Reference Verification Methodology User Guide

Version C-2009.06
June 2009

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

SYNOPSYS®

Copyright Notice and Proprietary Information

Right to Copy Documentation

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 (®)

Trademarks (™)

Service Marks (SM)

Contents

1 Coding and Compilation

2 Testbench Architecture

3 Common Message Service

4 Data and Transaction Models

5 Stimulus and Generation

6 Transactors

7 Design for Verification (DFV)

8 Verification Environment

9 Testcases

A Class Reference

10 OVA Checker Library Quick Reference

1

Coding and Compilation

RVM Declarations and Implementations

The RVM base class library is shipped with the Vera, and VCS and the Pioneer-NTB distribution sets. The header file can be found in the following locations:

```
$VERA_HOME/include/rvm_std_lib.vrh
$VCS_HOME/include/rvm_std_lib.vrh

$PIONEER_HOME/include/rvm_std_lib.vrh
```

To make the RVM declarations visible in your OpenVera source file, use the following statement:

```
#include <rvm_std_lib.vrh>
```

In Vera, the RVM base class library is automatically included. No additional compile-time or runtime options are required.

In VCS, the `-ntb_opts rvm` compile-time option is required in order to include the RVM base class library. For example:

```
vcs -ntb -ntb_opts rvm filename.vr
```

No additional runtime option is required. In Pioneer-NTB, the `-ntb_opts rvm` compile-time option is required in order to include the RVM base class library. No addtional runtime option is required. For example:

```
pioneer -ntb -ntb_opts rvm filename.vr
```

Directives

Throughout this user guide you will find directives that will help you to construct reusable, configurable testbenches. Sometimes the directives are merely guidelines that describe good practice, but in other cases the directives contain keywords that must be obeyed in order to achieve the intended results. The meaning of the keywords in the directives is explained in the table below.

Directive keywords	**Meaning**
Shall	The word "shall" dictates an absolute requirement. Any sentence containing the word 'shall' represents a firm requirement.
Should	The word "should" dictates a highly desirable characteristic. Any sentence containing the word "should" represents a design goal.

Directive keywords	Meaning
May	The words “may” dictates a desirable characteristic. Characteristics identified by “may” can be implemented at the discretion of the design team.
Will	The word “will” dictates a statement or declaration of intent. Any sentence containing the word “will” represents a statement of fact or a declaration of intent.

Although these directives are specific in their implementation, it is their objective that is important. In several cases, the stated directive implementation can be freely modified to match an pre-existing directive with similar objectives. The customizable aspects of a directive are shown in plain characters while the mandatory aspects are shown in **bold** characters. All examples and samples will follow the stated implementation.

The layout of these guidelines or directives is shown in the examples below:

All symbols representing run-time constants shall be in uppercase.

> This guideline dictates that all such symbols must follow this firm requirement. However, the actual style used to represent such symbols can be customized.

Coding Guidelines

The following sections outlines general coding guidelines for implementing testbenches using the RVM.

Unless superseded by a guideline in this document, use the coding guidelines specified in the *Reuse Methodology Manual* (3rd edition).

Most companies already have a set of coding guidelines applicable to various programming language, application of a language or methodology. These same guidelines should be used unless they conflict with a methodology requirement.

OpenVera-Specific Guidelines

This section outlines coding guidelines specific to the OpenVera language.

All global identifiers shall have prefixes to ensure uniqueness.

If two global identifiers have the same name, a run-time error will occur. It is necessary to ensure that global identifiers are unique to prevent collisions when code authored at different times and places can be integrated without modifications. Global identifiers in OpenVera include:

- Class names
- Global tasks and functions
- Preprocessor macros

The specific prefix to be used will depend on the nature and scope of the global identifier. Specific prefixes will be identified in subsequent guidelines. The following table identifies reserved prefixes.

Reserved Prefix	Usage
vmt	Synopsys Inc, Designware verification IP.

Reserved Prefix	Usage
rvm	Synopsys Inc, Methodology base classes.
vera	Synopsys Inc, language extensions.
snps	Synopsys Inc.

All global symbols in code authored by this business unit shall be prefixed with "bu_".

To ensure that no collision will occur between code written in different business units, each business unit shall use a different prefix for the global identifiers they create.

Blocking methods shall have a name that ends with the "_t" suffix.

In OpenVera, there is no way to force a method to be nonblocking (that is, not suspend the execution thread). When writing reusable transactors with user-extendable callback methods, it is necessary to be able to enforce that some methods be nonblocking to prevent a user form corrupting the transactor implementation or protocol.

This guideline makes explicit which methods are allowed to be blocking, and those that cannot be.

Note that a method that *may* be blocking may not necessarily be so in all cases. It is possible for a method to block the execution thread only some of the time.

Methods that do not have a name that ends with the "_t" suffix shall be nonblocking.

When calling a method without the blocking naming pattern, one must be able to rely on the execution thread to return from the method without being blocked.

Note that this guideline implies that constructors must be nonblocking methods.

Constructors shall be nonblocking.

It is confusing to have the execution thread block in the middle of a sequence of variable declaration because a constructor had a blocking implementation. A thread should only be blocked in sequential code.

If a blocking thread must be executed in a constructor, either `fork` it or move it to a `start` method.

This guideline is compatible with SystemVerilog 3.1.

Functions should be nonblocking.

Functions cannot be blocking in Verilog, SystemVerilog or VHDL. To simplify an eventual porting to an alternative implementation or interface into a different language, a similar semantic should be followed.

If a blocking method needs to return a value, use a `var` argument on a task.

This guideline is compatible with SystemVerilog 3.1.

Global functions and tasks shall not be used.

Global functions and tasks must be implemented in a *program* file and thus create management challenges when they must be shared across multiple programs. They consume global symbol space. Furthermore, their usage cannot be differentiated from calling pre-defined procedures and may conflict with pre-defined procedures that will be introduced in future version of the language.

Functions and tasks shall be implemented as methods.

This is a corollary to the previous guideline. Related functions and tasks shall be packaged into a class. To invoke a function or task, a user simply creates an instance of the class.

For example, a package of functions and tasks would be coded in a file named "package.vr":

```
class a_package {
   task a_task(...) {
      ...
   }
}
```

A user would call a task or function in the package using:

```
#include "package.vrh"
...
{
   a_package a_pkg = new;
   a_pkg.a_task(...);
}
```

Files and Directories

This section outlines general guidelines for naming files and structuring them across directories. They also specify the compilation mechanism used to compile those files into a simulatable object.

All OpenVera source file shall have the ".vr" type.

This file type is already recognized by tools as containing OpenVera source code.

All OpenVera aspect-oriented extension source file shall have the ".vra" type. (Not currently supported in VCS and Pioneer-NTB)

Current versions of Vera requires that AOP extensions be located in separate files from regular, non-AOP code. They must also be loaded using a different command-line option. It is therefore a good idea to name these files differently.

All manually written OpenVera header file shall have the ".vri" type.

This file type is already recognized by tools as containing OpenVera header declarations. The `.vrh` file type is reserved for automatically generated header files.

All automatically generated OpenVera header file shall have the ".vrh" type. (Not currently supported in VCS and Pioneer-NTB)

This file type is already recognized by tools as containing OpenVera header declarations and is automatically appended by the Vera compiler when creating a header file.

Header files corresponding to source files should be automatically generated. (Not currently supported in VCS and Pioneer-NTB)

Automatically generating header file ensures that they are up-to-date with respect to the implementation found in the source code.

