VMM12 User Guide

D-2009.12
Aug 2009

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

Copyright Notice and Proprietary Information

Right to Copy Documentation

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

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

Destination Control Statement

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

Disclaimer

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

Registered Trademarks (®)

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

Trademarks (™)

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

Service Marks (SM)

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

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

Contents

4 Configurations & Options

5 OSCI TLM 2.0 Interoperability (vmm_tlm*)

6 Transaction and Environment Debugging

7 Utility Features and Classes

1

Introduction

The VMM methodology defines industry best practices for creating robust, reusable and scalable verification environments using SystemVerilog. Introduced in 2005, it is the industry's most widely used and proven verification methodology for SystemVerilog. The VMM methodology enables verification novices and experts alike to develop powerful transaction-level, constrained-random verification environments. A comprehensive set of guidelines, recommendations and rules help engineers avoid common mistakes while creating interoperable verification components. The VMM Standard Library provides the foundation base classes for building advanced testbenches, while VMM Applications provide higher-level functions for improved productivity.

VMM 1.2 adds new classes and concepts to provide additional functionality and flexibility. We have added parameterization support to many existing classes including channels and generators. We have added TLM-2.0 support, which adds remote procedure call

functionality between components and extends support to SystemC modules. Configuration Options adds a rich set of methods for controlling testbench functionality from the runtime command line. The Class Factory allows for faster stimulus configuration and reuse. A new Common Base Class make possible multiple name spaces, hierarchical naming along with enhanced search functionality RTL configuration insures the same configuration is shared between RTL and testbench.

Also, we have added the concept of phasing and timelines for enhanced flexibility and reuse of verification components. Implicit phasing enables components to control their own status. Hierarchical phasing promotes block-to-system reuse, set-up, and restart of power domains. Serial phasing supports multiple timelines within a simulation, improving regression productivity.

VMM 1.2 is based on the feedback and contributions of many verification engineers who participate in the verification community discussion forums at VMM Central. Please join us at www.vmmcentral.org to engage with fellow verification engineers and industry experts, and to obtain VMM-related support. In addition, VMM Central offers the latest VMM-related news and a broad array of useful resources.

2

Class/Object Relationships

Understanding the relationships between classes (inheritance hierarchy) and objects (instance hierarchy) is a key aspect with verification environments. VMM not only provides a rich hierarchy of classes for all verification needs, but also provides mechanisms to create, control, and visualize the objects in a verification environment.

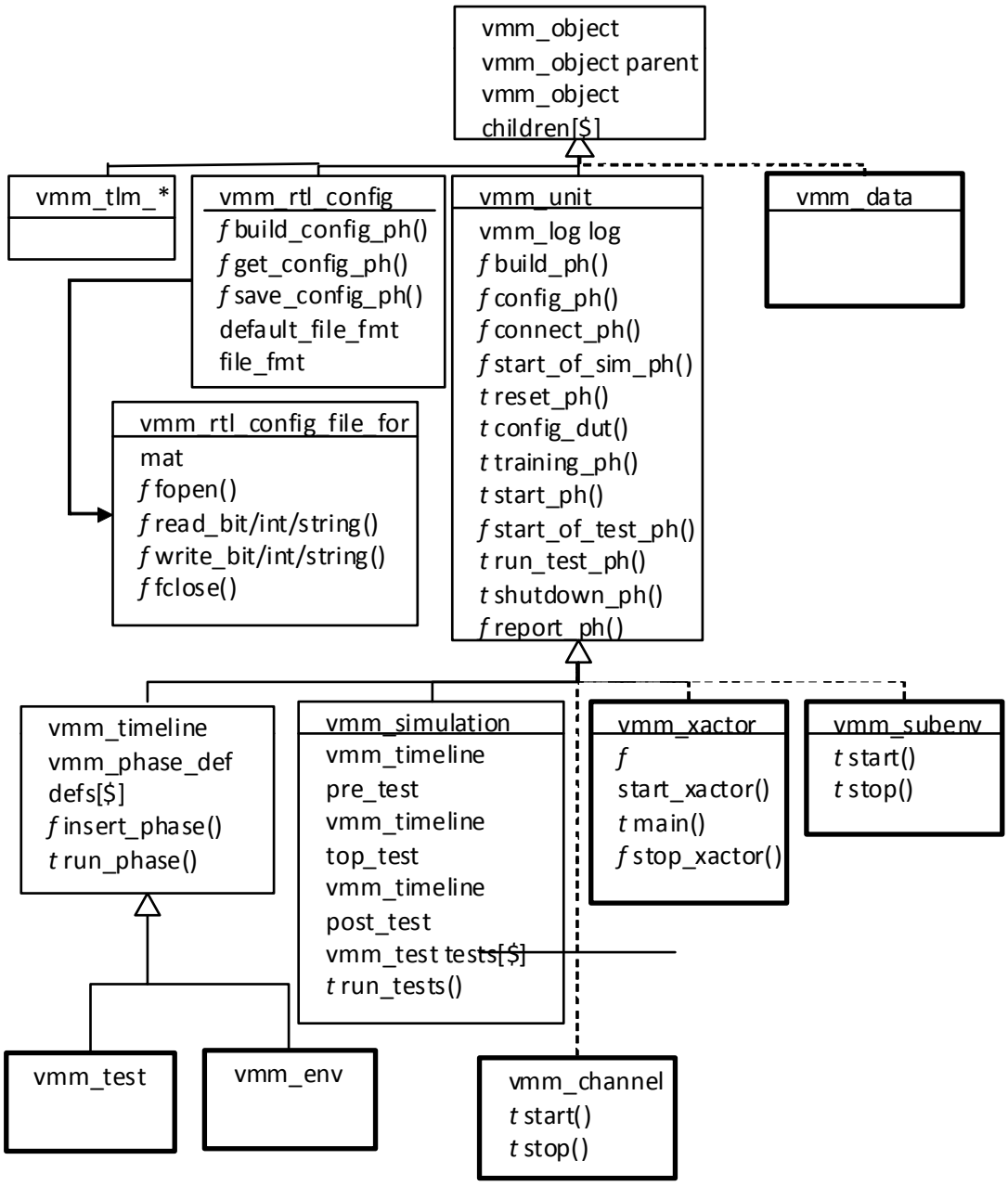
Common Object (vmm_object)

`vmm_object` is a virtual class that is used as the common base class for all VMM-related classes. This helps to provide parent/child relationships for class instances. Additionally, it provides local, relative and absolute hierarchical naming. It also provides multiple

namespaces for creating different object registries. The following diagram shows the relationship between the various VMM classes with `vmm_object` as the base class.

Figure 1 Class Diagram

v



Class/Object Relationships

The following sections describe some of the actions that can be performed using `vmm_object` methods. For a more comprehensive list of methods please reference the `VMM Standard Library Reference Guide (VMM 1.2).`

Setting Object Relationships

The `vmm_object` constructor takes the object parent and the object name as optional arguments. The ‘parent’ object, if passed, is used to set the association with the newly created object. Additionally, the new object is added to parent-child queue. In case the parent is not passed as a constructor argument, this object is created in the root scope.

Thus the parent-child association is created when any class extending from `vmm_object` is created. This association is set during construction by passing the ‘parent’ handle to the `super.new()` call within the constructor in the extended objects.

The following snippet of code shows how the parent-child association is built up during construction.

```
class A extends vmm_object;

function new(vmm_object parent=null, string name);

  super.new(parent,name);

endfunction

endclass


class B extends vmm_object;
```

```
function new(vmm_object parent=null, string name);
  super.new(parent,name);
endfunction
endclass

class C extends A;
function new(vmm_object parent=null, string name);
  super.new(parent,name);
endfunction
endclass

class D extends vmm_object;
   A a1;
   B b1;
   C c1;
   C c2;

   function new(vmm_object parent=null, string name);
       super.new(parent,name);
       a1 = new(this, "a1");
       b1 = new(this, "b1");
       c1 = new(this, "c1");
```

```
        c2 = new(null,"c2"); //No parent
    endfunction


endclass


initial begin
    D d1 = new(null,“d1”);
```

Instances `a1, b1, c1` and the parent `d1` are passed through the constructor and thus parent-child association between them is established. However, `c2` will not have a parent and will exist at the root scope:

[d1]

|--[a1]

|--[b1]

|--[c1]

[c2]

The `vmm_object` association can also be built up through methods provided in the `vmm_object` base class. There are methods to query for the parent object as well. These methods can be used dynamically to set the parent-child relationships, modify them and be used for structural introspection.

```
E e1 = new("e1_obj");
```

```
parent = e1.d1.c1.get_parent_object();

if (parent.get_object_name() != "d1")
    ` vmm_error(log, $psprintf("Expected d1 got %s”,
                                    par-
ent.get_object_name());

e1.d1.c1.set_parent_object(e1);
```

In the above example, `get_parent_object()` invoked from instance `c1` returns parent object of `c1`, which turns out to be `d1`. Then, `c1` parent can be dynamically changed to `e1` as shown above by using the `set_parent_object()` method.

Querying Objects

Given that different components directly or orthogonally extend the `vmm_object` base class, pre-defined methods can be used to query hierarchical names, find objects and children by name; find the root of an object and so on. While invoking these functions, the VMM regular expressions as defined in the Custom Regular Expressions section can be used to define the search patterns.

The methods `find_child_by_name()` and `find_object_by_name()` find the named child or object as a hierarchical name relative to this object in the specified namespace. The name can be a regular expression or a pattern. Similarly, the

`get_object_name(), get_object_hiername()` methods return the local and hierarchical names respectively of the object. The `get_nth_root()` and the `get_nth_child()` methods return the nth root and the nth child respectively of the specified object.

The following examples show how these various methods can be used to find and query objects:

```
class my_class extends vmm_object;
    E e1;
    function new (string name, vmm_object parent=null);
        super.new(parent,name);
        e1 = new("e1_obj",this);
    endfunction
endclass

my_class inst1 = new("inst1");
initial begin
    vmm_object obj;

    obj = inst1.find_child_by_name("e1_obj");
    if(obj.get_object_hiername != "inst1:e1_obj")
      `vmm_error(log, “Expected inst1:e1_obj got”);
```

```
obj = e1.get_nth_child(1);
if (obj != null) begin
    if (obj.get_object_name() != "e1_obj_f1_obj")
        `vmm_error(log “Expected e1_obj_f1_obj:”);

if (inst1.e1.d2.a1.get_object_hiername(inst1.e1) !=
     "d2_obj:a1_obj")
    `vmm_error(log, “Expected d2_obj:a1_obj”);

root = E::get_nth_root(1);
if (root != e1)
    `vmm_error(log, “Expected e1_obj”);
```

Printing and Displaying Objects

You can create an association between the various objects in the testbench using the `vmm_object` base class. This association can vary dynamically as the object is created, destroyed or associations change. At any point in time, it’s possible to view the complete hierarchy of the testbench or the sub-hierarchy of any instance of an object. This is possible using the `print_hierarchy()` function.

```
class E extends vmm_object;
    D d1;
    D d2;
```

```
    F f1;
    F f2;
    function new(string name, vmm_object parent=null );
        super.new(parent,name);
        d1 = new("d1",null);
        d2 = new("d2",this);
        f1 = new("f1",this);
        f2 = new("f2");
    endfunction
endclass

E e1 = new("e1");
initial
  vmm_object::print_hierarchy(e1);
```

The above code snippet produces the following output:

```
[e1]
|--[d2]
|   |--[a1]
|   |--[b1]
|   |--[c1]
|--[f1]
```

Object Traversing

The `vmm_object_iter` class instance traverses the hierarchy rooted at the specified root object, looking for objects whose relative hierarchical name in the specified namespace matches the specified name. Beginning at a specific object, you can traverse through the hierarchy via the `vmm_object_iter::first()` and `vmm_object_iter::next()` methods.

Continuing from the previous example, here is a way to traverse an object hierarchy:

```
   vmm_object my_obj;

   my_iter = new(pattern, e1);


   `vmm_note(log, $psprintf("Match pattern: %s with root
e1", pattern));

   my_obj = my_iter.first();

   if(my_obj.get_object_name() == "d2")

      `vmm_note(log, "Expected object matched - d2");


   my_obj = my_iter.next();

   if(my_obj.get_object_name() == "f1")

      `vmm_note(log, "Expected object matched - f1");
```

`foreach_vmm_object is a powerful macro to iterate over all objects of a specified type and name under a specified root.

The following example shows how to traverse an object's hierarchy using `foreach_vmm_object macro:

```
`foreach_vmm_object(vmm_object, "@%*", e1) begin
    `vmm_note(log, {"Got:", obj.get_object_name()});
end
```

3

Component Phasing and Simulation Timelines

Structural Component Class (vmm_unit)

The structural component class is the base class to create all structural components of a testbench, such as transactors, transaction level models, generators, monitors, scoreboards, etc.

Key features:

- Support structural composition and connectivity
- Integrate into a simulation timeline

A structural instance can be a leaf if parent argument is passed or a non-leaf component if parent argument is not/null passed.

```
class ahb_driver extends vmm_unit;
  function new(string name,
               string inst,
               vmm_object parent = null);
    super.new(name,inst,parent);
endclass
```

Implicit Phasing

The `vmm_unit` class has various virtual methods available for execution during different testbench phases. The user can override them with required functionality for a particular `vmm_unit` extension.

