(fifo_master xactor,
                              fifo_trans  trans);

    tr = trans ;                   // Save a handle to the transaction
    fifo_trans_cov.sample();        // Sample Coverage
```

```
  endtask

  function new();
    fifo_trans_cov = new();
  endfunction: new
endclass: fifo_master_cov_callbacks


//----------------------------------------------------------------------
// Filename  : test_02.sv
//            This is a test suite.
//----------------------------------------------------------------------
//----------------------------------------------------------------------
// coverage  FIFO Master Callback Class
//----------------------------------------------------------------------
`include "fifo_if.sv"
program test(intf intf);
`include "vmm.sv"

`include "fifo_cfg.sv"
`include "fifo_trans.sv"
`include "fifo_master.sv"
`include "fifo_monitor.sv"
`include "dut_sb.sv"
`include "sb_callbacks.sv"
`include "cov_callbacks.sv"
`include "dut_env.sv"

  class my_fifo_trans extends fifo_trans;
    constraint test_02 {
     wr_data_rate inside { 4,5,6 };
     rd_data_rate inside { 4,5,6 };
    }
  endclass: my_fifo_trans

  dut_env env;                       // DUT Environment

  initial begin
    env = new(intf);                  // Create the environment
    env.build();                      // Build the environment

  //create a randomized object and set it to the object
  begin
    my_fifo_trans trans = new();
    env.gen.randomized_obj = trans;
  end
    env.run();                        // Run all steps
  end

endprogram



AA
```

SYNOPSYS®