Automatically-generated header files should be generated using the -H option. (Not currently supported in VCS and Pioneer-NTB)

Unlike `-H`, the `-h` option generates header files that does not include the `#include` directives required to make declarations of symbols used in the generated file visible. This requires users of the header file to know which other files must be included before including a header file.

Verification IP

This section outlines file, directory and compilation guidelines to be used when implementing a verification IP package for a specific protocol. Each verification IP package would be a different instance of these guidelines.

In this section, the string "xyz" is used to represent the protocol name.

All global identifiers shall be prefixed with "bu_xyz_".

The prefix must be unique to the company authoring the verification IP and the protocol implemented by the verification IP.

A verification IP package shall be authored using the following directory structure.

This directory structure is used for authoring purposes only. A different structure is used for distribution or installation.

Table 1-1 Authoring Directory Structure

Directory	Meaning
xyz/	Base directory for protocol
xyz/bin/	Scripts

Table 1-1 Authoring Directory Structure

Directory	Meaning
xyz/doc/	User documentation
xyz/src/	Source code
xyz/include/	Header files
xyz/vro/	Object files
xyz/vro/random_compat/	Backward-compatible object files
xyz/vrps/	VIP files
xyz/templates/	User-extension templates
xyz/tests/	QA tests
xyz/examples/	Usage examples
xyz/examples/abc/	"ABC" usage example

A verification IP package shall be distributed using a single header file and a single object file.

This will facilitate usage of the verification IP by necessitating the inclusion of only one header file and the loading of only one object file.

A top-level source file shall include all ancillary source files.

Files are included in compilation order. This will facilitate the creation of a single header file and a single object file for the entire verification IP package as only the top-level file needs to be compiled. This allows the implementation of the verification IP to be scattered across an arbitrary number of files with an arbitrary naming convention, without affecting the user.

Note that, with the current version of Vera, the inclusion of ancillary source files makes it impossible to generate the header file using the `-H` option: the `include` directives for the source files themselves are put in the header file instead of the external declarations for the types and classes found in the included source files. It is therefore necessary to use the `-h` option to generate the header file. Because the `-h` option does not include the necessary `include` and `extern` directives in the generated header file, it will be necessary to fix the generated file afterward:

Top-level source file "xyz.vr":

```
#include "incl_extrn.vri"

#include "xyz_packet.vr"
#include "yxz_bfm.vr"
#include "ygz_generator.vr"
```

Header file generation and post-processing:

```
xyz.vrh xyz.vro: xyz.vr
      vera -cmp -h xyz.vr
      head -5 xyz.vrh     >tmp
      cat incl_extrn.vri >>tmp
      tail +5 xyz.vrh    >>tmp
      mv tmp xyz.vrh
```

No source file, other than the top-level file, shall use the #include directive.

This is a requirement for generating a valid, self-contained header file as described in the previous guideline.

A verification IP package shall be distributed using the following directory structure.

The name of the top-level directory should contain the version of number of the VIP distribution to allow multiple distributions of different versions to coexist in a client installation. This directory structure is used for distribution purposes only. A different structure is used for authoring or installation.

Table 1-2 Distribution Directory Structure

Directory	Meaning
xyz-X.Y.Z/	Base directory
xyz-X.Y.Z/version.txt	Text file with version number
xyz-X.Y.Z/bin/	Scripts
xyz-X.Y.Z/doc/	User documentation
xyz-X.Y.Z/xyz.vrh	Header file
xyz-X.Y.Z/xyz.vro	Object file
xyz-X.Y.Z/xyz.compat.vro	Backward-compatible object file
xyz-X.Y.Z/xyz.vrps	VIP file (optional)
xyz-X.Y.Z/templates/	User-extension templates
xyz-X.Y.Z/examples/	Usage examples
xyz-X.Y.Z/examples/abc/	"ABC" usage example

Verification Project

This section outlines file, directory and compilation guidelines to be used when implementing a verification project for a set of blocks and an overall system.

A verification project shall use the following directory structure.

The directory structure is specified for the verification activity only. It can be augmented with design and synthesis-related directories.

Table 1-3 Project Directory Structure

Directory	Meaning
verif/	Base directory
verif/bin/	Scripts
verif/doc/	Verification documentation
verif/vips/	Verification IP used by project
verif/vips/xyz-X.Y.Z/	Distribution for xyz VIP
verif/vips/xyz-latest@	Soft-link to latest release of xyz VIP
verif/vips/include/	Incude files for all VIPs
verif/vips/include/xyz.vrh@	Soft-link to latest include file for xyz VIP
verif/vips/vros/	Object files for all VIPS
verif/vips/vros/xyz.vro@	Soft-link to latest object file for xyz VIP
verif/blk/	Verification for block "*blk*"
verif/blk/env	Verification environment. There may be more than one environment, each in separate directories.
verif/blk/env/blk_xyz.vr	Environment-specific extensions of xyz VIP
verif/blk/env/coverage.vr	Environment-specific coverage model
verif/blk/env/env.list	List of source files implementing the environment
verif/blk/env/env.vr	Environment implementation
verif/blk/env/env.vrl	List of object files required by environment (can be automatically generated from *env.list*)

Table 1-3 Project Directory Structure

Directory	Meaning
verif/blk/env/env_shell.vr	Dummy program to generate shell file
verif/blk/env/env_shell.v	Shell file for environment
verif/blk/env/self_check.vr	Self-checking structure for environment
verif/blk/tests/	Testcases. If more than one environment exists, each directory of testcases for a particular environment would be named "*env-tests*" where "env" is the name of the environment directory.
verif/blk/tests/abc/	Family of tests "*ABC*"
verif/blk/tests/abc/test.vr	Testcase implementation file (one or two per testcases, many testcases per family).
verif/blk/tests/abc/logs/	Simulation output log files
verif/blk/tests/abc/cvr	Functional coverage databases
verif/sys/	Verification for system "*sys"*. Duplicates the structure found under "*verif/blk*".

All source files implementing an environment should be compiled using the -dep_check and -F options.

This option lets Vera deduce the compilation dependencies and eliminates the need for manually maintaining dependencies in a Makefile.

The following Makefile rule will compile all the source files used to implement an environment.

Vera:

```
env:
      vera -cmp -HC -I../../vips/include -dep_check -F \
      env.list
```

VCS:

```
env:
      vcs -ntb -ntb_opts rvm -ntb_incdir \
     ../../vips/ include -ntb_opts dep_check -f env.list
```

Pioneer-NTB:

```
env:
      pioneer -ntb -ntb_opts rvm -ntb_incdir \
     ../../vips/ include -ntb_opts dep_check -f env.list
```

A file named env.list should contain the name of all source files used to implement the environment.

This is a requirement of the preceeding guideline. Pathnames should be relative to the location of the list file. A hiearchical system of list files may be used. This file may be used to generate the `env.list` file.

For example, if an environment is implemented using the following structure:

```
env/
    env.vr
    self_check.vr
    scoreboard/
               ordering.vr
               transform.vr
    coverage/
             input_cvr.vr
             buffer_cvr.vr
             ordering.vr
```

The following list files would be used:

In file "`env/env.list`":

```
env.vr
```

```
self_check.vr
scoreboard/ordering.vr
scoreboard/transform.vr
-F coverage/coverage.list
```

In file "`env/coverage/coverage.list`":

```
input_cvr.vr
buffer_cvr.vr
ordering.vr
```

A testcase shall be implemented in a single file.

This localizes all testcase-specific code into a single file.

Due to a current implementation restriction, tests implementations that involve AOP extensions must be implemented using two files: one for the AOP extensions and one for the OOP extensions. In that case, the name of the AOP extension file should clearly indicate which testcase program file it is intended to be associated with. (Not currently supported in VCS and Pioneer-NTB)

A single AOP object file shall be loaded.