All the methods listed below are typically called as phase methods and execute in corresponding phases of the enclosing timeline of the unit instance. These methods are called in a pre-defined sequence of execution if run with a timeline with all default phases intact.

```
build_ph()
configure_ph()
connect_ph()
start_of_sim_ph()
reset_ph()
training_ph()
config_dut_ph()
start_ph()
start_of_test_ph()
run_ph()
shutdown_ph()
```

```
cleanup_ph()
report_ph()
destruct_ph()
```

The following example shows how to model a transactor that extends `vmm_unit`:

```
          program test;
            class vip extends vmm_unit;
     function new(string name,
                  string inst,
                  vmm_object parent = null);
                 super.new(name,inst,parent);
               endfunction

function void build_ph();
   `vmm_note(log, " in %M...");
endfunction:build_ph

function void connect_ph();
   `vmm_note(log, " in %M...");
endfunction:connect_ph

task config_dut_ph();
   `vmm_note(log, " in %M...");
```

```
endtask:config_dut_ph


task run_ph();
  `vmm_note(log, " in %M...");
endtask:run_ph
            endclass:vip


            vmm_timeline tl;
            vip  vip1;


            initial begin
              tl = new ("top_timeline", "tl");
              vip1 = new (“vip”,”vip1”,tl);
              tl.run_phase();
            end


endprogram
```

For this unit test example, the output is:

```
------------------------------------------------
Normal[NOTE] on class test.vmm_unit(vip1) at 0:
      build_ph...
```

```
Normal[NOTE] on class test.vmm_unit(vip1) at 0:
     connect_ph...
Normal[NOTE] on class test.vmm_unit(vip1) at 0:
      config_dut_ph...
Normal[NOTE] on class test.vmm_unit(vip1) at 0:
     run_ph...
--------------------------------------------------
```

Disabling a Unit Instance

For a test specific objective, or to debug part of the code, we may want to disable one or more unit instances. In this case, the disabled unit phases are no longer considered by the timeline to which it belongs to, implicit timeline is `vmm_test`. A unit instance can be disabled as follows:

```
class top_unit extends vmm_unit;
    ahb_driver drv0,drv1;
    `vmm_typename(top_unit)
    virtual function void build_ph();
       drv0 = new(“ahb_driver”,”drv0”,this);
       drv1 = new(“ahb_driver”,”drv1”,this);
   endfunction
endclass
```

```
  //single driver test
  class my_test1 extends vmm_test;
  `vmm_typename(my_test1)

    virtual function void build_ph();
      top_unit_inst = new(“top_unit”, ”top_unit_inst”,
this);
    endfunction

    virtual function void configure_ph();
        // Disable drv1
        top_unit_inst.drv1.disable_module();
    endfunction
  endclass
```

Adding / Inserting / Deleting / Overriding Phases for a Unit Instance

We can add/delete/insert a phase of an enclosing timeline for a particular unit instance. These methods are handy when you want to do something out of default execution for a unit instance. The following examples shows how to insert, override and delete a given phase:

```
  class udf_delay_def extends vmm_fork_task_phase_def
#(unitExtension);
      `vmm_typename(udf_delay_def)
      virtual task do_task_phase(unitExtension obj);
         if(obj.is_enabled())
            obj.udf_delay_ph();
      endtask:do_task_phase
  endclass:udf_start_def

  class unitExtension extends vmm_unit;
    `vmm_typename(unitExtension)

    virtual task udf_delay_ph();
      #10;
    endtask

    function build_ph();
      vmm_timeline t = this.get_timeline();
      udf_delay_def udfdelayph = new;

      // insert new udf_start phase after reset
      t.insert_phase("udf_start", "reset", udfdelayph);
```

```
    // override reset definition with udfdelayph phase
    t.insert_phase("reset", "reset", udfdelayph);

    // delete udf_start phase
    t.delete_phase("udf_start");

  endfunction
endclass
```

Creating a Unit Level Component Using vmm_unit

You can model structural components using `vmm_unit`:

```
class vip1 extends vmm_unit;
`vmm_typename(vip1)

  function new(string inst, vmm_unit parent = null);
    super.new(get_typename(), inst, parent);
  endfunction

  virtual task config_dut_ph();
    `vmm_note(log, "Configuring dut...");
```

```
      #5;
     `vmm_note(log, "Done Configuring dut...");
  endtask
endclass

class vip2 extends vmm_unit;
`vmm_typename(vip2)

  function new(string inst, vmm_unit parent = null);
    super.new(get_typename(), inst, parent);
  endfunction

  virtual task run_ph();
    `vmm_note(log, "Starting run_phase...");
     #5;
    `vmm_note(log, "Ending run_phase...");
  endtask
endclass
```

Instantiating Structural Components Using vmm_unit

You can reuse structural components to create a higher layer of structural component by simply extending the `vmm_unit` type:

```
class block extends vmm_unit;

`vmm_typename(block)

  vip1 v1;

  vip2 v2;


  function new(string inst, vmm_unit parent = null);

    super.new(get_typename(), inst, parent);

  endfunction


  virtual function void build_ph();

      v1 = new("v1", this);

      v2 = new("v2", this);

  endfunction

endclass
```

Simulation Timelines (vmm_timeline)

Classes and method used to coordinate the simulation execution.

Key features:

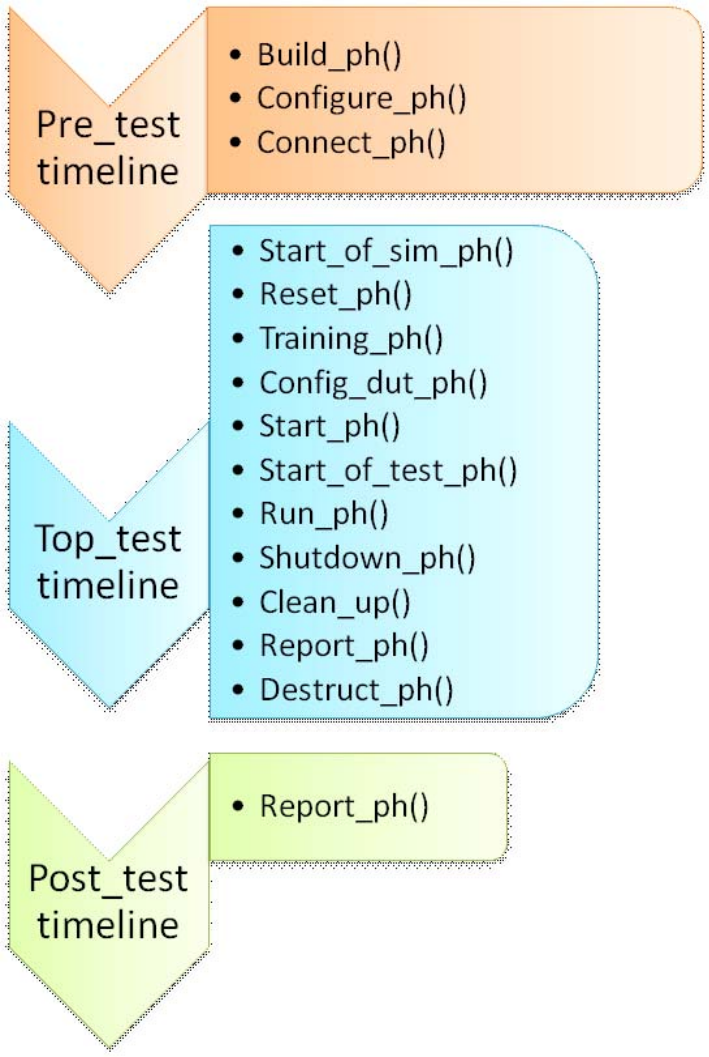
- Support distributed phasing.
- Support different phase timelines, such as detached timelines or phase rollback
- Support test concatenation

Introduction to the Different Phases of Execution

In the implicit phasing functionality, a simulation consists of a series of timelines, composed of pre-defined and user-defined phases. A simulation executes a pre-test timeline, then one or more test timelines, then a post-test timeline. The simulation is kick-started by calling `vmm_simulation::run_tests()` from the top level testbench.

A timeline controls the phasing of all its children that derive `vmm_unit`. The root `vmm_object` instances are implicitly controlled by the pre-test, top-level test and post-test timelines. A `vmm_object` hierarchy may contain timelines at different levels. The pre-test, top and post-test timelines are located in `pre_test(), top_test(), post_test()` respectively predefined in the base class.

The following diagram shows phases call order and location within timelines:



When `vmm_simulation::run_tests()` is called, the following sequence of steps is executed:

- **Step 1: Run the pre-test timeline**: on the all `vmm_unit` hierarchies (excluding top-level `vmm_test` instances that are not selected to be run). The pre-test timeline contains the pre-defined phases “rtl config”, “build”, “configure”, and “connect”.
- **Step 2: Run the top-level test timeline**: including the top-level `vmm_test` instance(s) for the test(s) that is executed. Multiple `vmm_test` instances may be included when multiple tests are executed concurrently. The top-level timeline contains the pre-defined phases “start_of_sim”, “reset”, “training”, “config_dut”, “start”, “start_of_test”, “run_test”, “shutdown”, “cleanup”, “report”, “destruct”.
- **Step3: Execute multiple tests if present:** Reset the top-level test timeline to the “start_of_sim” phase, if there are more tests to be run consecutively, repeat step 2 for the next test.
- **Step 4: Execute the post-test timeline**. The post-test timeline contains the pre-defined phase “report”

The roles of different pre-defined phases in an environment are summarized below:

- Pre-test timeline
  - **RTL configuration**: Generate configurations suitable for RTL compilation. See section “RTL Configuration” in this user guide for examples.
  - **Build**: Instantiate and allocate environment components. VMM channel connections between components may be optionally made here. This is called top-down.
  - **Configure**: Each component (transactor, generator etc.,) provides default configuration values, and a configuration for the environment is generated, either randomly from the top-level, in a directed fashion, or using default values in the components. This is called top-down.
  - **Connect**: VMM-TLM connections between components will be made here. VMM channel connects should be made here if not made at the end of the “Build” phase. This is called top-down.
- Test timeline

- **Start_of_sim**: Each component or environment is rolled back to this phase, and it is called top-down.
- **Reset**: Perform DUT reset which is typically an activity of the top-level environment. However, in some situations such as low-power mode/testcase, there may be multiple resets happening on different interfaces, in which case the lower level components may implement some functionality in this phase. This task is forked off.
- **Training**: Some interfaces / lower level components require a training phase. This task is forked off.
- **Config_dut**: DUT configuration sequences are executed, possibly using Multi-stream scenarios. This task is forked off.
- **Start**: Start any daemons such as generators, transactors etc., This task is forked off.
- **Start_of_test**: Function which is called top-down.
- **Run_test**: test execution which is forked off.
- **Shutdown**: Environment waits until completion based on consent from all the components and the test, before executing a shut-down. It is recommended to use *vmm_consensus* to have a scalable end-of-test implementation. This task is forked off.
- **Cleanup**: Tests may cleanup and restore state back to where they found it. This task is forked off.

- Post-test timeline

- **Report**: A custom report of pass/fail and other auxiliary information regarding test status, prior to completion. This function is called top-down.
- **Destruct**: Any cleanups to be performed by the environment. This function is called bottom-up.

In addition to the built-in phases, user defined phases may be added, inserted between these phases by any component in the environment. There is complete flexibility to control or modify user-defined phases from other components to provide powerful capabilities.

Any phase that executes before the “build” phase executes on the root objects only, because the object hierarchy has not been built yet.

When a vmm_timeline object is encountered, it executes the sub-timeline up to the currently-executing phase in the higher-level timeline. This allows sub-timelines to create phases that do not exist in the top-level phase or that may have the same name—but be different from—other phases in other timelines.

Creating a Transactor with Default Phases

The following is an example of implementing some pre-defined phases in a transactor that extends vmm_unit:

```
`include “vip_trans.sv”
class vip extends vmm_unit;
   rand int param;
   vip_data_chan in_chan;
   function new(string name, string inst,
                vmm_object parent = null);
      super.new(name,inst,parent);
   endfunction
//build any sub-units or helper classes
   function void build_ph;
      `vmm_note(log, " build_ph...");
   endfunction:build_ph
//set defaults for configuration
```

```
  function void configure_ph;
    `vmm_note(log, "configure_ph...");
    param = vmm_opts::get_object_int(
      is_set, this, "param", "SET param value", 0);
    if (is_set)
      param = param.rand_mode(OFF); //or set a default
endfunction:configure_ph
//if using VMM-TLM, connect sub-units
  function void connect_ph;
    `vmm_note(log, "connect_ph...");
  endfunction:connect_ph

  task start_ph;
    `vmm_note(log, "start_ph...");
    fork
    main(); //implement core functionality in main-loop
    join_none
  endtask:start_ph
endclass:vip
```

Creating an Environment with Default Phases

An environment is implemented by extending `vmm_unit`. It usually contains the various testbench components such as transactors, monitors, generators, coverage model, etc.

To recap from the introduction to this section, the top level timeline is comprised of pre-test, test and post-test timeline.

The top-level pre-test timeline is called top-down. The pre-test timeline contains the pre-defined phases “rtl config”, “build”, “configure”, and “connect”.

The top-level test timeline contains the pre-defined phases “start_of_sim”, “reset”, “training”, “config_dut”, “start”, “start_of_test”, “run_test”, “shutdown”, “cleanup”. The “start_of_sim” and “start_of_test” are called top-down. The “reset”, “training”, “config_dut”, “start”, “run_test”, “shutdown” and “cleanup” phases are forked off to run in parallel.

Here is an example of an environment containing one transactor and one generator. It demonstrates how to instantiate them, connect them using a channel, and implement a few relevant phases.

```
`include “vip_trans.sv”

class my_env extends vmm_unit;

   `vmm_typename(my_env)

   vip   bfm1;

   gen1 gen1;

   vip_trans_chan chan1;
```

```
  function new(string inst, vmm_unit parent = null);
    super.new(get_typename(), inst, parent);
  endfunction

  virtual function void build_ph;
    v1 = new("v1", this);
    gen1 = new("gen1", this);
    chan1 = new(“chan1”, this);
  endfunction
//override default configurations
  function void configure_ph;
    `vmm_note(log, "configure_ph...");
  // override default configuration for the environment
    vmm_opts::set_int("v1:param",2);
  endfunction:configure_ph
//connect sub-units
  function void connect_ph;
    `vmm_note(log, "connect_ph...");
    gen1.out_chan = chan1;
    bfm1.in_chan = chan1;
  endfunction:connect_ph
```

```
      //Device specific reset
      task reset_ph;
         `vmm_note(log, " reset...");
      endtask:reset
   //Implement DUT initialization sequences
      task config_dut();
         `vmm_note(log, " reset...");
      // Drive directed sequences, or a
      // specific initialize scenario from MSS generator
      endtask:reset
   //wait-till-end-of-test consensus
      task shutdown_ph();
      //Ideally need vmm_consensus: scalable end
         wait @consensus;
         `vmm_note(log, " All children signal comple-
tion...");
      endtask:start
   endclass: my_env
```

Note that the pre-test timeline phases in `vmm_unit` are implicitly called top-down, and thus the phases for the sub-units of the environment will be automatically called in the correct order.

Also, note that the test timeline phases (except “start_of_sim” and “start_of_test”) are forked off. Each phase **needs to complete** before the next phase can start. **Thus, the lower level units should take care to fork off a main thread which performs the functions of the transactor / generator.**

Multi-test Management (vmm_test)

`vmm_test` base class provides an easy way to develop test cases. You simply have to create a new class that extends `vmm_test` and add test specific implementation to this class. As `vmm_test` extends `vmm_timeline`, all underlying transactors that do extend `vmm_unit` are automatically run and controlled from your test.

Creating a Test

A simple default test is trivially implemented as follows:

```
// New default test

class test1 extends vmm_test;

   `vmm_typename(test1)


   function new(string name);

      super.new(name);

   endfunction


endclass: test1
```

A typical test changes or adds a few constraints to existing transactions, introduces modifications etc. Here is an example of adding a few constraints to the existing transaction, and inserting it back using the object factory. The key learning here is the test-specific timeline implementation. This example shows the test implementing the *start_of_sim_ph()* phase, which automatically gets called at the appropriate time by the top level timeline.

```
//new test with some constraints
`include “vip_trans.sv”
class test2_trans extends vip_trans;
   `vmm_typename(test2_trans)
   constraint { … }
endclass: test2_trans

class test2 extends vmm_test;
   virtual function void start_of_sim_ph();
      `vmm_note(log,
      $psprintf("Starting %s - %M",
                this.get_typename()));
   //Do some test-specific activity
   //i.e add constraints to obj, replace prototypes, etc
      vip_data::override_with_new(“@%*”,
```

```
                        test2_trans::this_type, log);
   endfunction
endclass: test2
```

Creating a Top Level and Executing Single and Multiple Tests with Default Phases

The environment and tests are instantiated in a top level testbench container. A *program* is recommended for race-free testbench, though a module may be used as well. The top level container uses the `vmm_simulation` global singleton class. The simulation is kick-started by calling `vmm_simulation::run_tests()` from the top level testbench.

```
program automatic top;
    test1 t1;
    test2 t2;
    initial begin
        my_env env;
        env = new("env");
        t1 = new("test1");
        t2 = new("test2");
        vmm_simulation::run_tests();
    end
endprogram
```

The simulation executable can now select the different tests at run-time. For example, to start test1:

```
%simv +vmm_test=test1
```

If we want to run both test1 and test2 back to back in the same simulation run, simply do:

```
%simv +vmm_test=test1+test2
```

This will automatically start test1 first, complete it, and the environment takes care of resetting the phase to `start_of_sim_phase()`. Test2 then starts and executes to completion. A snippet of the simulation log would look like:

```
Command: ./simv +vmm_test=test1+test2 -l run.log


Normal[NOTE] on vmm_simulation(class) at 0:
     test name is test1+test2
Normal[NOTE] on vmm_simulation(class) at 0:
     Running Test Case test1


     ...


Normal[NOTE] on vmm_simulation(class) at 0:
     Test Case test1 Done
Normal[NOTE] on vmm_simulation(class) at 0:
     Running Test Case test2
```

```
  Normal[NOTE] on Testcase(test2) at 0:

       Starting class P.test2 - P.\test2::start_of_sim_ph


       ...


  Normal[NOTE] on vmm_simulation(class) at 0:

       Test Case test2 Done

  Simulation PASSED on /./ (/./) at                      0
(6 warnings, 0 demoted errors & 0 demoted warnings)
```

It is important to notice that when serializing multiple tests they may not behave the same way when they are running standalone, unless special care is taken to ensure that they start with a clean slate. VMM phasing provides the capability for tests to restore the initial state at the end of the test using the “cleanup” phase. In situations where the tests may be serialized, this phase should be used by each test to restore the environment to the state found by the test on entry, including, but not limited to restoring the state of the random generators. This is a topic for advanced discussion.

Advanced Usage

Adding User-Defined Phases to a Transactor, Environment or Test

User defined phases can be created and added to a the parent timeline of any component (based on `vmm_unit`) or test (based on `vmm_test`). Let's say that we would like to add a phase *udf_delay_phase* to the transactor *vip*, with the requirement that it be a task (so that it can consume time) and that it be forked off.

```
class vip extends vmm_unit;

   `vmm_typename(vip)

   ...

   //user defined method

   virtual task udf_delay_ph();

      #(test_cfg.unit_delay);

   endtask

endclass
```

First we create a user defined phase wrapper extending from `vmm_fork_task_phase_def.` This base class is chosen since the requirement was that the inserted phase be a task and be forked off. If we wanted it to be a function with top-down execution, we would have implemented the user defined phase as an extension of

`vmm_topdown_function_phase_def` . Note that the implementation of this extension wraps around the actual task *udf_delay_ph* which the user wants to add to the phase sequence.

```
//define new user defined phase

class udf_delay_def extends vmm_fork_task_phase_def
#(vip);

   `vmm_typename(udf_delay_def)


   virtual task do_task_phase(vip obj);

      if(obj.is_enabled())

         obj.udf_delay_ph();

   endtask

endclass
```

We now insert the new user defined phase into the parent timeline. This should be done in the *build_ph* phase as shown below.

```
class vip extends vmm_unit;


   virtual function void build_ph;

      vmm_timeline tl = this.get_timeline();

      udf_delay_def udfdelayph = new;


//schedule unit_delay phase exec before reset for vip
```

```
    tl.insert_phase("unit_delay", "reset", udfdelayph);

  endfunction


endclass: vip
```

Inserting phases in the environment is done in exactly the same way, since an environment is also a `vmm_unit`.

Inserting phases in the test is conceptually similar, by has a few differences in implementation:

The insertion of the user-defined test-specific phase should be in the *start_of_sim_ph* since it is test-specific.

The `vmm_test` class derives from `vmm_timeline`, hence the user defined phase can be inserted by calling *this.insert_phase* directly

The example below shows the insertion of *udf_delay_ph()* in a test, before the reset phase.

```
class test1 extends vmm_test;
  `vmm_typename(test1)
  ...
//user defined method for my_env
  virtual task udf_delay_ph();
    #(env_cfg.test_delay);
  endtask
```

```
   virtual function void start_of_sim_ph;
      udf_delay_def udfdelayph = new;
//schedule test_delay phase execution before reset
      this.insert_phase("test_delay", "reset",
                        udfdelayph);
   endfunction
endclass
```

Controlling / Disabling / Nullifying Existing Phases

Timelines cannot be disabled once created, but unit instances can be disabled for future phase execution as shown under “Disabling unit instance” section.

The `vmm_null_phase_def` class should be used to override a predefined or existing phase to skip its implementation for a specific `vmm_unit` instance. The following example shows how the pre-defined training phase in the *vip1* transactor present in the environment can be skipped / nullified in a testcase.

```
class test1 extends vmm_test;
  `vmm_typename(test1)
  my_env env;

   virtual function void start_of_sim_ph;
```

```
      vmm_null_phase_def nullph = new;


      //skip training phase in vip1 unit hierarchy
      env.vip1.override_phase("training",null_ph);
   endfunction
endclass
```

Execution Threads vs. Phases

Phasing refers to the progress of the overall simulation, not the state or existence of execution threads. An execution thread may span multiple phases, be started or stopped by different phases, but it is not a phase in and of itself. The following simplified example highlights a couple of behaviors of phases, which the user should be cognizant of.

The example shows the task *main()* which in VMM1.1-style transactors (derived from `vmm_xactor`) is the task which is executed when the environment called `vmm_xactor::start_xactor`*().* In this example, the *main()* task is started from the user implementation of the pre-defined *run_ph()*. The first aspect to note is that *main* task is forked off, and continues execution even after *run_ph()* completes. It is the user responsibility to stop this task appropriately in a later phase, most likely in the shutdown phase, or in user-defined phases.

```
class vip extends vmm_unit;
   `vmm_typename(vip)
```

```
    local bit activate_mainloop;
    task main();
      while(activate_mainloop) begin: main_task
          #100;
           //user logic meant to run forever…
      end
    endtask

    task run_ph();
       fork
           begin
              activate_mainloop = 1;
              this.main();
           end
       join_none
    endtask

    //one example of allowing a top level transactor to
control thread
    task void stop_xactor;
       activate_mainloop = 0;
    endtask
```

```
   //shutdown phase
   task shutdown_ph();
       if ( /*some_end_condition_has_been_seen*/ )
           this.stop_xactor;
           //give consent using a vmm_consensus..
   endtask
endclass
```

The other observation is that *run_ph()* should ensure that *it forks off* tasks which run for an arbitrary (or infinite) amount of time. If this task is executed without forking off, the *run_ph()* would not terminate, and would block the execution of succeeding phases for the rest of the component hierarchy.

Event Generation or Callbacks on Phase Execution

Each phase has associated events and status flags which are available to check from source code during execution.

`vmm_phase::is_done()`, `vmm_phase::is_running`(): The method is_done() may be used to check the execution status of any phase. For a particular timeline, calling is_done() on that phase will return '1', whenever a particular phase execution finishes.

```
initial begin
    vmm_phase         ph;
```

```
vmm_timeline top = new("top", "top");

ph = top.get_phase("connect");

wait(ph.is_done() == 1);
```

The method `vmm_phase::is_running()` checks the status for any task phase. This is not meaningful for any function phase as the phase will execute in zero time and the result of the `vmm_phase::is_running()` will always be ‘0’.

```
initial begin

    vmm_phase          ph;

    vmm_timeline top = new("top", "top");

    ph = top.get_phase("reset");

    wait(ph.is_running() == 1);
```

Completed , started: There are events within phase definitions which gets triggered at different times for phase execution , which user can use to monitor the phase executon status

```
initial begin

    vmm_phase          ph;

    vmm_timeline top = new("top", "top");

    ph = top.get_phase("reset");

    fork

         begin

             @(ph.started);

             `vmm_note(log, “ reset phase is running”);

         end
```

```
          begin
            @(ph.completed);
            `vmm_note(log, “ reset phase is completed”);
          end
     join
 end
```

Options to Control Phasing from Command Line and from Source Code

You have several options to control the step-by-step progress for phase execution. The `vmm_timeline::step_function_phase` may be used to execute the specified function phases in this timeline. Note that the phase to be executed must be a function phase and must be the next executable phase.

```
class my_test extends vmm_test;
   ...
endclass

my_test test = new();

initial begin
   //do step execution for rtl_config and build phases
   test.step_function_phase(“rtl_config”);
```

```
   test.step_function_phase(“build”);


   //once build seems correctly done run it till the end
   test.run_phase();    //or vmm_simulation::run_tests();
```

There are options available from the command line to stop at different phases (either globally or for a specific timeline). The command line below shows the options for stopping on reset phase and on training phase. A *$stop* is executed before these phases are set to execute. More details in the user guide.

```
% ./simv +vmm_break_on_reset_phase \
          +vmm_break_on_training_phase
```

The above functionality is enabled by using the `vmm_opts` capabilities, and hence can be executed from source code as well. For example, to stop on top level reset phase, the following code snippet may be added in the top level program. This is the source code equivalent of the command line example above.

```
//top level program
initial begin
   //global for all timelines, can be instance specific
   vmm_opts::set_bit("break_on_reset_phase",null);
end
```

4

Configurations & Options

Hierarchical Options (vmm_opts)

Configurations can now be set from the simulator command line or a file. These configurations can be set on an instance basis or hierarchically by using regular expressions.

Configuration parameters can be set from three different sources, in order of increasing priority: within the code itself using `vmm_opts::set_*()` methods, external option files and command-line options.

Configuration parameter values can be set from two different sources: by explicitly calling the `vmm_opts::get_*()` methods and by randomizing the vmm_object hierarchies.

In addition to existing VMM 1.1 vmm_opts methods, a few new methods let you specify configurations for:

- Global configurations with expressions like the following one to set `Field=Value` for all objects that contain option `Field`

```
% ./simv +vmm_opts+Field=Value
```

- Instance-specific objects by using expressions like the following one to set `Field=Value` for unique object `Top0.instanceX`

```
% ./simv +vmm_opts+Field=Value@Top0.instanceX
```

- Hierarchical objects by using expressions like the following one to set `Field=Value` for all objects under `Top0` that contain option `Field`

```
% ./simv +vmm_opts+Field=Value@Top0
```

Specifying placeholders for hierarchical options

Configurations are usually modeled as a class and correspond to a container where all possible options are defined as data members.

The static methods `vmm_opts::get_object_*` assign a specific data member with a value from the `vmm_opts::set_*` methods.

The following example shows a configuration where 2 members: boolean `b` and integer `i` are flagged with `B` and `I` tags respectively, and the `is_set` variable is set when the option is overridden from command line. This is handy to find out whether a used value is a default one or comes from the command line.

```
class B extends vmm_object;

   bit b;
```

```
  int i;
  function new(string name, vmm_object parent=null);
    bit is_set;
    super.new(parent,name);
    b = vmm_opts::get_object_bit(is_set, this, "B",
                                 "SET b value", 0);
    i = vmm_opts::get_object_int(is_set, this, "I", 0,
                                 "SET i value", 0);
  endfunction
endclass
```

Setting Options in Code Block

Configurations can be set from the simulator command line or a file. These configurations can be set on an instance basis or hierarchically by using regular expressions.