If a testcase or simulation requires more than one AOP extensions located in separate files, an additional file shall be created, including the other files and compiled to create the single AOP object file.

Command-lines cannot be archived or source controlled. A testcase should be embodied in a file that is source controlled and archived. (Not currently supported in VCS and Pioneer-NTB)

A testcase implementation shall be compiled independently from all other testcases.

The "compile all files" strategy used for verification IP and the environment is fine for these components because they are always used as a complete system. Testcases are independent of each other and therefore need not all be compiled to run an invidual testcase.

The following Makefile rules will compile all the source files used to implement a testcase (not currently supported in VCS and Pioneer-NTB):

```
test_00.vro: test_00.vr
      vera -cmp -vlog -I../../vips/include -I../../env
test_00.vr

test_00_ao.vro: test_00_ao.vr
      vera -cmp -aop test_00_ao.vr
```

The default seed used to simulate a testcase shall be random.

The default random seed in an OpenVera testbench is 1 in a Vera, or VCS or Pioneer simulation. Too many times users consider the job done when the simulation succeeds with the default seed and do not think about trying different seeds. Using a random seed by default will cause multiple variations to be tried without efforts.

The more recent versions of Vera now provide the `+vera_auto_random_seed` command line option to select a random seed. When using older versions, the following C program can be used to generate a random number that can then be passed to the `+vera_random_seed` argument:

```
#include <stdlib.h>
#include <time.h>

main() {
  time_t *t;
  srand(time(t));
  printf("%d\n", rand());
}
```

There shall be a Makefile target to simulate each individual testcase.

This allows the command line used to simulate a testcase to be archived and source controlled. The following Makefile rules will simulate a testcase (not currently supported in VCS and Pioneer-NTB):

```
test_00: test_00.vro test_00_ao.vro
      vcs +vera_random_seed=‘random‘ \
         +vera_aop_vros=test_00_ao.vro
+vera_vros=test_00.vro \
          +vera_mload=env.vrl
```

Examples

RVM examples are located in the following installation directories:

```
$VCS_HOME/doc/examples/nativetestbench/openvera/rvm

$PIONEER_HOME/doc/examples

$VERA_HOME/examples/rvm
```

A subset of RVM examples is discussed in the following sections.

Example 1: bu_layer

This example is found in:

```
$VCS_HOME/doc/examples/nativetestbench/openvera/rvm/
bu_layer
```

```
$PIONEER_HOME/doc/...
```

```
$VERA_HOME/examples/rvm/bu_layer
```

Note:

> When a `filename.vr` appears in blue, there is a link to a PDF version of the example.

This is a template when developing a business-unit-specific set of intermediate base classes to introduce best practices and standard specific to that business unit on top of an above those provided by the RVM base classes. Simply replace the "*bu_*" prefix with a suitable identifier for the business unit.

As only three RVM class are used as base classes, there are only three classes included in this template. All other RVM classes are designed to be instantiated, not derived from.

Top-Level File

The file `bu_std_lib.vr` is the top-level file used to encapsulate the individual files that compose the business-specific verification IP. By encapsulating the individual files into a single file, it simplifies the usage of the verification IP by having only one header file to include and one object file to load. The structure, name and number of the implementation files can be modified without affecting users.

Data and Transaction Model

The file *bu*`_data.vr` is a business-unit-specific extension of the `rvm_data` class. All data and transactions models created by the business unit should be based on this class.

Transactor

The file *bu*`_xactor.vr` is a business-unit-specific extension of the `rvm_xactor` class. All transactor models created by the business unit should be based on this class.

Verification Environment

The file *bu*`_env.vr` is a business-unit-specific extension of the `rvm_env` class. All verification environments created by the business unit should be based on this class.

Example 2: host_if (Not currently distributed with VCS and Pioneer-NTB)

This example is found in:

```
$VERA_HOME/examples/rvm/host_if
```

This is a simple example to verify the memory-mapped registers accessible via a Intel-style management interface. If this was all that was required to verify in a design, a simple directed testcase would have been much simpler to write and use. Using the RVM methodology with such a simple verification task looks like, and is, overkill. However, the objective of this example is to focus on the RVM implementation details, and not on the complexity of the verification task. Therefore, the source files in this example make an excellent template with which to start a new project, because the DUT-specific functionality is trivial and easy to extract.

DUT Source Code

The HDL source code for the DUT can be found in the `hdl` directory.

It implements 32 8-bit registers. The registers at addresses 0 through 7 are read-only, whereas registers at addresses 8 through 31 are read/write.

The active low reset input is asynchronous. Upon reset, all registers are set to 8'h00.

The host interface is an Intel-style asynchronous parallel management interface, as defined in the Utopia Level 2, Version 1.0 specification (ATM Forum Technical Committee, document af-phy-0039.000, June 1995, section A.2.4).

The DUT contains no other functionality or interfaces.

Management Interface

A reusable transactor for the Intel-style management interface is located in the `intel_mgmt` directory.

This directory contains a transaction model, a bus master, a bus monitor, and a transaction scenario generator. The bus master model implements a blocking interface.

These data models, transactors, and generators are not written specifically for the DUT to be verified and can be reused in other projects or in a verification environment where the same physical interface is used.

Verification Environment

The verification environment for the DUT can be found in the `host_if/env` directory.

This directory contains a testcase configuration model, a functional coverage model, a scoreboard, and a constrainable, random, functional, verification environment that integrates the functional coverage model, the scoreboard, and the reusable Intel-style management interface.

The individual tests are located in the `tests` directory.

The file `env.vr` is not a testcase itself. It is only used to create the Vera shell file for the environment. Because the interface to the DUT is constant for all testcases, it is not necessary to generate individual shell files for each test. Once the shell file has been generated, it is possible to built a single simv image that will be reused for all testcases.

Testbench Files

The testcases are found in the files named `test_`*`XX`*`_name.vr`. The two digits *`XX`* are used to document the order in which they are written or run. To run a particular testcase, use the command:

```
make test_XX
```

Test 00 is the trivial testcase. It defines a single scenario with only two cycles. The first cycle is a write and the second a read, both at the same (random) address. This testcase is used to debug the DUT and the environment. The functional coverage databases generated by this test are usually ignored.

Test 01 is the broad-spectrum unconstrained testcase. This testcase should be run multiple times with different seeds, collecting cumulative functional coverage. The cumulative coverage results are then analyzed to identify the testcases that remain uncovered.

Test 02 is a constrained testcase that constrains the data value to an interesting and relevant one, and modifies the address distribution to better match the functional coverage model. Running this testcase for a few different random seeds should complete the coverage model.

Test 03 is an example of a directed testcase. This testcase performs sequential, back-to-back write and read cycles of random data values through the address space.

Test 04 is an example of error injection. This particular testcase injects errors every few write cycles. These errors should be detected and reported by the scoreboard in subsequent read cycles at the same addresses.

Example 3: atm

This example is found in:

```
$VCS_HOME/doc/examples/nativetestbench/openvera/rvm/atm
```

```
$PIONEER_HOME/doc/examples...
```

```
$VERA_HOME/examples/rvm/atm
```

This is a simple example to use as a template when developing data models and generators.

Top-Level File

The file atm.vr is the top-level file used to encapsulate the individual files that compose the ATM verification IP. By encapsulating the individual files into a single file, it simplifies the usage of the verification IP by having only one header file to include and one object file to load. The structure, name and number of the implementation files can be modified without affecting users.

Data Model

The file atm_cell.vr contains a data model for a UNI ATM cell.