The following example shows how to assign configuration members: boolean `b` and integer `i` in a test. This is made possible by using `vmm_opts::set_int` and `vmm_opts::set_bit` in the program block. Of course, these assignments could be anywhere in `vmm_timelines` or in `vmm_test:set_config()`

```
initial begin
  B class_b0 = new("class_b0", null);
  B class_b1 = new("class_b1", null);
```

```
    vmm_opts::set_bit("class_b0:b",null);
    vmm_opts::set_int("class_b0:i",null);
    vmm_opts::set_bit("class_b1:b",null);
    vmm_opts::set_int("class_b1:i",null);

    $display(" Value of class_b0:b is %0b", class_b0.b);
    $display(" Value of class_b0:i is %0d", class_b0.i);
    $display(" Value of class_b1:b is %0b", class_b1.b);
    $display(" Value of class_b1:i is %0d", class_b1.i);
  end
```

Note that it's also possible to set configurations that only belong to a given hierarchy, for instance the following line assigns `B` configurations for all `b*` matching objects that are under the `d2` root object:

```
vmm_opts::set_int("%b*:B", 99, d2);
```

Setting Options on Command Line

Once configurations have been defined, it's possible to change their values from:

- the simulator command line with `+vmm_opts+Field=Value` or `+vmm_Field=Value`

- An option file with the following syntax for assigning `d2:b1.b=88`, all `d1.*.b=99`, `i=1'b0` globally and `c2.b1.str="NEW_VAL2"`

```
+B =88@d2:b1

+B =99@d1*

+I = 0

+STR=NEW_VAL2@c2:b1
```

The following example shows how it's the default values that are returned when no options are specified on the command line:

```
% ./simv

Chronologic VCS simulator copyright 1991-2008

Contains Synopsys proprietary information.


 Value of class_b0:b is 0

 Value of class_b0:i is 0

 Value of class_b1:b is 0

 Value of class_b1:i is 0

Simulation PASSED on /./ (/./) at                    0
(0 warnings, 0 demoted errors & 0 demoted warnings)

   V C S   S i m u l a t i o n   R e p o r t

Time: 0

CPU Time: 0.020 seconds; Data structure size:   0.0Mb
```

The following example shows how to globally assign values for boolean `b=1'b1` and integer `i=10`:

```
% ./simv +vmm_opts+I=10 +vmm_B=1

Chronologic VCS simulator copyright 1991-2008

Contains Synopsys proprietary information.


 Value of class_b0:b is 1

 Value of class_b0:i is 10

 Value of class_b1:b is 1

 Value of class_b1:i is 10

Simulation PASSED on /./ (/./) at                    0
(0 warnings, 0 demoted errors & 0 demoted warnings)

   V C S   S i m u l a t i o n   R e p o r t

Time: 0

CPU Time: 0.030 seconds; Data structure size:   0.0Mb
```

The following example shows how to assign values for boolean `class_b0.b=1'b1` and integer `class_b1.i=10`:

```
% ./simv +vmm_opts+I=10@class_b1 +vmm_B='1@*b0'

Chronologic VCS simulator copyright 1991-2008

Contains Synopsys proprietary information.


 Value of class_b0:b is 1
```

```
  Value of class_b0:i is 0

  Value of class_b1:b is 0

  Value of class_b1:i is 10

  Simulation PASSED on /./ (/./) at                    0
(0 warnings, 0 demoted errors & 0 demoted warnings)

    V C S   S i m u l a t i o n   R e p o r t

  Time: 0

  CPU Time: 0.020 seconds; Data structure size:   0.0Mb
```

More Options

Available methods for getting options are:

```
function bit          get_bit(input string name)
function string      get_string(input string name)
function int          get_int(input string  name)
function void         get_range(input  string name)
function vmm_object get_obj(output bit is_set)
function bit          get_object_bit(output bit is_set)
function string     get_object_string(output bit is_set)
function int          get_object_int(output bit is_set)
function void        get_object_range(output bit is_set)
function vmm_object get_object_obj(output bit is_set)
```

Available methods for setting options are:

```
function void  set_bit(input string name, input bit val)
function void  set_string(input string name,
                          input string val)
function void  set_int(input string name,
                          input int val)
function void  set_range(input string name,
                          input int min, input int max)
function void  set_object(input string name,
                          input vmm_object obj)
```

Please refer to the VMM Reference Guide for further information.

Structural Configuration using Options

Structural configuration is an important aspect of verification environment composition. This is usually required for dynamically building verification components based upon configurations specified either on the command line or in a command file. These configurations can be set on an instance basis or hierarchically by using regular expressions.

Configuration parameters that affect the structure of the environment itself must be set during the "`build`" phase and be implemented in the `vmm_unit::build_ph()` method. These configuration parameters can be specified using options, but are typically set using RTL configuration parameters. Because the

`vmm_unit::build_ph()` methods are invoked in a top-down order, procedural parameter settings from higher-level modules will supersede procedural parameter settings from lower-level modules.

Due to the nature of structural configurations, there is no need for automatic randomization of structural configuration parameters.

The use model of structural configuration is similar to hierarchical configurations except that specific `vmm_unit` short-hand macros must be used to instrument transactors that extend the `vmm_unit` base class.

Structural configuration parameters can be set from three different sources, in order of increasing priority: within the code itself using `vmm_opts::set_*()` methods, external option files and command-line options. These parameters are set by explicitly calling the `vmm_opts::get_*()` methods in `vmm_timeline` or environment.

Here are the use models available for specifying a structural configuration:

- Global configurations by using expressions like the following one to set `Field=Value` for all objects that contain option `Field`

```
% ./simv +vmm_opts+Field=Value
```

- Instance-specific objects by using expressions like the following one to set `Field=Value` for unique object `Top0.instanceX`

```
% ./simv +vmm_opts+Field=Value@Top0.instanceX
```

- Hierarchical objects by using expressions like the following one to set `Field=Value` for all objects under `Top0` that contain option `Field`

```
% ./simv +vmm_opts+Field=Value@Top0
```

Specifying Structural Configuration Parameters in Transactors

Structural configuration declarations should sit in the transactor that extends `vmm_unit`.

A pre-defined set of shorthand macros can be used to attach structural configuration parameters to transactor structural tags, which can be accessed either from the command line or a command file.

Here are the available `vmm_unit` short hand macros:

```
`vmm_unit_config_begin(transactor)
   `vmm_unit_config_int(     int_data_member,”doc",
                                def_value, transactor)
  `vmm_unit_config_boolean(boolean_data_member,"doc",
                                def_value, transactor)
   `vmm_unit_config_string( string_data_member,"doc",
                                def_value, transactor)
`vmm_unit_config_end(transactor)
```

The following example shows a structural configuration where 3 data members: {boolean `b`, integer `i`, string `s`} are tagged with the {`B`, `I`, `S`} keywords respectively.

```
class vip1 extends vmm_unit;
```

```
    `vmm_typename(vip1)
    int i;
    bit b;
    string s;

    function new(string inst, vmm_unit parent = null);
        super.new(get_typename(), inst, parent);
    endfunction

  `vmm_unit_config_begin(vip1)
     `vmm_unit_config_int(i,1,"doc",0,vip1)
     `vmm_unit_config_boolean(b,"doc",0,vip1)
     `vmm_unit_config_string(s,"doc", ”null”,vip1)
  `vmm_unit_config_end(vip1)
endclass
```

Setting Structural Configuration Parameters

Structural configuration parameters can now be set from the simulator command line or a file. These configurations can be set on an instance basis or hierarchically by using regular expressions.

The following example shows how to assign the `v1` configuration members: {boolean `b`, integer `i`, string `s`} in a test. This is made possible by using `vmm_opts::set_int` and `vmm_opts::set_bit` in the program block.

From a user standpoint, there is no difference in setting structural or hierarchical configurations.

Of course, these assignments could be anywhere in `vmm_timeline` or in `vmm_test:set_config()`

```
initial begin

    vip1 v1;

    vmm_opts::set_bit("v1:b",1);

    vmm_opts::set_int("v1:i",2);

    vmm_opts::set_string("v1:s",”Burst”);


    v1 = new("v1");

    v1.configure_ph;

    $display("v1.i=%0d, v1.b=%0d”, v1.i, v1.b);

end
```

Setting Options on Command Line

Once configurations have been defined, it’s possible to change their values from:

- The simulator command line with `+vmm_opts+Field=Value` or `+vmm_Field=Value`
- An option file with following syntax for assigning `d2:b1.b=88`, all `d1.*.b=99`, `i=1'b0` globally and `c2.b1.str="NEW_VAL2"`

```
+B =88@d2:b1

+B =99@d1*

+I = 0

+STR=NEW_VAL2@c2:b1
```

The following example shows how to assign values for boolean `v1.b=1'b0` and integer `v1.i=9`:

```
./simv +vmm_b=0 +vmm_opts+i=9@v1
```

RTL Configuration

RTL configuration is an important aspect for ensuring that the RTL and testbench share the same configuration. This can be handy for sharing parameters such as:

- Number of input ports for a given protocol
- Number of output ports for a given protocol
- Architectural parameters like FIFO sizes, DMA capabilities, IRQs
- Latency, bandwidth limitations, etc.
- Specific operating modes

As opposed to hierarchical or structural configurations, RTL configuration solely depends on an input file that describes available options for a given instance.

Here are the key features supported by this set of VMM base classes:

- Support configurable RTL
- Support RTL configuration with randomized / directed parameters
- Support functional coverage of configuration
- Support composition of RTL configurations
- Support multiple instances of the same RTL module with different configurations
- Support partially-specified configurations

RTL configuration is performed using compile-time conditional code (i.e. `` `ifdef/`endif `` or parameter values, all of which are set before simulation runs. It is therefore impossible to randomize RTL configuration in the same simulation run as well as to run the test that will verify that configuration.

A two-pass process must be used:

- 1st pass to generate the RTL configuration to use. This can be manual or external to VCS
- 2nd pass to verify that configuration. This pass may be repeated multiple times to apply multiple tests to the same configuration. During this pass RTL and testbench are compiled using RTL configuration as created in 1st pass

The (multiple) second pass must not depend on random stability to reproduce the same RTL configuration. Instead, it should depend on a configuration specification file that is read in to set the RTL configuration parameters. This enables the RTL configuration to be specified manually, not only randomly.

Defining RTL Configuration Parameters

RTL configuration parameters should be declared in a transactor configuration that extends the `vmm_rtl_config` base class. Note that this transactor configuration acts as a data member container and is not supposed to be run.

A pre-defined set of shorthand macros can be used to attach RTL configuration parameters to transactor RTL tags, which can be accessed either from the command line or a command file.

Here are the available `vmm_rtl_config` shorthand macros:

```
`vmm_rtl_config_begin(transactor_config)

`vmm_rtl_config_int(RTL_config_name, RTL_config_fname)

`vmm_rtl_config_boolean(RTL_config_name,RTL_config_fname)

`vmm_rtl_config_string(RTL_config_name,RTL_config_fname)

`vmm_rtl_config_end(transactor_config)
```

The following example shows how to model a configuration where RTL configuration parameters: {boolean `mst_enable`, integer `addr_width`} are tagged with {`mst_enable`, `mst_width`} keywords respectively. By default, these data members can be randomized and associated with user-specific constraints:

```
class ahb_master_config extends vmm_rtl_config;
    rand  int addr_width;
    rand  bit mst_enable;
    string    kind = "MSTR";

    constraint cst_mst {
      addr_width == 64;
      mst_enable == 1;
    }
    `vmm_rtl_config_begin(ahb_master_config)
        `vmm_rtl_config_int(addr_width, mst_width)
        `vmm_rtl_config_boolean(mst_enable, mst_enable)
        `vmm_rtl_config_string(kind, kind)
    `vmm_rtl_config_end(ahb_master_config)

   function new(string name = "", vmm_rtl_config parent
= null);
        super.new(name, parent);
    endfunction
```

```
endclass
```

Using RTL Configuration in vmm_unit Extension

A transactor can simply refer to the RTL configuration by declaring a handle to this class that gets associated in the `vmm_unit::configure()` phase.

The static method `vmm_rtl_config::get_config` can be used to handle this association.

The following example shows how to associate a previously declared `ahb_master_config` object within a transactor:

```
class ahb_master extends vmm_unit;
    ahb_master_config cfg;

    function new(string name, vmm_unit parent = null);
        super.new(get_typename(), name, parent);
    endfunction

    function void configure_ph();
        $cast(cfg, vmm_rtl_config::get_config(this);
    endfunction
endclass
```

Instantiation in Top Environment

Once the transactor is instantiated in its enclosing environment, it must be properly constructed and associated with the right RTL configuration file.

This assumes that a RTL configuration file with name like "`INST.rtlconfig`" was previously created using the `+vmm_gen_rtl_config` 1st pass (see section below).

The following example shows how to map the previously declared `ahb_master` transactor with the right RTL configuration file name:

```
class env extends vmm_unit;
    ahb_master mstr;

    function new(string name, vmm_unit parent = null);
        super.new(get_typename(), name, parent);
    endfunction

    function void build_ph();
       mstr = new("mst", this);
      env_cfg.map_to_name("^");
      env_cfg.mst_cfg.map_to_name("env:mst");
    endfunction
endclass
```

1st pass: Generation of RTL Configuration Files

The 1st pass to generate the RTL configuration can take place once the transactor configuration is ready.

The file generation is activated when running the simulation with `+vmm_gen_rtl_config` option.

In this case, the simulator considers all objects that extend `vmm_rtl_config` base class. During this phase, all transactor configurations are created, randomized and their content is dumped to multiple RTL configuration files. No simulation is run during this pass.

The following example shows how to create RTL configuration files by prefixing all output files with '`RTLCFG`':

```
% ./simv +vmm_rtl_config=RTLCFG +vmm_gen_rtl_config

% more RTLCFG:env_cfg:mst_cfg.rtl_conf

mst_width  : 64;

mst_enable : 1;

kind        : MSTR;
```

2nd pass: Simulation using RTL Configuration File

The following example shows how to kick off a simulation by reading all RTL configuration files that are prefixed with '`RTLCFG`' shows:

```
./simv +vmm_rtl_config=RTLCFG
```

5

OSCI TLM 2.0 Interoperability (vmm_tlm*)

For users who are unaware of the complexities and nuances of OSCI-TLM2.0, please visit the OSCI website and read through the OSCI-TLM2.0 user manual before proceeding. Users who want full transaction level performance with a single usage model API should use `vmm_channel for all object connectivity.

The OSCI-TLM2.0 User Manual describes the Blocking Transport in *Chapter 4* of the TLM_2_0_user_manual.pdf included in the OSCI-TLM tarball available at: http://www.systemc.org/members/download_files/check_file?agreement=tlm_2-0_080606

The SystemVerilog VMM implementation of the OSCI-TLM 2.0 API is explained in the VMM-Class Reference Guide. Please refer to `vmm class reference guide` for further information.

Transport Interfaces in OSCI TLM2.0

VMM supports the following SystemVerilog flavors of transport interfaces in OSCI TLM 2.0:

- Blocking transport
- Non-blocking transport
- Sockets
- Generic Payload

Blocking Transport

As you will read in the OSCI-TLM2.0 user manual:

“The new TLM-2 blocking transport interface is intended to support the loosely-timed coding style. The blocking transport interface is appropriate where an initiator wishes to complete a transaction with a target during the course of a single function call, the only timing points of interest being those that mark the start and the end of the transaction. The blocking transport interface only uses the forward path from initiator to target.”

Due to its loosely timed application with single `vmm_tlm_blocking_transport_if` the blocking transport interface is considered more simplistic than non-blocking transports and only implements the forward path port called `vmm_tlm_blocking_transport_port`.

The following snippets of SystemVerilog code show how the parent-child association is built up during construction.

Firstly, instance the port in the initiator and call the `b_transport()` method from within the port

```
class initiator extends vmm_xactor;
   vmm_tlm_b_transport_port#(initiator,my_trans, vmm_tlm)
   b_port = new(this,”initiator_port”);
   ...
   virtual protected task main();
   ...
   b_port.b_transport(trans,ph);      // send transaction
   endtask: main
endclass : initiator
```

Secondly, instance the export in the target and implement the `b_transport()` functionality locally

```
class target extends vmm_xactor;
   vmm_tlm_b_transport_export#(target,my_trans,vmm_tlm)
   b_export = new(this,”target_export”);
...
   task b_transport(int id = -1, my_trans trans,
                    vmm_tlm ph, int delay = 0);
//execute transaction
     trans.display(“From Target”);
   //finish completion model
```

```
      #5s ph.sync = vmm_tlm::TLM_COMPLETED;
   endtask : b_transport
endclass : target
```

Thirdly, instance the initiator / target and bind the export with the port:

```
class my_env extends vmm_env;
   initiator initiator0; target target0;
   function build();
   //bind port->export
      initiator0.b_port.tlm_bind(target0.b_export);
```

Non-blocking Transport

As you will read in the OSCI-TLM2.0 user manual:

“The non-blocking transport interface is intended to support the approximately-timed coding style. The nonblocking transport interface is appropriate where it is desired to model the detailed sequence of interactions between initiator and target during the course of a each transaction. In other words, to break down a transaction into multiple phases, where each phase transition marks an explicit timing point.”

Both forward and backward directions are available in the non-blocking transport called `vmm_tlm_fw_nonblocking_transport_if` and `vmm_tlm_bw_nonblocking_transport_if` respectively. These classes are virtual and used as foundation for other TLM transport interfaces as described hereafter.

The following is an example of how to use a non-blocking forward port for non-blocking transportation.

Firstly, instance the port in the initiator and call the nb_transport_fw() API from within the port

```
class initiator extends vmm_xactor;

vmm_tlm_nb_transport_port#(initiator,my_trans,vmm_tlm)

                nb_port = new(this,”initiator_port”);

   …

   virtual protected task main();

   …

       nb_port.nb_transport_fw(trans,ph);

   endtask: main

endclass : initiator
```

Secondly, instance the export in the target and implement the nb_transport_fw() functionality locally

```
class target extends vmm_xactor;

   vmm_tlm_nb_transport_export#(target,my_trans,vmm_tlm)

                nb_export = new(this,”target_export”);

…

   function vmm_tlm::sync_e nb_transport_fw(int id=-1,

             my_trans trans, vmm_tlm ph, int delay=0);
```

```
   //execute transaction
   trans.display(“From Target”);
   //finish completion model
      return vmm_tlm::TLM_ACCEPTED;
   endfunction : nb_transport
endclass : target
```

Thirdly, instance the initiator and target and bind the nb_export with the nb_port.

```
class my_env extends vmm_env;
   initiator initiator0; target target0;
   …
   function build();
   …
   //connectivity
      initiator.nb_port.tlm_bind(target0.nb_export);
   endfunction : build
endclass : my_env
```

Sockets

OSCI-TLM 2.0 makes use of sockets to communicate between transaction level elements. A similar set of methods are available to VMM users thereby lowering the learning curve for SystemC engineers to understand how VMM objects can be connected to fulfill necessary communication completion models.

Sockets group together all the necessary core interfaces for transportation and binding thereby simplifying and allowing more generic usage models than just TLM core-interfaces.

The OSCI-TLM 2.0 says usage of TLM core-interfaces without sockets is not recommended. However, the socket infrastructure restricts the binding model and in SystemVerilog we need to implement all functions even if they are not used. This can be seen as unnecessary and the flexibility of the core-interfaces is more suitable for verification connection models.

The `vmm_tlm_simple_initiator_socket` and `vmm_tlm_simple_target_socket` are generic convenience sockets ready for you to use. These sockets can be used as blocking or as non-blocking transportation mechanisms.

Firstly, instance an initiator socket in the initiator and call the `nb_transport_fw` method. Note, the backward path function `nb_transport_bw` must be implemented even if it isn't used, as other sockets could call this function.

```
class initiator extends vmm_xactor;

   vmm_tlm_simple_initiator_socket#(

                  initiator,my_trans,vmm_tlm) socket=
```

```
                                new(this, "initiator_put");

   virtual function vmm_tlm::sync_e nb_transport_bw(
   int id=-1, my_trans trans, vmm_tlm ph, int delay);
   endfunction : nb_transport_bw
   virtual protected task main();
   …
      socket.nb_transport_fw(trans);
   endtask: main
endclass: initiator
```

Secondly, instance a target socket in the target and implement the `nb_transport_fw()` functionality locally. Note, the `b_transport()` task must be implemented even if not used as other socket could call this.

```
class target extends vmm_xactor;
vmm_tlm_simple_target_socket#(target, my_trans, vmm_tlm)
                      socket = new(this, "target_put");

   virtual function vmm_tlm::sync_e nb_transport_fw(
int id = -1, my_trans trans, vmm_tlm ph, int delay);
   //execute transaction
```

```
      trans.display(“From Target”);
   //finish completion model
      return vmm_tlm::TLM_ACCEPTED;
   endfunction: nb_transport_fw
   virtual task b_transport(int id = -1, my_trans trans,
               vmm_tlm ph = null , int delay = -1 );
   endtask : b_transport
endclass : target
```

Thirdly, instance the initiator and target and bind the sockets together.

```
class env extends vmm_env;
   initiator initiator0;  target target0;
   function build();
       initiator0.socket.tlm_bind(target0.socket);
   . . .
```

Generic Payload

You should derive a transaction from `vmm_data` to have complete control over the data object and an abstract implementation that can be reused throughout the environment. This `vmm_data` can be used with all objects including generators and channels.

The `vmm_tlm_generic_payload` is derived from `vmm_rw_access` and is predominantly used to simplify users bringing existing TLM SystemC generic payload objects into VMM environments.

Below is an example showing the use of generic payload where the initiator class has a bi-directional non-blocking port parameterized on `vmm_tlm_generic_payload`:

The initiator class below has a bi-directional non-blocking port parameterized on vmm_tlm_generic_payload.

```
class initiator extends vmm_xactor;
   vmm_tlm_nb_transport_port#(initiator,
                   vmm_tlm_generic_payload, vmm_tlm)
                 nb_port = new(this,”initiator_port”);
   …
   virtual protected task main();
   vmm_tlm_generic_payload trans;
   …
    nb_port.nb_transport_fw(trans,ph);
   endtask: main
   function vmm_tlm::sync_e nb_transport_bw(int id=-1,
                   vmm_tlm_generic_payload trans,
                   vmm_tlm ph, int delay);
   endfunction
```

```
endclass : initiator
```

Secondly, the target below which connects to the port of the initiator should also use the `vmm_tlm_generic_payload`.

```
class target extends vmm_xactor;
  vmm_tlm_nb_transport_export#(target,
                vmm_tlm_generic_payload, vmm_tlm)
              nb_export = new(this,”target_export”);
  …
  function vmm_tlm::sync_e nb_transport_fw(int id= -1,
                vmm_tlm_generic_payload  trans,
                vmm_tlm ph,int delay = 0);
  //execute transaction
    trans.display(“From Target”);
  //finish completion model
    return vmm_tlm::TLM_ACCEPTED;
  endfunction : nb_transport
endclass : target
```

Interoperability between vmm_channel and TLM2.0

VMM provides a methodology for connecting `vmm_xactors` with `vmm_xactors` using a channel interface to vmm_xactors with. Conversely it is also possible to have TLM2.0 interfaces directly connected to `vmm_channel`. The `vmm _channel` can be connected to the blocking transport interface, non-blocking forward interface, non-blocking bidirectional interface, analysis interface. The `vmm_channel` can act like a producer by binding the channel's TLM port to an external TLM export or as a consumer by binding the channel's TLM export to an external TLM port.

`vmm_channel Usage & vmm_channel_typed

Below is an example to showthat shows how to connect a consumer with a `vmm_channel` to a producer with a TLM blocking port.

Producer In this case, producer with a blocking transport port calling the `b_transport` method of the blocking port.

```
class initiator extends vmm_xactor;
    vmm_tlm_b_transport_port#(initiator,my_trans)
                        b_port=new(this,”initiator_port”);
    task main();
        …
        b_port.b_transport(tr);
    endtask
endclass: initiator
```

Target with a `vmm_channel` instantiated using the `vmm_channel_typed` class

```
class target extends vmm_xactor;

  #(my_trans) in_chan = new(“target”,”in_chan”);

  virtual task main();

      in_chan.get(tr);

      …

   endtask

endclass : target
```

Optionally, target with a `vmm_channel` instantiated using the ` `vmm_channel` macro

```
`vmm_channel(my_trans)

 class target extends vmm_xactor;

    my_trans_channel in_chan = new(“target”,”in_chan”);

    virtual task main();

        in_chan.get(tr);

        …

    endtask

endclass : target
```

Then bind the channel's blocking transport export to the blocking transport port of the initiator using the `tlm_bind` method of the `vmm_channel` class.

```
class subenv extends vmm_subenv;

    initiator i0, target t0;

    function build();

t0.in_chan.tlm_bind(i0.b_port,vmm_tlm::TLM_BLOCKING_EXPORT)
;

    endfunction

endclass : subenv
```

TLM2.0 Accessing Generators

The VMM atomic and scenario generators have an built-in `vmm_channel` called `out_chan`. The output channel's blocking or non-blocking forward transport port can be connected to a consumers blocking or non-blocking forward export.

Below is an example showing how to use an atomic generator with blocking transport export.

Firstly, have a consumer class with a blocking transport export.

```
class driver extends vmm_xactor;

   vmm_tlm_b_transport_export#(driver,my_trans)

                b_export=new(this,"driver_export");
```

```
   task b_transport(int id=-1, my_trans trans,
                    vmm_tlm ph, int delay);
        …
  //process the transactions received from the generator
   endtask
  endclass: driver
```

Then create an atomic generator using the short hand macro.

```
`vmm_atomic_gen(my_trans,”atomic_gen”)
```

Then instantiate the driver and atomic generator and bind the generators blocking transport port to the drivers blocking transport export using tlm_bind.

```
class env extends vmm_env;
   my_trans_atomic_gen gen;
   driver d0;
   function build();
      gen = new(“Simple Atomic Gen”,0);
      d0  = new(this,”d0”);

gen.out_chan.tlm_bind(d0.b_export,vmm_tlm::TLM_BLOCKING_POR
T);
   endfunction
```

```
endclass : env
```

Forward Path Non-Blocking Connection

Below is an example showing how to use the `vmm_channel` with a non-blocking transport connection on the forward path. The transactor with the `vmm_channel` is the producer connected to the non-blocking forward export of the consumer.

Firstly create a producer class with a `vmm_channel`. The short-hand macro ` `vmm_channel` or `vmm_channel_typed` class can be used.

```
`vmm_channel(my_trans)

class initiator extends vmm_xactor;

my_trans_channel out_chan = new(“initiator”,”out_chan”);

     //OR USE

vmm_channel_typed#(my_trans) in_chan = new (“target”,
”out_chan”);

   virtual task main();

   out_chan.put(tr);

   …

   endtask

endclass : initiator
```

Then create a consumer class with a non-blocking forward transport export.

```
class target extends vmm_xactor;
  vmm_tlm_nb_transport_fw_export#(target,my_trans)
              nb_export=new(this,”target_export”);
  function vmm_tlm::sync_e nb_transport_fw(int id=-1,
              my_trans trans, vmm_tlm ph, int delay);
  …
//process the transactions received from the initiator
  endfunction
endclass: target
```

Then connect the non-blocking forward transport port of the channel to the non-blocking forward export of the target.

```
class env extends vmm_env;
  initiator i0;
  target t0;
  function build();
  …
    i0.out_chan.tlm_bind(t0.nb_export,
                    vmm_tlm::TLM_NONBLOCKING_FW_PORT);
  endfunction
```

```
endclass : env
```

Bidirectional Non-Blocking Connection

Below is an example showing how to connect a consumer with a `vmm_channel` to a producer with a TLM non-blocking bi-directional port. In this case, producer with a blocking transport port calls the `b_transport` method of the blocking port.

```
class initiator extends vmm_xactor;
   vmm_tlm_nb_transport_port#(initiator,my_trans)
                    nb_port=new(this,”initiator_port”);
   task main();
   …
      nb_port.nb_transport_fw(tr);
   endtask
   function vmm_tlm::sync_e nb_transport_bw(int id=-1,
               my_trans trans,vmm_tlm ph, int delay);
// method is called when target notifies vmm_data::ENDED
// on a particular transaction
   endfunction
endclass: initiator
```

Below is an example that shows how to model a target with a `vmm_channel` instantiated using the `vmm_channel_typed` class.

```
class target extends vmm_xactor;

    vmm_channel_typed#(my_trans) in_chan =

                                    new(“target”,”in_chan”);

   virtual task main();

     in_chan.get(tr);

   …

//calls the nb_transport_bw

     tr.notify.indicate(vmm_data::ENDED);

                    // method of the initiator with the
current transaction

    endtask

endclass : target
```

Then bind the channel’s non-blocking bi-directional export to the non-blocking bi-directional port of the initiator using the `tlm_bind` method of the `vmm_channel` class.

```
class subenv extends vmm_subenv;

    initiator i0, target t0;

    function build();

        …
```

```
t0.in_chan.tlm_bind(i0.nb_port,vmm_tlm::TLM_NONBLOCKING_EXP
ORT);

      endfunction

   endclass : subenv
```

Broadcasting using TLM2.0

Analysis ports are useful to broadcast transactions to multiple observers like scoreboards and functional coverage models. Analysis ports can be bound to multiple observers and analysis exports can be bound to multiple producers.

As you will read in the OSCI-TLM2.0 manual:

Analysis ports are intended to support the distribution of transactions to multiple components for analysis, meaning tasks such as checking for functional correctness or collecting functional coverage statistics. The key feature of analysis ports is that a single port can be bound to multiple channels or subscribers such that the port itself replicates each call to the interface method **write** with each subscriber. An analysis port can be bound to zero or more subscribers or other analysis ports, and can be unbound. Each subscriber implements the **write** method of the **tlm_analysis_if**.