The file test_data.vr verifies the correctness of the data model implementation.

Atomic Generator

The file atm_atomic_gen.vr contains an atomic ATM cell generator. It can be used as a template to create atomic generators for other data models.

The file test_atomic_gen.vr verifies the correctness of the default implementation of the atomic generator.

The file test_atomic_gen_constrained.vr shows (and tests) how to add constraints to the data stream generated by the atomic generator.

Coverage Model

The file atm_cover.vr shows how to implement a coverage object as part of a coverage model. It samples various elements of an ATM cell supplied via a task and fills the relevant coverage points based on the sampled values.

The sampling interface (the atm_cover::cover() task) is designed to facilitate integration of the coverage object in the verification environment (see test_atomic_gen.vr and test_atomic_gen_constrained.vr). The coverage samples and states are designed to facilitate interpretation of the functional coverage model and its analysis.

Example 4: wishbone

This example is found in:

```
$VCS_HOME/doc/examples/nativetestbench/openvera/rvm/ \
wishbone
```

```
$PIONEER_HOME/doc/examples...
```

```
$VERA_HOME/examples/rvm/wishbone
```

This is a simple example to use as a template when creating transactors for bus protocols.

List of files:

- README
- top.v
- config.vr

- `cycle.vr`
- `slave.vr`
- `test.vr`
- `Makefile`
- `wb_if.vri`

Example 5: log (not currently distributed with VCS or Pioneer-NTB)

This example is found in:

```
$VERA_HOME/examples/rvm/log
```

This is a set of small examples demonstrating the versatility and power of the message service interface. Each example shows how the message service interface can be used to provide a specific benefit. Each technique can be combined to create the required features.

File names and line numbers

The file `rvm_log_fileline.vri` is an include file that redefines the standard message macros to prepend the name of the file and line number where the message was actually issued.

xactor_base_log.vr

The file `xactor_base_log.vr` shows how multiple instances of the message service interface can be used to provide a finer granularity of control over messages issued by a verification component.

2

Testbench Architecture

Introduction

This chapter gives an overview of the RVM testbench architecture. The concepts outlined here are expanded in following chapters and form the basis of a testbench methodology that we refer to as the Reference Verification Methodology (RVM).

- The objective of this methodology is to create a test environment with the following characteristics or features (in decreasing order of concern):
- Use functional coverage metrics to direct the verification effort and measure progress.
- Maximum use of random stimulus.
- Creation and use of reusable verification components.

- Portability across various level of abstraction of the DUT.
- Portability from block-level to system-level.

An important principle underlying the testbench methodology is the minimization of the **overall** code that is required to write to verify a design to a certain level of confidence. Note the emphasis on "*overall*".

There will be cases where a short-term objective could be achieved using a small amount code. However, a longer-term view that ensures code reusability, will usually necessitate a more verbose implementation. For example, if writing an extra 100 lines of code in the verification environment will eliminate the need to write 20 lines of code in 20 different tests, those 100 lines of code are a good investment.

Writing reusable verification components such as generators, bus-functional models, and monitors is a natural consequence of the methodology. To minimize the amount of code that needs to be written, a verification environment must be built from verification components that are highly reusable across testcases, reusable across the verification environment, and reusable across projects.

Coverage-Driven Verification Strategy

Once all the test conditions that need to be verified have been identified, there are different strategies that could be used to create those conditions. The implementation strategy used in this methodology is to do random testing first, and directed testing last. However, the testbench architecture, coding guidelines and base classes are all usable in a purely directed strategy.

Instead of writing directed testcases (or almost-directed testcases that use randomization only to fill in irrelevant details) the strategy is to write a single verification environment that will, on its own, generate the interesting conditions identified earlier. Of course, it will need a certain level of prodding in the form of constraints. But the objective is to avoid writing individual testcases unless absolutely necessary.

Functional coverage identifies which test conditions have been automatically created by the random stimulus. Instead of coding individual interesting conditions in individual directed testcases, they will be coded as individual coverage points in a functional coverage model. Testcases will then be used to steer the environment toward the uncovered points in the coverage model.

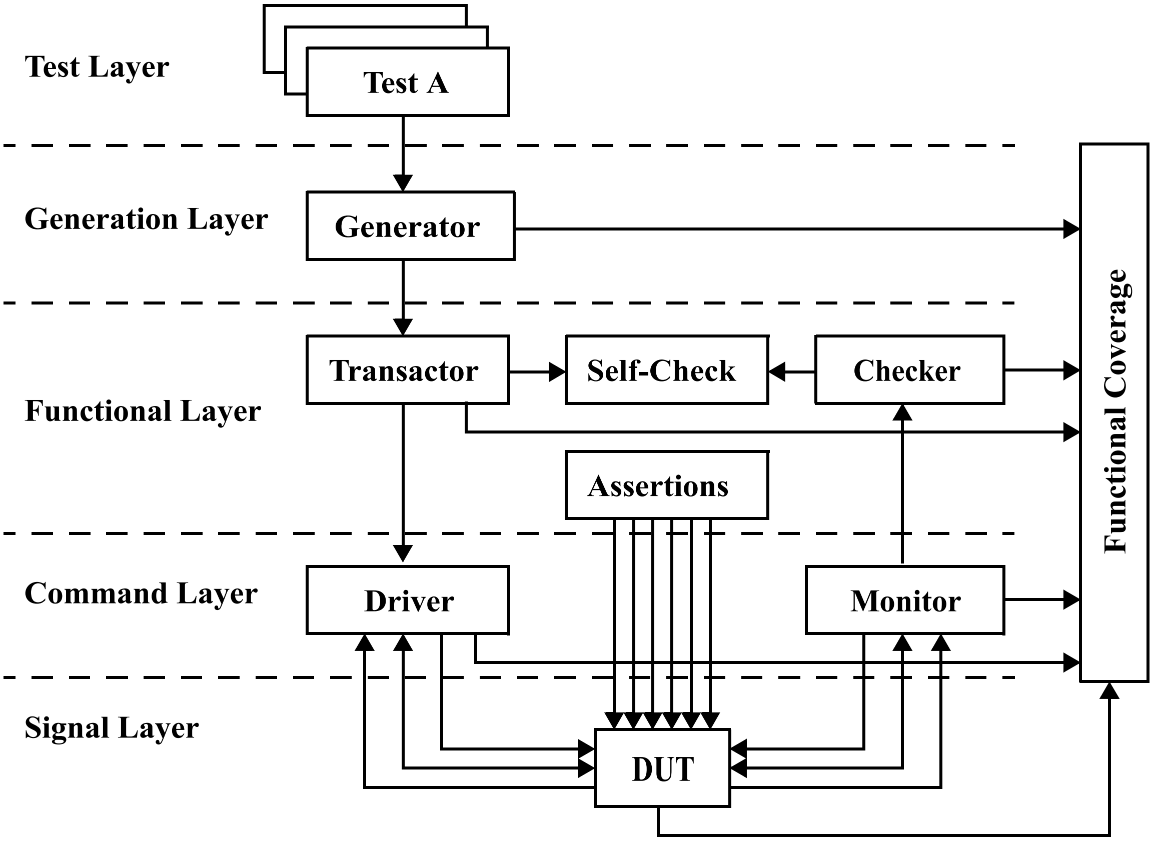
This strategy trades off testcase writing time for simulation run-time, using functional coverage to focus the remaining testcase writing effort.

Layered Model

Testcases are implemented on top of a layered verification environment, as illustrated in Figure 2-1. Each layer provides a set of services to the upper layers, while abstracting it from the lower level details.