Analysis Port Usage with Many Observers

Below is an example to show the usage of an analysis port connected to many observers:

Firstly, instance the `analysis_port` within the transmitter and call `write`() function:

```
class monitor extends vmm_xactor;
   vmm_tlm_analysis_port#( monitor,my_trans)
        analysis_port=new(this,”target_analysis_port”);
   …
   task perform_update();
// Transmit trans to observers
     analysis_port.write(trans);
   endtask
endclass: monitor
```

Secondly, instance the `analysis_export` within the observer and implement the `write()` functionality:

```
class observer extends vmm_object;
   vmm_tlm_analysis_export#( observer,my_trans)
          scb_analysis= new(this,”observing_analysis”);
   …
   virtual function write(int id= -1, my_trans trans);
//operation on transaction received.
     trans.display(“”);
   endfunction
```

```
endclass : observer
```

Optionally, instance the `analysis_export` within different observers and implement the `write()` functionality:

```
class cov_model extends vmm_object;
   …
   vmm_tlm_analysis_export#(cov_model,my_trans)
             cov_analysis= new(this,”coverage_analysis”);

covergroup covg with function sample(my_trans incoming);
     coverpoint incoming.rw;
   endgroup
   covg cg=new();
   …
   virtual function write(int id= -1, my_trans trans);
     this.cg.sample(trans);
   endfunction : write
endclass: cov_model
```

Thirdly, instance the objects and connect the ports:

```
class env extends vmm_env;
```

```
   …
   monitor  mon0; observer observe0; cov_model cov_inst;
   virtual function void build();
   …
   target0.analysis_port.tlm_bind(observe0.scb_analysis);
   target0.analysis_port.tlm_bind(cov_inst.cov_analysis);
   endfunction : build
endclass: my_env
```

Analysis Port Multiple Ports per Observer

There is no restriction with OSCI-TLM2.0 to limit the number of observer hooks using `analysis_export` per observation class. However in SystemVerilog you can only have one implementation of a function present in a class. Therefore, if you have two `analysis_exports` these would use the same `write`() implementation.

If the observer requires a unique implementation of write method for each port then either multiple analysis exports can be instantiated in the observer with a unique implementation of write for each binding using the short-hand macro or multiple ports can be connected to the same export instance using peer id.

Shorthand Macro ID’s

Below is an example to show how to use multiple `analysis_exports` within a single observer.

Firstly, instance the `analysis_port` within the transmitter and call the `write`() function.

```
class monitor extends vmm_xactor;
   …
   vmm_tlm_analysis_port#( monitor,my_trans)
   analysis_port = new(this,”monitors_analysis_port”);
   task perform_update()
      analysis_port.write(trans);
   endtask
   …
endclass : monitor
```

Secondly, instance two `analysis_exports` within the observer and implement the `write<ID>()` functionality.

```
class scoreboard extends vmm_object;
   `vmm_tlm_analysis_export(_1) //uniquifier ID
   `vmm_tlm_analysis_export(_2) //uniquifier ID
   vmm_tlm_analysis_export_1#(scoreboard,my_trans)
        scb_analysis_1=new(this,”scoreboard_analysis_1”);
   vmm_tlm_analysis_export_2#( scoreboard,my_trans)
        scb_analysis_2=new(this,”scoreboard_analysis_2”);
   …
```

```
  virtual function write_1(int id= -1, my_trans trans);
    `vmm_note(log,”From scoreboard write_1”);
  endfunction
  virtual function write_2(int id= -1, my_trans trans);
    `vmm_note(log,”From scoreboard write_2”);
  endfunction
endclass : scoreboard
```

Thirdly, instance the objects and bind the ports to respective places.

```
class env extends vmm_env;
  monitor       mon[2];
  scoreboard  scb;
  …
  virtual function void build();
    mon[0].analysis_port.tlm_bind(scb.scb_analysis_1);
    mon[1].analysis_port.tlm_bind(scb.scb_analysis_2);
  endfunction : build
endclass : my_env
```

Peer IDs

By using peer IDs there is only one `write()` implementation required and within it you can identify which port is performing the access, then execute the appropriate functionality.

The following example shows how to use `single_export` with `peer_id`.

Firstly, instance the `analysis_port` within the transmitter and call the `write()` function.

```
class monitor extends vmm_xactor;

   vmm_tlm_analysis_port#( monitor, my_trans)

        analysis_port=new(this,”monitor_analysis_port”);

   …

   task perform_update()

      analysis_port.write(trans);

   endtask

endclass: monitor
```

Secondly, instance one `analysis_export` within the observer and implement the `write()` functionality.

Maximum binding must be specified in the `analysis_export` constructor.

```
class scoreboard extends vmm_object;
```

```
   vmm_tlm_analysis_export#( scoreboard,my_trans)
      scb_analysis=new(this,”scoreboard_analysis”, 2, 0);
   …
   virtual function write(int id= -1, my_trans trans);
      case(id)
         0: do_compare_from_port0(trans);
         1: do_compare_from_port1(trans);
      endcase
   endfunction
endclass : scoreboard
```

Thirdly, instance the objects and bind the ports to respective places using peer IDs.

```
class env extends vmm_env;
   monitor  mon[2];  scoreboard  scb;
   …
   virtual function void build();
      …
      mon[0].analysis_port.tlm_bind(scb.scb_analysis, 0);
      mon[1].analysis_port.tlm_bind(scv.scb_analysis, 1);
   endfunction : build
endclass : my_env
```

Analysis Ports in vmm_callback

While the Peer-ID and short-hand unique macro provide adequate mechanisms for any number of analysis ports to be coded it doesn't solve the issue of which ones to have connected at any given time.

By making use of the well documented `vmm_callback` mechanism and placing the analysis port/export within a callback class the environment immediately benefits from having more flexibility. The callbacks can be appended depending on the verification requirements and any number of callbacks can be created.

One such usage for this would be to have a callback class from a monitor_xactor for performing scoreboarding. When you move to the system-level, the scoreboarding information may be required by other scoreboards in the system for checking data-integrity between various interfaces or abstraction levels or the original scoreboard may be required to receive information from a new source such as low-power logic that will affect the scoreboard comparison functionality.

Often when developing the verification block level environment there is not a need or foresight to implement all possible communication API's in all objects, nor is this required for vmm_callbacks. At the system-level the new callback classes can be used to have added functionality for the system-level. Thereby, allowing better reuse from the block-level environment to the system-level without affecting the original code or topology.

The following code shows the relevant parts for implementing analysis ports within callbacks even when not originally planned for upfront.

Firstly, create the callback you want and the default implementation.

```
class scoreboard extends vmm_xactor_callbacks;
  virtual task get_tx(my_trans trans);
    $display("GOT IT");endtask
endclass
```

Secondly, create the xactor and access the callback API

```
class TX_monitor extends vmm_xactor;
  . . .
  virtual task main();
    . . .
//callbacks API
    `vmm_callback(scoreboard,get_tx(trans));
  endtask
endclass
```

Thirdly, you can create more callbacks with various functionality

```
//connection from module verif to system verification
class scbd_system_connector extends scoreboard;
vmm_tlm_analysis_port#(scbd_system_connector, my_trans)
       TX_analysis_port=new(this,"send_to_system_scbd");
```

```
   virtual task get_tx(my_trans trans);
// send data to system-observers
     TX_analysis_port.write(trans);
   endtask
endclass
```

The callbacks are appended or prepended in the normal fashion. Only append/prepend the callbacks that are relevant for the current envionment. So below I don't need the "scoreboard" callback but do need the "scbd_system_connector"

```
class env extends vmm_env;
   . . .
   TX_monitor TX = new();
   scbd_system_connector top_scbd = new();
   function build();
     . . .
     TX.prepend_callback(top_scbd);
     . . .
```

Lastly, you can now have any number of analysis export listening to the callback analysis_port.write()

Updating Data in Analysis Port from vmm_notify

VMM has a default subscription based listener model based on `vmm_notify`. The VMM notification service (`vmm_notify`) can be used to connect a transactor (or channels or any other testbench component) to a scoreboard or functional coverage collector or any other passive observer. There can be multiple observers, and they will all see the same transaction stream.

There are pre-defined notification in `vmm_xactor` and `vmm_channel` readily available for review and use.

Firstly, configure your notification as per normal and call the indicate API as normal.

```
class monitor extends vmm_xactor;

   . . .

   int OBSERVED;

   function new(string name);
      this.OBSERVED=this.notify.configure();

   endfunction

   task main()

      . . .

      this.notify.indicate(this.OBSERVED, my_trans)

   endtask
. . .

endclass
```

Secondly, within a subscriber callback implement the indicated() functionality pass the info onto observer.

```
class subscribe extends vmm_notify_callbacks;

   . . .

   local observer obs;

   function new(observer obs);
    this.obs = obs;
  endfunction
  virtual function void indicated(vmm_data status);
     this.obs.observe(status);
  endfunction

   . . .

endclass
```

Thirdly, the observer class implements the observe() function which executes the `analysis_port.write`()

```
class observer;

   vmm_tlm_analysis_port#( subscribe, my_trans)

         analysis_port =
new(this,"observer_analysis_port");

   string name;

  function new(string name, vmm_notify ntfy, int id);
    subscribe cb = new(this);
   ntfy.append_callback(id, cb);
   this.name = name;
```

```
      endfunction

      function void observe(vmm_data tr);

         analysis_port.write(tr);

      endfunction
endclass
```

Lastly, you would instance the objects and bind the analysis port to any subscribing analysis_export. Thereby, anytime the vmm_notifier indicates the data object any analysis_exports would observe the data object.

6

Transaction and Environment Debugging

Starting in VCS 2009.12, you can dump the `vmm_data` class members which are registered using shorthand macros, set the transaction verbosity level and turn on/off the transaction dumping and the ability to get transactor states.

These action are done with system function `$tblog` and a `$vcdpluson` call that allows dumping of transactions or strings directly to the VPD file.

Features

Here are the key features that take advantage of `$tblog`

- Dump the data with various severity levels:
- NORMAL_SEV, TRACE_SEV, DEBUG_SEV, VERBOSE_SEV

- Dump channel name and description of transaction
- Dump notification ID each time the notification is indicated or reset. This includes the description of status descriptor in case it's associated with the indication
- Dump consensus name whenever participant is called, or any wait on the `vmm_voter` is called
- Dump name of `vmm_env` base class phase whenever a phase method is called.
- Dump sub-environment name and description of the state change request
- Dump timeline name and a description of the phase currently executing.
- Dump transactor name and a description of any state change or state change requests

The following base classes have built-in $tblog support and can be debugged in DVE

```
vmm_data
vmm_notify
vmm_channel
vmm_scenario
vmm_mss
vmm_scenario_gen
vmm_atomic_gen
vmm_rw_access
vmm_xactor
vmm_env
vmm_sub_env
```

```
vmm_timeline
vmm_simulation
vmm_consensus
```

Here is an example that shows how to dump info for `vmm_xactor` class and `vmm_env` class:

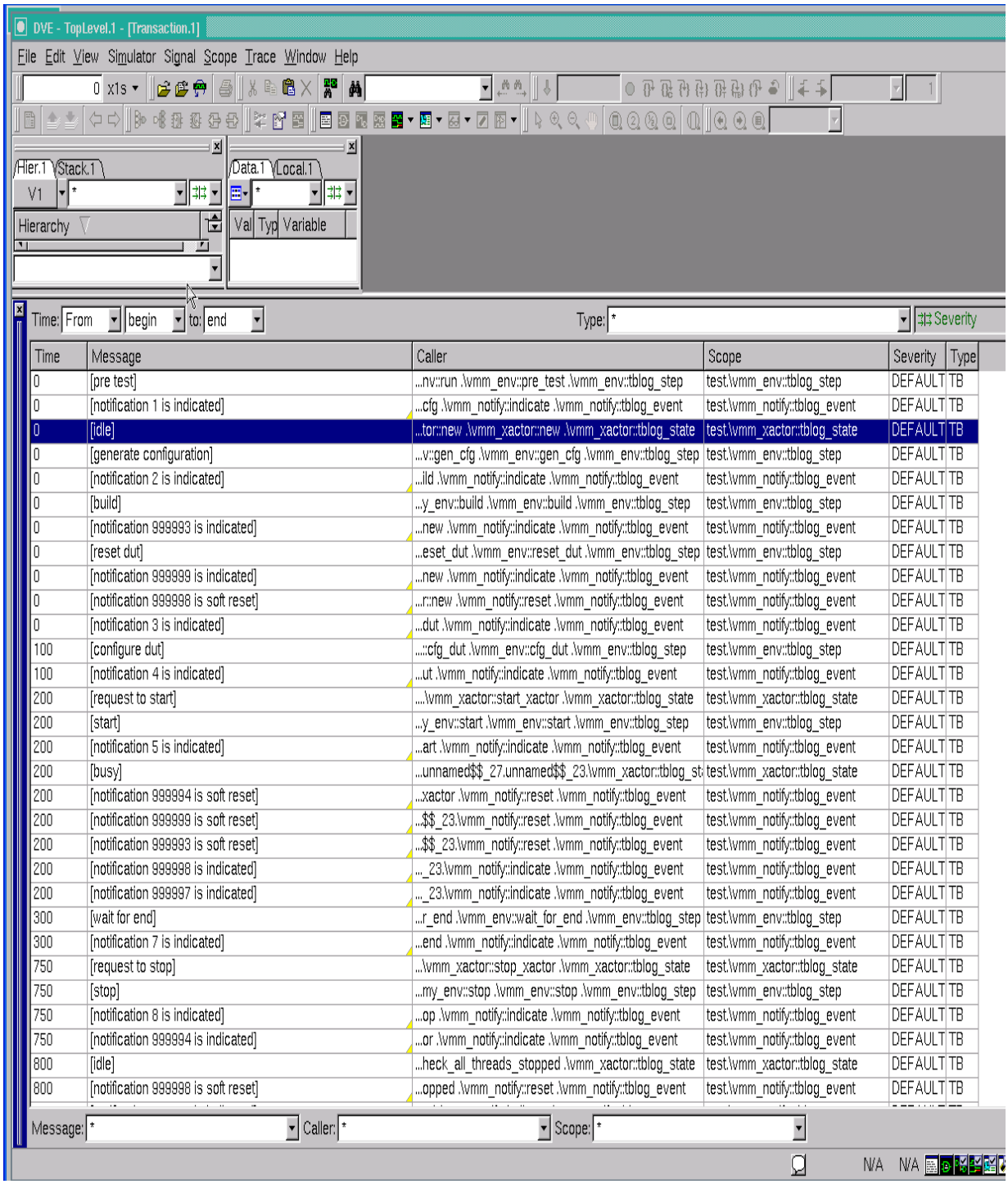
```
initial begin
    my_env env = new();
    env.run();
end


initial begin
    logic a=1;
    $vcdpluson();
end
```

VCS command line that is used to turn on `vmm_xactor` and `vmm_env` debugging is:

```
% ./simv +vmm_tr_verbosity=debug
```

The following snapshot shows various phases of the `vmm_env`, state of the `vmm_xactor`, and `vmm_notification` ID. These messages can be filtered out and individual messages that are in the transaction pane can be exported to the waveform viewer.