The layered architecture makes no assumption about the DUT model. It can be an RTL or gate-level model as well as a transaction-level model. The DUT can also be simulated natively in the same simulator as the verification environment, co-simulated on a different simulator, or emulated on a hardware platform.

Figure 2-1 Layered Verification Environment



Signal Layer

This layer provides signal-level connectivity into the physical representation of the DUT (which can be HDL, SystemC or physical emulation). This layer provides signal name abstraction and connectivity to the event driven world of most simulation engines. This layer may also abstract synchronization and timing of the signal with respect to a reference signal from the verification environment. For example, if the DUT is implemented in Verilog and the verification environment is in OpenVera, the signal layer would simply consist of the OpenVera interface declaration(s). The OpenVera interface also defines signal synchronization (sampling clock) and timing (setup and hold) time.

The verification environment uses virtual signal names instead of the actual, often simulator-specific, signal names. This allows for changes in the actual signal names or path – and some timing characteristics – in the DUT without affecting the verification environment or the testcases. The signal abstraction provided by this layer should be accessible to all layers and testcases above it. However, verification environments and testcases should be implemented in terms of the high-level services provided by the lower layers and avoid accessing signals directly unless absolutely necessary.

Command Layer

The command layer typically contains bus-functional models, physical-level drivers and monitors associated with the various interfaces and physical-level protocols present in the DUT. It provides a consistent, low-level transaction interface to the DUT, regardless of how the DUT is modeled. At this level, a transaction is defined as an atomic data transfer or command operation on an interface, such as a register write, the transmission of an Ethernet frame, or the fetching of an instruction. Atomic operations are typically defined using individual timing diagrams in interface specifications.

When interfacing with an RTL or gate-level model, the physical abstraction layer may translate transactions to or from signal assertions and transitions. When interfacing with a transaction-level model, the physical abstraction layer becomes a passthru layer. In both cases, the same transaction-level interface presented to the higher layers remains the same, allowing the same verification environment and testcases to run on different models of the DUT, at different levels of abstraction, without any modifications.

The transaction-level interface of this layer may be separated from its implementation. For example, a bus-functional model may be implemented in HDL but controlled via an OpenVera testbench. Similarly, a bus-functional model may be implemented using RTL code and simulated on an emulator. This layer simply provides a mechanism for interfacing atomic transaction-level commands to and from the verification environment to the DUT interface.

Reading and writing registers is an example of an atomic operation. The command layer provides methods to access registers in the DUT. To speed-up device initialization, this layer may have a mechanism that bypasses the physical interface to peek and poke the register values directly into the DUT model. Such choice should be selectable at run-time, where all subsequent register accesses would be done in the same manner until the mode selection is modified. Note that the implementation of a direct-access, register read/write driver is dependent upon the implementation of the DUT.

The services provided by the command layer may not be limited to atomic operations on external interfaces around the DUT. They can be provided on internal interfaces for missing, or temporarily removed, design components. For example, an embedded memory acting as an elastic buffer for routed data packets could be replaced with a testbench component to help track and check packets in and out of the buffer rather than only at DUT endpoints. Or an embedded code memory in a processor could be replaced with a reactive monitor that would allow on-the-fly instruction generation instead of using pre-loaded static code.

This layer includes a functional coverage model for the atomic stimulus and response transactions. It records the relevant information on all transactions processed or created by this layer.

Functional Layer

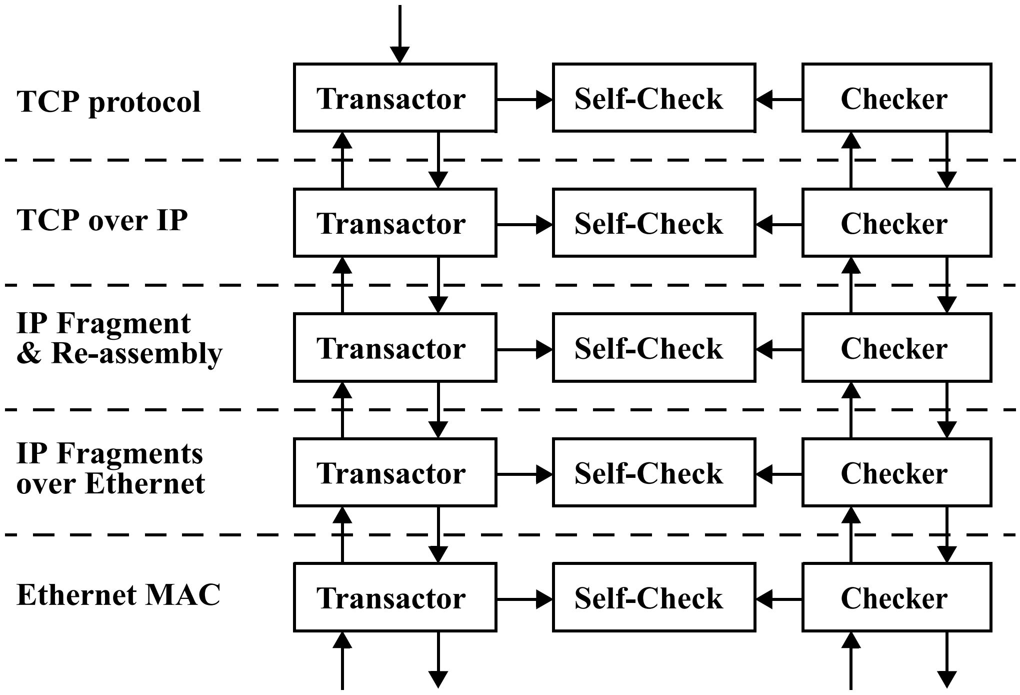
The functional layer provides the necessary abstraction layers to process application-level transactions and verify the correctness of the DUT.

Unlike interface-based transactions of the physical layer, the transactions in the functional layer may not have a one-to-one correspondence with an interface or physical transaction. Functional transactions are abstraction of the higher-level operations performed by a major subset of or the entire DUT, beyond the physical interface module. A single functional transaction may require the execution of dozens of command-layer transactions on different interfaces. It may depend on the completion status of some physical transaction to retry some transactions or delay others.

At all times, the self-checking structure included in this layer verifies the correctness of the response of the DUT, based on the configuration and stimulus streams. The correctness may be determined at various levels of abstraction – physical or functional – according to the functionality being verified. The correctness of the response should not imply or require that a particular model of the DUT is used, nor should it depend on unspecified ordering or timing relationships between the transactions.

This layer may be sub-layered according to the protocol structure. For example, a functional layer for a USB device should contain a sub-layer to translate from a scheduled USB transaction to USB packets. Additional sub-layers may be provided to perform USB transaction scheduling, to translate from scheduled USB transfers to unscheduled USB transactions, from unscheduled USB transfers to scheduled USB transfers and from USB device enumeration command to unscheduled USB transfers.

Figure 2-2 Functional Sub-Layers



A test is performed as a series of functional transactions at the appropriate level of abstraction, rather than always using low-level physical transactions or physical signals.

It must be possible to turn off the higher sub-layers. As tests are implemented, they are first concerned with verifying the lower-level operations of the DUT. These lower level operations correspond to the lowest sub-layers of the functional layer. Stimulus – and self-checking – is performed at the relevant abstraction sub-layer to easily create the relevant scenarios and corner cases for that level of abstraction. This requires that any stimulus provided by the higher sub-layers be turned off to prevent undesirable noise from affecting a testcase. This requirement is often a by-product of the implementation of the verification environment itself: it is typically implemented bottom-up, with the low-level testcases implemented first. As additional levels of functionality are being verified, additional sub-layers are added to the functional layer. To maintain backward compatibility with the existing lower-level testcases, these additional sub-layers must be disabled by default.

The functional layer is also responsible for configuring the DUT according to a configuration descriptor. The configuration descriptor is a high-level description of the DUT configuration that is “compiled” into the necessary register reads and writes and embedded memory images.

This layer includes a functional coverage model for the high-level stimulus and response transactions. It records the relevant information on all transactions processed or created by this layer.

Generation and Scenario Layer

This layer provides controllable and synchronizable data and transaction generators. By default, they generate broad-spectrum stimulus to the DUT. Different generators are used to supply data and transactions at the various sub-layers of the functional layer. This layer also contains a DUT configuration generator.

Atomic generators generate individually constrained transactions. They are suitable for generating stimulus where putting constraints on sequences of transactions is not necessary. For example, the configuration description generator is an atomic generator.

Scenarios are sequences of transactions with certain random parameters. Each scenario represents an interesting sequence of individual transactions to hit a particular functional corner case. For example, a scenario in an ethernet networking operation would be a sequence of frames with a specified density – i.e. a certain portion of the time; the ethernet line is busy sending/receiving and idle for the remainder of the time. Scenario generators generate scenarios in random order and sequence, and produce a stream of transactions that corresponds to the generated scenarios.

This layer may be partially or completely subsumed by the test layer above it, depending on the amount of directedness required by the testcase. Similarly, generators must be able to be turned off, either from the beginning or in the middle of a simulation, to allow for the injection of directed stimulus. The generator must also be able to be restarted to resume the generation of random stimulus after a directed stimulus sequence.

Test Layer

Testcases involve modifying constraints on generators, the definition of new random scenarios, synchronization of different transactors and the creation of directed stimulus.

This layer may also provide additional testcase-specific self-checking not provided by the functional layer at the transaction level. Typically, this layer deals with high-level algorithm checks that govern the flow of traffic of the individual transactions, such as quality-of-service. It can also perform checks where correctness will depend on timing with respect to a particular synchronization event introduced by the testcase.

3

Common Message Service

Introduction

This chapter describes an object-oriented message service that shall be used by all components of a simulation. The section "Message Reporting Class - rvm_log" on page 278 details the user interface of the `rvm_log` class that implements the message service.

The message service uses the following concepts to describe and control messages: source, filters, type, severity and handling.

Message Source

Each instance of the `rvm_log` class represents a message source. A message source can be any component of a testbench: a command-layer transactor, a sub-layer of the self-checking structure, a testcase, a generator, a verification IP block or a complete verification environment.

Each message source has a descriptive name and an instance name. Regular expressions applied to the name and instance name of message sources are used to select several sources to be controlled at the same time. It is thus important that a relevant nomenclature be used to name message sources to allow them to be controlled in a useful fashion.

Message sources can also be collected into a logical hierarchy. A message source can be logically configured to be the parent of a set of message sub-sources. The same message source can also be configured to be a sub-source of another message source, along with other message sources. A complete hierarchy of message source can be controlled at the same time, regardless of their name, instance name, or physical object hierarchy. Used judiciously when building transactors and verification environments, it allows testcase writers to be able to control message sources for entire sub-systems of the verification environment or complex transactors, without having to be familiar with their internal structure.

Messages from each source can be controlled independently of the messages from other sources.

Message Filters

Filters can prevent or allow a message from being issued. Filters are associated and disassociated with message sources. They are applied in order of association and control messages based on their identifier, type, severity or content. They can promote or demote messages severities, modify message types and their simulation handling.

After a message has been subjected to all the filters associated with its source, its effective type and severity may be different from the actual type and severity originally specified in the code used to issue a message.

Message Type

Individual messages are categorized into different types by the author of the code used to issue the message. Assigning messages to their proper types allows a testcase or simulation to produce and save only (and all) messages that are relevant to the concerns addressed by a simulation.

For example, messages relating to timing may not be relevant unless a gate-level simulation is being performed. Similarly, messages related to X's detected on physical signals may not be relevant until the DUT has been completely configured and X's have had a chance to percolate out.

The following table summarizes the available message types and their intended purposes:

Table 3-1

Message Type	Purpose
Failure	An error has been detected. The severity of the error is categorized by the message severity.
Note	Normal message used to indicate the simulation progress.
Debug	Message used to provide additional information design to help diagnose the cause of a problem. Debug messages of increasing details are assigned lower message severities.
Timing	A timing error has been detected (e.g. setup or hold violation).
X Handling	An unknown or high-impedance state has been detected or driven on a physical signal.
Report, Notify, Protocol, Transaction, Command, Cycle	Message types used by VMT verification IP and provided for backward compatibility. Not used when creating verification environments or testcases.

Message Severity

Individual messages are categorized into different severities by the author of the code used to issue the message. A message's severity indicates its importance and seriousness and must be chosen with care.

For safety reasons, certain message severities cannot be demoted to arbitrary severities.

The following table summarizes the available message severities and their meaning:

Table 3-2

Message Severity	Indication
Fatal	The correctness or integrity of the simulation has been definitely compromised. By default, simulation is aborted after a fatal message is issued. Fatal messages can only be demoted into error messages.
Error	The correctness or integrity of the simulation has been compromised but simulation may be able to proceed with useful result. By default, error messages from all sources are counted and simulation aborts after a certain number have been observed. Error messages can only be demoted into warning messages.
Warning	The correctness or integrity of the simulation has been potentially compromised and simulation can likely proceed and still produce useful result.
Normal	This message is produced through the normal course of the simulation. It does not indicate that a problem has been identified.
Trace	This message identifies high-level internal information that is not normally issued.
Debug	This message identifies medium-level internal information that is not normally issued.
Verbose	This message identifies low-level internal information that is not normally issued.

Simulation Handling

Different messages require different action by the simulator once the message has been issued.

The following table summarizes the available message handling and their default trigger:

Table 3-3

Simulation Handling	Action
Abort	Terminates the simulation immediately and return to the command prompt, returning an error status. This is the default handling after issuing a message with a *fatal* severity.
Count as error	Count the message as an error. If the maximum number of such message from all sources has exhausted a user-specified threshold, the simulation is aborted. This is the default handling after issuing a message with an *error* severity.
Stop	Stop the simulation immediately and return to the simulation run-time control command prompt.
Debug	Stop the simulation immediately and start the graphical debugging environment.
Dump	Dump the callstack or any other context status information and continue the simulation.
Continue	Continue the simulation normally.

Creating Messages

This section specifies guidelines for creating messages from within components of a verification environment, the verification environment itself or testcases.

All simulation log output shall be done through the message service.

Do not use `printf()` to manually produce output messages. If a predefined method that produces output text must be invoked (such as the `rvm_data::display()` method), do so within the context of a message:

```
rvm_log log = new(...);
...
if (log.start_msg(log.NOTE_TYP, log.NORMAL_SEV)) {
   void = log.text("Executing transaction...");
   void = log.text();
   transaction.display("   ");
   log.end_msg();
}
```

Message of type "Failure" shall be of severity "Warning", "Error" or "Fatal" only.

A failure of lower severity does not make sense, except when being demoted to prevent its issuance.

Messages of type "Failure" should be issued using the "rvm_warning()", "rvm_error()" or "rvm_fatal()" macros.

These macros provide a shorthand notation for issuing single-line failure messages.

Message of type "Note" shall be of severity "Normal" only.

A note of higher or lower severity does not make sense, except when being demoted to prevent its issuance or promoted to detect unexpected code execution.

Messages of type "Note" should be issued using the "rvm_note()" macro.