DVE - TopLevel.1 - [Transaction.1]
File Edit View Simulator Signal Scope Trace Window Help
Hier.1 Stack.1
V1
Hierarchy
Data.1 Local.1
Val Typ Variable
Time: From begin to: end
Type: *
Severity
Time Message Caller Scope Severity Type
0 [pre test] ...nv::run .\vmm_env::pre_test .\vmm_env::tblog_step test.\vmm_env::tblog_step DEFAULT TB
0 [notification 1 is indicated] ...cfg .\vmm_notify::indicate .\vmm_notify::tblog_event test.\vmm_notify::tblog_event DEFAULT TB
0 [idle] ...tor::new .\vmm_xactor::new .\vmm_xactor::tblog_state test.\vmm_xactor::tblog_state DEFAULT TB
0 [generate configuration] ...v::gen_cfg .\vmm_env::gen_cfg .\vmm_env::tblog_step test.\vmm_env::tblog_step DEFAULT TB
0 [notification 2 is indicated] ...ild .\vmm_notify::indicate .\vmm_notify::tblog_event test.\vmm_notify::tblog_event DEFAULT TB
0 [build] ...y_env::build .\vmm_env::build .\vmm_env::tblog_step test.\vmm_env::tblog_step DEFAULT TB
0 [notification 999993 is indicated] ...new .\vmm_notify::indicate .\vmm_notify::tblog_event test.\vmm_notify::tblog_event DEFAULT TB
0 [reset dut] ...eset_dut .\vmm_env::reset_dut .\vmm_env::tblog_step test.\vmm_env::tblog_step DEFAULT TB
0 [notification 999999 is indicated] ...new .\vmm_notify::indicate .\vmm_notify::tblog_event test.\vmm_notify::tblog_event DEFAULT TB
0 [notification 999998 is soft reset] ...r::new .\vmm_notify::reset .\vmm_notify::tblog_event test.\vmm_notify::tblog_event DEFAULT TB
0 [notification 3 is indicated] ...dut .\vmm_notify::indicate .\vmm_notify::tblog_event test.\vmm_notify::tblog_event DEFAULT TB
100 [configure dut] ...::cfg_dut .\vmm_env::cfg_dut .\vmm_env::tblog_step test.\vmm_env::tblog_step DEFAULT TB
100 [notification 4 is indicated] ...ut .\vmm_notify::indicate .\vmm_notify::tblog_event test.\vmm_notify::tblog_event DEFAULT TB
200 [request to start] ...\vmm_xactor::start_xactor .\vmm_xactor::tblog_state test.\vmm_xactor::tblog_state DEFAULT TB
200 [start] ...y_env::start .\vmm_env::start .\vmm_env::tblog_step test.\vmm_env::tblog_step DEFAULT TB
200 [notification 5 is indicated] ...art .\vmm_notify::indicate .\vmm_notify::tblog_event test.\vmm_notify::tblog_event DEFAULT TB
200 [busy] ...unnamed$$_27.unnamed$$_23.\vmm_xactor::tblog_st test.\vmm_xactor::tblog_state DEFAULT TB
200 [notification 999994 is soft reset] ...xactor .\vmm_notify::reset .\vmm_notify::tblog_event test.\vmm_notify::tblog_event DEFAULT TB
200 [notification 999999 is soft reset] ...$$_23.\vmm_notify::reset .\vmm_notify::tblog_event test.\vmm_notify::tblog_event DEFAULT TB
200 [notification 999993 is soft reset] ...$$_23.\vmm_notify::reset .\vmm_notify::tblog_event test.\vmm_notify::tblog_event DEFAULT TB
200 [notification 999998 is indicated] ..._23.\vmm_notify::indicate .\vmm_notify::tblog_event test.\vmm_notify::tblog_event DEFAULT TB
200 [notification 999997 is indicated] ..._23.\vmm_notify::indicate .\vmm_notify::tblog_event test.\vmm_notify::tblog_event DEFAULT TB
300 [wait for end] ...r_end .\vmm_env::wait_for_end .\vmm_env::tblog_step test.\vmm_env::tblog_step DEFAULT TB
300 [notification 7 is indicated] ...end .\vmm_notify::indicate .\vmm_notify::tblog_event test.\vmm_notify::tblog_event DEFAULT TB
750 [request to stop] ...\vmm_xactor::stop_xactor .\vmm_xactor::tblog_state test.\vmm_xactor::tblog_state DEFAULT TB
750 [stop] ...my_env::stop .\vmm_env::stop .\vmm_env::tblog_step test.\vmm_env::tblog_step DEFAULT TB
750 [notification 8 is indicated] ...op .\vmm_notify::indicate .\vmm_notify::tblog_event test.\vmm_notify::tblog_event DEFAULT TB
750 [notification 999994 is indicated] ...or .\vmm_notify::indicate .\vmm_notify::tblog_event test.\vmm_notify::tblog_event DEFAULT TB
800 [idle] ...heck_all_threads_stopped .\vmm_xactor::tblog_state test.\vmm_xactor::tblog_state DEFAULT TB
800 [notification 999998 is soft reset] ...opped .\vmm_notify::reset .\vmm_notify::tblog_event test.\vmm_notify::tblog_event DEFAULT TB
Message: * Caller: * Scope: *
N/A N/A

7

Utility Features and Classes

Custom Regular Expressions

Advanced regular expressions perform hierarchical name matching in a specific hierarchical namespace. Some of the key aspects of the advanced regular expressions supported with VMM1.2 include:

Introduction to VMM Custom Regular Expressions

- "`:`" used as hierarchical name separator, '.' character with no need to be escaped
- "@" used to indicate match pattern
- "`/`" used for usual regular expressions

A match pattern matches every character as-is, except for meta-characters, which match as follow:

- . Matches any one character, except ‘:’
- * Matches any number of characters, except ‘:’
- ? Matches zero or one character, except ‘:’
- % Matches zero or more colon-separated names, including the final colon

Pattern Matching Rules

The following table describes some pattern matching rules:

Pattern	Match	Doesn’t match
@%.	t t:s t:s:v	t:sub_env “.” matches any single character except “:” “%” matches one or more levels of hierarchy e.g., “t:”, “t:s:”, “t:s:v:”, etc., So the pattern matches any path ending with a single character as the last element
@%*	Matches any hierarchical path: top top:sub_env top:sub_env:vip	top: top:sub_env: top:sub_env:vip: “%” matches one or more levels of hierarchy e.g. “t:”, “t:s:”, “t:s:v:”, etc., “*” matches the occurrence of any number of any character except “:”

@%?	t t:s t:s:v t ::v	top top:sub_env top:sub_env:vip “?” matches the occurrence of zero or one character except “:” “%” matches one or more levels of hierarchy e.g. “t:”, “t:s:”, “t:s:v:”, etc., Note: difference between “@%.” and “@%?” is that the latter matches a null string whereas the former doesn’t.
@top:*:vip	top:sub_env0:vip top:sub_env1:vip	top:vip top:sub_env0:slice0:vip “*” matches the occurrence of any number of any character except “:”
@top:???:vip	top:s:vip top:su:vip top:sub:vip	top:sub_env0:vip “?” matches the occurrence of zero or one character except “:”

Namespaces

A namespace can be optionally specified at the beginning of a pattern using the namespace scope operator ‘::’. A namespace may contain any character except a ‘:’. If namespace is not specified, the object namespace is used. An error is issued if an unknown namespace is specified.

For example, looking for a leaf object named “X” in the “RAL” namespace would be specified as:

```
RAL::%:X
```

Namespace names starting with “VMM” are reserved.

Factory Service

Factory Service is an important concept of verification methodology. It provides an easy way to replace any kind of object, transaction, scenario and transactor by a similar object. This replacement can take place from anywhere in the verification environment.

Typical situations are for OO extensions:

- Replace a class by a derived class
- Replace a parameterized class by a derived class
- Replace a transaction modeled using `vmm_data` by a derived class
- Replace a scenario extending `vmm_scenario` by another scenario
- Replace a transactor modeled using `vmm_xactor` or `vmm_unit` by a derived transactor

Similarly, it is possible to use factory to replace similar objects between classes:

- Switch configurations between transactors
- Switch scenarios between generators

Factory service acts as a replacement for object construction. Rather than declaring an object and constructing it using its new() method, VMM provides facilities to consider objects as factory.

Factory service use model is as follows:

- Implement object as a class and use `vmm_class_factory macro for having this object becoming factory enabled
- Create object instance by using a method `class::create_instance()`, this object instance in turn becomes a factory, e.g an object that can be *replaced*
- Replace this factory by using another set of static methods for either copying or allocating a new object using `class::override_with_copy()` or `class::override_with_new()`

Factory service is very handy for modeling generators. Usually generators declare a transaction and then randomize it by applying its built-in constraints. When other set of constraints should be applied to this transaction, this transaction can be replaced by a new transaction that derives the latter one. These steps can be easily carried out by declaring the generator transaction with `class::create_instance()` and replacing it in your test with `class::override_with_new()` method. Of course, special care needs to be taken regarding phases where factory should be created and replaced. Creation should take place in `configure_dut_ph()`phase and replacement should normally be recorded in a preliminary phase like `start_of_sim()`. The following sections provide more details on how to model, add and replace factories.

Modeling a Transaction to be Factory Enabled

This section explains how to model a transaction so that it can be considered as a factory, either in transactor or in the verification environment.

This requires that the class implements a general-purpose constructor, `allocate`() and `copy`() methods.

Note that these methods are automatically implemented with `vmm_data` extension when using short-hand macros.

Since any class factory is mostly based on user-friendly macros, replacing it by an extended class requires the following guidelines:

- Provide a general-purpose constructor. All constructor arguments must have a default so that calls to `new`() are allowed. If some specific members need to be initialized to user specific values, `set*/get*` methods can be used to handle these assignments. Another approach is to use advanced options as described in following section.
- Create a new object by using the `allocate`() method. In this case, the extended class provides the necessary implementation to allocate data members, subsequent objects, etc.
- Create a new object by using the `copy`() method. In this case, the extended class provides the necessary implementation to copy data members, subsequent objects, etc.

The following example shows how to model a simple transaction that extends `vmm_data`:

```
class ahb_trans extends vmm_data;

   rand bit [31:0] addr;

   rand bit [7:0] data;

   `vmm_typename(ahb_trans)

   `vmm_data_member_begin(ahb_trans)

      `vmm_data_member_scalar(addr, DO_ALL)
```

```
    `vmm_data_member_scalar(data, DO_ALL)

  `vmm_data_member_end(ahb_trans)

  `vmm_class_factory(ahb_trans)

endclass
```

Note that `vmm_typename() creates the get_typename() function that returns a string like “ahb_trans”. This is very convenient for displaying this object type. Short-hand macros are used to model data members and `vmm_class_factory declares all necessary methods required to turn this transaction into a factory. Short-hand macros vmm_data_member_* automatically implement allocate() and copy() methods.

The following example shows how to model a simple transaction that extends vmm_object:

```
class ahb_trans extends vmm_object;

  `vmm_typename(ahb_trans)

  rand bit [31:0] addr;

  rand bit [7:0] data;


  function new(string name = "",

               vmm_object parent = null);

    super.new(parent, name);

  endfunction


  virtual function ahb_trans allocate();
```

```
allocate = new(this.get_object_name(),
                get_parent_object());
  endfunction


  virtual function ahb_trans copy();
    ahb_trans tr = new(this.get_object_name(),
                       get_parent_object());
    tr.addr = this.addr;
    return tr;
  endfunction
endclass
```

In case you need to add extra content to this transaction, like new {members, constraints, methods}, you just extend its base class. For instance, the following example shows how to model a simple transaction that extends `ahb_trans`. The only required step is to add a `` `vmm_class_factory `` statement at the end of this transaction to make the class factory ready:

```
class my_ahb_trans extends ahb_trans;
  `vmm_typename(my_ahb_trans)
  …
  `vmm_class_factory(my_ahb_trans)
endclass
```

Adding a Factory to a Transactor

The previous section explains how to model a transaction so that it can be considered as a factory. Let's see now how to instantiate this object in a transactor that extends `vmm_unit`.

Usually, objects are declared in transactors and constructed in its `new`() task. However, when dealing with a factory, this modeling style doesn't apply anymore. A factory needs to be explicitly created in either the `config_dut_ph()` phase or the `build()` phase using the `create_instance()` method. In case there are serial tests to override factories, a common usage is to add transactor factory creation in a test() timeline like `config_dut_ph().`

Note that the `create_instance()` method is static and needs to be prefixed with its class name.

The following example shows how to add a factory to a transactor:

```
class ahb_gen extends vmm_unit;

    `vmm_typename(ahb_gen)

    ahb_trans tr;


    virtual function void config_dut_ph();

        tr = ahb_trans::create_instance(this, "Ahb_Tr0");

    endfunction

endclass
```

Replacing a Factory

As the factory is now available in the transactor, it can be used as is, or replaced in the test, either by copying it from another transaction or by constructing it from scratch. Both use models are made possible using `override_with_copy()` or `override_with_new()` functions.

The following example shows how to add two transactors to a program block and use the default factory, that is to say `ahb_trans`:

```
class env extends vmm_unit;

    ahb_gen gen0, gen1;


    virtual task build_ph();

        gen0 = new("gen0");

        gen1 = new("gen1");

        vmm_log log = new("prgm", "prgm");

      // Displays 0

        `vmm_note(log, {“gen0.tr.addr=”, gen0.tr.addr);

    endtask

endclass
```

To replace a factory by another instance of the same type with different data member values, `override_with_copy()` can be used with a regular expression that matches a specific pattern, either in the vmm_object hierarchy or by referring to the transactor structure.