This macro provide a shorthand notation for issuing single-line note messages.

Message of type "Debug" shall be of severity "Trace", "Debug" or "Verbose" only.

A debug message of higher severity does not make sense, except when being promoted to detect unexpected code execution.

Messages of type "Debug" should be issued using the "rvm_trace()", "rvm_debug()" or "rvm_verbose()" macros.

These macros provide a shorthand notation for issuing single-line debug messages.

Calls to printf() procedures should only be made once it has been confirmed that a message will be issued.

The `printf()`, `sprintf()` and `psprintf()` procedures are run-time expensive. They should only be called when their formatted output will actually be required. For example, the following code will always create the formatted output, whether or not the output is actually needed:

```
{
   string msg;
   sprintf(msg, "Executing command #%0d...", cmd_id);
   rvm_trace(log, msg);
}
```

However, the following two (equivalent) examples will only create the formatted output if the message will be issued:

```
rvm_trace(log, psprintf("Executing command #%0d...",
cmd_id));

if (log.start_msg(log.DEBUG_TYP, log.TRACE_SEV)) {
  void = log.text(psprintf("Executing command #%0d...",
cmd_id));
   log.end_msg();
}
```

Messages may contain the filename and line number where they were issued.

The pre-defined macros `__FILE__` and `__LINE__` can be used to include the name of the file and line number of the location where the message is issued:

```
rvm_trace(log, psprintf("%s, line %0d: Executing command
#%0d...", __FILE, __LINE__, cmd_id));
```

The file `$VERA_HOME/rvm_examples/log/ rvm_log_fileline.vri` redefines the message macros to prepend the filename and line number to each message.

Controlling Messages

This section specifies guidelines for controlling messages issued from within components of a verification environment, the verification environment itself or testcases.

Conditional compilation should not be used to control message issuance.

If messages are turned off using a pre-processor directive, they are turned off at compile time. To turn them back on, it is necessary to recompile the source file. Furthermore, messages are turned on or off for all instances. It is not possible to turn messages on or off for specific instances:

```
#ifdef TRACE_MSG
   rvm_trace(log, psprintf("Executing command #%0d...",
cmd_id));
#endif
```

Use the run-time message issuance control provided by the message service instead:

```
rvm_trace(log, psprintf("Executing command #%0d...",
cmd_id));
```

The method rvm_log::format() shall only be called in the constructor of the verification environment.

The message format is defined by the last invocation of this method and must be under the control of the user. A verification component that sets the message format may interfere with the desired format by the user.

The message format must also be defined in the environment so it will be consistent across all testcases on that environment.

The method rvm_log::stop_after_n_errors() shall only be called in the constructor of the verification environment.

The maximum number of error messages to issue before aborting the simulation is defined by the last invocation of this method and must be under the control of the user. A verification component that sets the maximum error message count may interfere with the desired count by the user.

Setting this count in the environment constructor allows individual testcases to override it.

4

Data and Transaction Models

Introduction

This chapter describes how to create data and transaction models that are reusable, constrainable and extendable.

One of the challenges when transitioning from a procedural language, such as Verilog or VHDL, to an object-oriented language such as OpenVera, is making effective use of the object-oriented programming model. This section contains guidelines or directives to help strike the right balance between objects and procedures.

Data and Transactions

Data units shall be modeled as objects

A data unit is any amount of data processed by the DUT. Packets, instructions, pixels, picture frames, SDH frames and ATM cells are all examples of data units. A data unit can be composed of smaller data units. Similarly, an object can be composed of smaller objects. For example, a picture frame object would contain thousands of pixel objects. Being modeled as objects, it is simple to create a stream of data units by creating a stream of object instances.

Transactions shall be modeled as objects

This one is not initially obvious. The natural tendency is to model transactions as procedures such as `read()` and `write()`. Using the RVM methodology, transactions are implemented using procedures but they are modeled (that is, defined) using objects. This approach offers the following advantages:

- It is easy to create a series of random transactions. Generating random transaction becomes a process identical to generating random data.
- Random transactions can be constrained. Constraints can only be applied to object properties. Constraining the transactions modeled using procedures requires additional procedural code. Procedural constraints, such as weights in a `randcase` statement, cannot be modified without modifying the source code, thus preventing reusability.
- New transactions can be added without modifying interfaces. A new transaction can be added by simply creating a new variant of the transaction object. No new class is created, no class interface is modified and no testcase is changed.

- It allows easier integration with the scoreboard. Since a transaction is fully described as an object, a simple reference to that object instance, passed to the scoreboard, is enough to completely define the stimulus and derive the expected response.

Data and transaction model classes shall be derived from the bu_data class

This base class provides a standard set of properties and methods proven to be useful in implementing verification environments and testcases. Furthermore, since OpenVera does not have the concept of a *void* or anonymous type, it provides a common base type for writing generic data processing and transfer components.

There should not be any protected members in data and transaction classes

Data and transaction objects should not have a need for protected properties or methods.

A channel class shall be declared for any class derived from the bu_data class

The channel object is the primary transaction and data interface mechanism used by transactors. It must be declared in the same file as the class it will carry to prevent possible multiple declarations.

The macro rvm_channel has been provided to make the declaration of the channel object simple and easy:

```
class my_data extends bu_data {
   ...
}
rvm_channel(my_data)  // No semicolon
```

This will automatically declare and implement a class named "my_data_channel" that can be used to transport an instance of "my_data" class or its derivatives. See “Transaction Interface Class - rvm_channel” on page 304 for more details on the channel object interface.

Properties / Data Members

This section gives directives for properties and methods that can be used to model data and transactions.

All data classes shall have a public static property referring to an instance of the message reporting object

This message reporting object instance is used to issue messages from any object instance when a more localized message source (such as a transactor) is not readily or clearly available. The property must be public to be controllable.

A class-static instance is used to avoid creating and destroying too many instances of the message reporting object as there will be thousands of object instances created and destroyed throughout a simulation.

```
class my_data extends rvm_data {
   static rvm_log log = new(“rvm_data”, “class”);
   task new() {
      super.new(this.log);
   }
}
```

Data and transaction descriptors will flow through various transactors in the verification environment. Messages related to a particular data object instance should be issued through the message reporting object in the transactor where the need to issue the message is identified. That way, the location of the message source can be easily identified - and controlled. Information about the data or transaction that caused the message can be included in the text of the message or by using the `rvm_data::display()` method.

```
class mii_xactor extends rvm_xactor {
   ...
   if (frame.fcs !== 32'h0000_0000) {
      if (this.log.start_msg(VMM_FAILURE_TYP,
                             VMM_WARNING_SEV)) {
         this.log.text("Invalid FCS bits in frame");
         frame.display("   ");
         this.log.end_msg();
      }
   }
   ...
}
```

Do not provide a new instance of a message reporting object with each data or transaction descriptor as this will cause significantly more run-time memory to be used and affect the run-time performance of the message service management procedures. It will not provide more information as the apparent source of the messages will be the same, regardless of the location of the data or transaction descriptor in the verification environment, making it more difficult to localize the problem.

Similarly, do not use the message reporting object of the transactor that created the data or transaction descriptor. The apparent source of all data-related messages would be that transactor, regardless of its current location in the verification environment.

All properties corresponding to a protocol property or field shall be rand

A property cannot be made rand after the fact, but it can be turned off.

A rand property shall be used to define the kind of transaction

A class must be able to model all possible kinds of transactions for a particular protocol. Do not use inheritance to model each individual transaction. Instead, use a property to identify the kind of transaction modeled by the object instance.