The following example shows how to replace a specific test that replaces initial factory with a copy:

```
class test1 extends vmm_test;

    virtual function void start_of_sim_ph();

        ahb_trans tr = new("gen0_trans");

        tr.addr = 5;

        ahb_trans::override_with_copy("@%*", tr, log);

    endfunction

endclass: test1
```

To replace a factory by a derived class, `override_with_new()` can be used with a regular expression that matches a specific pattern, either in the vmm_object hierarchy or by referring to the transactor structure. In this case, new transaction type to be considered for factory replacement is returned by `my_ahb_trans::this_type.` This transaction type is usually a derived class as `create_instance()` method considers its underlying base class by default, so there is no point to use a statement like: ahb_trans::override_with_new("@%*",ahb_trans::this_type, log).

The following example shows how to replace the initial factory with a derived object:

```
class test2 extends vmm_test;
```

```
    virtual function void start_of_sim_ph();
        ahb_trans::override_with_new("@%*",
                        my_ahb_trans::this_type, log);
    endfunction
endclass: test2
```

Note that factory replacement takes place in `test2::start_of_sim_ph()` phase, this is necessary as this phase should always be called before `ahb_gen::config_dut_ph(),` otherwise subsequent calls to `override_with_new()` are not considered.

Factory for Parameterized Classes

Factories are general-purpose and apply to any kind of object. Modeling transactions can be achieved either by extending `vmm_data`, `vmm_object` or no object at all.

The following examples show how to write a factory on top of a parameterized class:

```
class A #(type I=int);
  I i;
  `vmm_class_factory(A#(I))
endclass

class AB#(type I=int, J=int) extends A#(I);
  J j;
```

```
   `vmm_class_factory(AB#(I,J))
endclass

class typed_gen #(type T=int) extends vmm_unit;
   `vmm_typename(typed_gen)
   A #(T) a;
   virtual function void configure_dut_ph();
      a = A#(T)::create_instance(this, "A0");
   endfunction
endclass

typedef A#(string) A_str;
typedef AB#(string,int) AB_str_i;

class test extends vmm_test;
   virtual function void start_of_sim_ph();
      A_str::override_with_new("@%*",
                               AB_str_i::this_type, parent);
   endfunction
endclass
```

Factory for Multiple Transactors and Atomic Scenario Generators

Atomic generators are aimed at randomizing unrelated transactions and posting them to a `vmm_channel`. By default, `atomic_gen` comes with a transaction blueprint called `randomized_obj` that can be replaced by a factory. This factory can have the same type as `randomized_obj` or be an extension of it.

Let's consider an example where the `atomic_gen` is wrapped in a `vmm_unit`. This is necessary to ensure its run flow is properly controlled:

```
class ahb_env extends vmm_unit;

    `vmm_typename(env)

    ahb_trans_atomic_gen gen;


    function new(string name);

        super.new(get_typename(), name);

    endfunction


    virtual function void build_ph();

        gen = new("gen", 0);

    endfunction


    virtual task start_ph();
```

```
        gen.start_xactor();
    endtask
endclass
```

In the test, it's now possible to directly replace `atomic_gen::randomized_obj` by a factory using `override_with_new` for a given generator. This is made possible by simply passing the generator's implicit name to this method.

Similarly, a copy of `atomic_gen::randomized_obj` can be overridden in the other generator by also passing its implicit name to the `override_with_copy()` method.

The following example shows how to replace an `atomic_gen::randomized_obj` factory in specific generator by its name:

```
class test extends vmm_test;
   ahb_trans mtr;
   virtual function void start_of_sim_ph();
   // replace factory in env0.gen
   ahb_trans::override_with_new(
                            "@env0:gen:randomized_obj",
                            my_ahb_trans::this_type, log,
                             `__FILE__, `__LINE__);
   // copy factory in env1.gen
   mtr = new; mtr.addr = 'h55;
```

```
   ahb_trans::override_with_copy(
                              "@env1:gen:randomized_obj",
                              mtr, log,
                              `__FILE__, `__LINE__);
   endfunction
endclass: test
```

Factory for Scenario Generators

Scenario generators are aimed at randomizing lists of related transactions and posting them to a `vmm_channel`. By default, `scenario_gen` comes with a user-defined transaction list blueprint that can be replaced by a factory. This factory can be of the same type as the user-defined transaction list or an extension of it.

As described in preceding sections, scenarios are similar to any kind of transaction and need to implement general-purpose `new`(), `allocate`() and `copy`() methods. These methods are directly called by the `override_with_copy()` and `override_with_new()` methods. The only required step is to add a `vmm_class_factory` statement at the end of the scenario to make this class factory ready.

The following example shows how to model a scenario:

```
class test_scenario extends my_scenario;
    int TST_KIND;
    constraint cst_test {
        scenario_kind == TST_KIND;
```

```
        foreach (items[i]) {
            items[i].data == 'ha5a5a5a5;
        }
    }
    function new();
        TST_KIND = define_scenario("tst_scenario", 3);
    endfunction

    function vmm_data allocate();
        test_scenario scn = new;
        allocate = scn;
    endfunction

    function vmm_data copy(vmm_data to = null);
        test_scenario scn = new;
        scn.TST_KIND = this.TST_KIND;
        scn.stream_id = this.stream_id;
        scn.scenario_id = this.scenario_id;
        copy = scn;
    endfunction
    `vmm_class_factory(test_scenario)
endclass
```

Similarly to other transactors, `scenario_gen` is wrapped in a `vmm_unit`. This is necessary to ensure its run flow is properly controlled and that the factory is properly created using a 2-phase approach.

The following example shows how to wrap a `scenario_gen` into a controllable `vmm_unit,` where "`gen.my_scenario`" is the factory:

```
class env extends vmm_unit;

    `vmm_typename(env)

   ahb_trans_scenario_gen gen;

   my_scenario scn;


    function new(string name);

      super.new(get_typename(), name);

    endfunction


    virtual function void build_ph();

        gen = new("gen", 0);

        scn = new();

        gen.register_scenario("my_scenario", scn);

    endfunction
```

```
    virtual task start_ph();
        gen.start_xactor();
    endtask
endclass
```

In the test, it's now possible to directly replace "`gen:my_scenario`" by a factory using `override_with_new`. This is made possible by simply passing the generator's implicit name to this method.

Similarly, a copy of `my_scenario` can be overridden in the scenario generator by passing its implicit name to the `override_with_copy()` method.

The following example shows how to replace `vmm_scenario_gen` factory by its name:

```
class test extends vmm_test;
   my_scenario other_scn;

   virtual function void start_of_sim_ph();
// replace factory in env0.gen with new scenario
     my_scenario::override_with_new(
                          "@env0:gen:my_scenario",
                          test_scenario::this_type,
                          log, `__FILE__, `__LINE__);
```

```
      // copy factory in env1.gen
      other_scn = new;
      my_scenario::override_with_copy(
                              "@env0:gen:my_scenario",
                              other_scn, log,
                              `__FILE__, `__LINE__);
   endfunction
endclass: test
```

Modifying a Testbench Structure Using a Factory

Because the test timeline executes after the pre-test timeline, a test cannot use the `override_with_new`() or `override_with_copy`() factory methods to modify the structure of an environment. By the time the test timeline starts to execute, the environment will already have been built during its “build” phase and all of the testbench component instances will already have been created, hence subsequent calls to `override_with_new`() or `override_with_copy`() won’t be considered. A test can only use the factory replacement methods to affect the instances dynamically created by generators. A test should use `vmm_xactor_callbacks` to affect the behavior of testbench components, not factories. This is not a limitation caused by the implementation, but requirements for test concatenation. When concatenating multiple tests, a test must be able to restore the environment to its original state. It is simple to

do by removing callback extensions, but would have been impossible to do if the environment has been constructed with a test-specific instance.

However, to simplify the use model when not using test concatenation, the `vmm_test::set_config()` method is executed before the phasing of the pre-test timeline. It is thus possible for a test to set factory instances by using the `override_with_new`() or `override_with_copy`() factory methods. But it would not be possible to concatenate such a test with other tests as its modification of the environment would interfere with the configuration of other tests. This method is invoked only if there is only one test selected for execution.

Here is an example showing how to use `vmm_test::set_config()`

```
class mytest extends vmm_test;

   `vmm_typename(mytest)


   ahb_gen gen0 = new("gen0");

   ahb_gen gen1 = new("gen1");

   ahb_trans tr;

   vmm_log log = new("prgm", "prgm");


   function new(string name);

      super.new(name);

   endfunction
```

```
  function void set_config();
    ahb_trans::override_with_new(
                    "@%*", my_ahb_trans::this_type,
                    log, `__FILE__, `__LINE__);
  endfunction
endclass
```

Parameterized Atomic and Scenario Generators

Parameterized atomic and scenario generators come with built-in class factory. It also helps to interactively debug the generator definition that is not hidden inside a macro.

Main classes created for this purpose are:

```
class vmm_atomic_gen #(type T)
class vmm_scenario_gen #(type T)
class vmm_ss_scenario #(type T)
```

These are generic classes with parameterized transaction type. `vmm_atomic_gen/`vmm_scenario_gen macros are redefined with these parameterized atomic/scenario generators and scenarios using typedef as shown below. This makes sure that the existing atomic/scenario usage can be continued without making any changes by the user.

```
typedef vmm_atomic_gen#(T) T_atomic_gen;
```

```
typedef vmm_scenario_gen#(T) T_scenario_gen;

typedef vmm_ss_scenario#(T) T_scenario;
```

Using Parameterized Atomic and Scenario Generators

There are two ways to declare `vmm_channel`, `vmm_atomic_gen`, `vmm_scenario_gen` objects, either by macros or with parameterized class.

The first way is to instantiate generators and handles as in VMM1.1. Callbacks and scenarios can be used in the same way as well.

```
class ahb_trans extends vmm_data;

    rand bit [31:0] addr;

    rand bit [31:0] data;

endclass


`vmm_channel(ahb_trans)

`vmm_atomic_gen(ahb_trans, “AHB Atomic Gen”)

`vmm_scenario_gen(ahb_trans, “AHB Scenario Gen”)


ahb_trans_channel chan0 = new(“ahb_trans_chan”,“chan0”);

ahb_trans_atomic_gen   gen0  = new(“AhbGen0”, 0, chan0);

ahb_trans_scenario_gen gen1  = new(“AhbGen1”, 0, chan0);
```

```
class user_callbacks0 extends
                      ahb_trans_atomic_gen_callbacks;
endclass


class user_callbacks1 extends
                      ahb_trans_scenario_gen_callbacks;
endclass


class user_scenario extends ahb_trans_scenario;
endclass
```

The second way is to directly instantiate the parameterized generator and channels. In case of scenario, you have to extend parameterized `vmm_ss_scenario` class.

```
vmm_channel_typed#(ahb_trans) chan0 =
                      new(“ahb_trans_chan”, “chan0”);
vmm_atomic_gen #(ahb_trans)   gen0  =
                      new(“AhbGen0”, 0, chan0);
vmm_scenario_gen #(ahb_trans) gen1  =
                      new(“AhbGen1”, 0, chan0);


class user_callbacks0 extends
                  vmm_atomic_gen_callbacks#(ahb_trans);
```

```
endclass


class user_callbacks1 extends
              vmm_scenario_gen_callbacks#(ahb_trans);
endclass


class user_scenario extends
                           vmm_ss_scenario#(ahb_trans);
endclass
```

Notify Observer

The VMM Notify Observer simplifies subscription to a notify callback class. It is a parameterized extension of `vmm_notify_callbacks`. Any subscriber (like scoreboard, coverage model, etc) can get the transaction status whenever notification event is indicated. Call the `` `vmm_notify_observer `` macro, specifing the observer and its method name to be called.

```
class vmm_notify_observer #(type T, type D = vmm_data)
       extends vmm_notification_callbacks
```

Using Notify Observer

Consider a subscriber such as a scoreboard having a method *observe_trans*(). Define `vmm_notify_observer` macro specifying subscriber name (scoreboard) and the method name(observe_trans).

```
class scoreboard;

    virtual function void observe_trans(ahb_trans tr);

            ……

    endfunction

endclass

`vmm_notify_observer(scoreboard, observe_trans)
```

Instantiate parameterized `vmm_notify_observer` passing subscriber handle, `vmm_notify` handle and the notification id.

```
scoreboard        sb = new();

vmm_notify_observer#(scoreboard, ahb_trans)

     observe_start =new(sb, mon.notify,mon.TRANS_START);
```

Whenever the notification event is indicated, subscriber method (`observe_trans`()) is called.

Connect Utility (vmm_connect)

VMM connect utility class `vmm_connect` can be used for connecting channels and notifications in the `vmm_unit::connect_ph()` method. It does additional check on whether the channels are already connected to a producer and to a consumer. This connection is usually done with `vmm_channel set_consumer()` and `set_producer()` methods.

```
class vmm_connect #(type T, type D = vmm_data)
```

`vmm_connect` has the following methods that can be used for channel/notification connectivity.

```
class vmm_connect#(T)::channel(ref T upstream,
                               downstream,
                               string name= “”,
                               vmm_object parent = null);
```

Using Connect Utility

vmm_connect#(T)::channel() method is used to connect the channels as follows.

```
class ahb_unit extends vmm_unit;
   ahb_trans_channel gen_chan;
   ahb_trans_channel drv_chan;


   virtual function void build_ph();
```

```
    drv_chan = new(“ahb_chan”, “drv_chan”);
    gen_chan = new(“ahb_chan”, “gen_chan”);
  endfunction

  virtual function void connect_ph();
    vmm_connect#(ahb_trans_channel)::connect(gen_chan,
                          drv_chan,  “gen2drv”,  this);
  endfunction

endclass
```

It is an error to attempt to connect two channels that are already connected together or to another channel.

`vmm_connect#(T, D)::notify()` method is used to connect notification to the subscriber (scoreboard) as follows.

```
  class scoreboard;
  virtual function void observe_trans(ahb_trans tr);
  ……
  endfunction
  endclass
  `vmm_notify_observer(scoreboard, observe_trans)
```

```
class ahb_unit extends vmm_unit;
   scoreboard sb;

virtual function void build_ph();
   sb = new();
endfunction

virtual function void connect_ph();
   vmm_connect#(scoreboard, ahb_trans)::notify(sb,
                    mon.notify, mon.TRANS_STARTED);
endfunction
endclass
```

RAL updates

Scoreboard updates