This example uses a single class to model both `read` and `write` transactions:

```
class bus_transaction {
   enum kind_t = READ, WRITE;
   rand kind_t kind;

   rand reg [15:0] address;
   rand reg [15:0] data;
}
```

The size() of a rand array property shall be unconditionally constrained to limit its value

The current array randomization semantics in OpenVera will cause the length of a randomized array to remain the same if it is unconstrained. In the following example, the size of the `data` property will not be randomized and remain equal to 5 because there are no constraints on `data.size()`.

```
class packet {
   rand reg [15:0] address;
   rand reg [ 7:0] data[*];
}
program p {
   packet p = new;
   p.data = new [5];
   repeat (10) vod = p.randomize();
}
```

However, should a constraint on `data.size()` be added later on, this definition of the length of the array no longer applies:

```
class my_packet extends packet {
   enum kind_t = CONTROL, USER;
   rand kind_t kind;

   constraint format {
      if (kind == CONTROL) data.size() == 2;
   }
}
```

Because, in the example above, there is a constraint on `data.size()`, the length of the array **will** be randomized. If the value `USER` is selected for the `kind` property, the size of the `data` array is effectively unconstrained and will be randomized to an average length of 2^{30}. A data model that was working fine suddenly breaks when extended. To avoid this situation, the size of a randomized array should **always** be constrained to a reasonable value. It is a good idea to locate this constraint in a separate constraint block to allow it to be turned off or overridden.

```
class packet {
   rand reg [15:0] address;
   rand reg [ 7:0] data[*];

   constraint limit_data {
      data.size() < 1024;
   }
}
```

All rand properties shall be public

This will make it possible to constrain them when creating sequences of random objects or via the `randomize-with` statement.

All non-rand properties should be local

Object state information should be accessed via public methods. This will ensure that the implementation can be modified while preserving the interface. Also, if properties are inter-related, using methods to set their value will ensure they remain consistent.

However, if you end up with a set of independent properties, each with a pair of `set()` and `get()` methods, you might as well make the property public and do away with the methods.

Transaction objects should have implementation and context references

Objects modeling transactions at a different layer in a protocol stack should have a list of references to the lower-level transactions used to implement them. This list would be added to by lower-level transactors in the verification environment as they implement the higher-layer transaction. The completed list will only be valid when the transaction's processing has ended. A scoreboard can then make use of the list of sub-transactions to determine its status and what response to expect.

Conversely, a low-level transaction should have a reference to the higher-level transaction it helps implement. This will help the scoreboard or other verification environment components to make sense of the transaction and help determine the expected response.

Example:

```
class usb_packet extends rvm_data {
   ...
   usb_transaction context;
   ...
}

class usb_transaction extends rvm_data {
   ...
   usb_packet   packets[$];
   usb_transfer context;
   ...
}

class usb_transfer extends rvm_data {
   ...
   usb_transaction transactions[$];
   ...
}
```

Data protection properties shall model their validity, not their value

The information in a CRC, HEC or parity properties is not in the actual value of the property but in their correctness. These properties must be modeled using a mask, indicating which bit of the property value is to be valid or corrupted (on transmission) or was valid or not (on reception). A value of 0 indicates a 100% valid value.

The actual value of these properties is computed using methods and inserted or compared in the data object only upon packing and unpacking.

Fixed payload data shall be modeled using explicit properties

Some protocols define fixed fields and data in normally user-defined payload for certain data types. For example, fixed-format 802.2 link-layer information may be present at the front of the user data payload in an Ethernet frame. Another example is the management-type frame in 802.11: the content of the user-payload is replaced with protocol management information.

Fixed payload data should be modeled using explicit properties, as if they were located in non-user-defined fields. The length of the remaining user-defined portion of the payload should be reduced by the corresponding number of bytes used by the fixed payload data, not modeled in explicit properties.

User-defined data is often modeled as an array of bytes. Leaving it up to the user to correctly interpret or format fixed payload data is error-prone. Applying constraints to payload elements becomes cumbersome as fixed data may overlap multiple bytes or be concatenated in the same byte.

Class inheritance shall not be used to model variance

Data units and transactions often contain information that is optional or is unique to a particular kind of data or transaction. For example, Ethernet frames may or may not have virtual LAN (VLAN) properties. Another example is a USB isochronous transaction which does not have any acknowledgement while the other types of transactions do.

Using traditional object-oriented design practices, inheritance looks like an obvious implementation: use a base class for the common properties then extend it for the various differences. This creates three problems, two of which are related to randomization and constraint - concerns that do not exist in traditional object-oriented languages.

The first problem is the difficulty of generating a stream containing a random mix of different kinds of objects. Because they share a common base type, they can use the same interfaces. In OpenVera, objects must first be instantiated before they can be randomized. Because instances must be created based on their ultimate type, not their base type, the particular format or variance of a class would be determined before randomization of its content. It would thus be impossible to use constraints to control the distribution of the various object formats or to express constraints on an object format as a function of the content of some other property (for example, if the destination address is equal to this, then the packet must have a VLAN tag).

The second problem is the difficulty of adding constraints to be applied to all formats or variations of a class. To add constraints to a transaction or object, the most flexible mechanism is to create an derived class. To add a constraint that must apply to all variations of a class cannot be done by simply extending the base class common to all variations as it simply creates yet another class unrelated to the other derivatives. It would require extending each one of the ultimate class extensions. Aspect-oriented programming helps alleviate this concern but does not help resolve the second problem.

The final problem is that you will not be able to recombine different but orthogonal extensions. For example, an Ethernet frame may have the optional VLAN properties and may have the optional SNAP link-layer information. Because OpenVera does not support multiple inheritance, using inheritance to model this simple case will require four different classes: one for each combination of the presence or absence of the optional properties. Add another orthogonal optional property, and the number of classes goes up to eight. This approach creates an exponential growth in the number of classes. This problem could be solved if the language supported multiple inheritance but it would not help in solving the more serious previous two problems.

A rand property shall be used to indicate if optional or unique properties are present

Instead of using class inheritance to model the presence of optional or different properties, use the value of a discriminant property. It will be necessary for methods (such as `byte_pack()` and `compare()`) to procedurally check the value of these discriminant properties to determine the proper course of action.

Because a single class is used to model all variations, constraints can be specified to apply to all variants of a data type. Also, constraints can be used to express relationships between the format of the data and the content of other properties. Orthogonal variations are modeled using different discriminant properties allowing all combinations of variations to occur within a single model.

Example:

```
class ethernet_frame {
   rand reg [47:0] da;
   rand reg [47:0] sa;
   rand reg        has_vlan; // Discriminant property
   rand reg        cfi;      // if has_vlan == 1
   rand reg [ 2:0] priority; // if has_vlan == 1
   rand reg [11:0] vlan_id;  // if has_vlan == 1
   rand reg [15:0] len_typ;
   rand integer    len;
   rand reg [7:0]  data[*] dynamic_size len;
   rand reg        fcs;

   virtual task display(string prefix = "") {
     printf("%sDA=48'h%h, SA=48'h%h, len/typ=16'h%h\n",
        prefix,da, sa, len_typ);
     if (has_vlan) {
        printf("%sVLAN: cfi=%b pri=%0d, id=12'h%h\n",
           prefix, cfi, priority, vlan_id);
     }
     printf("%sData=%h %h .. %h (length=%0d)\n", prefix,
         data[0], data[1], data[data.size()-1], len);
        printf("%sFCS = %s\n", prefix,
          (fcs) ? "good" : "BAD");
}
```

This approach has only one disadvantage: there is no type checking to prevent the access of a property that is not currently valid given the current value of a discriminant property.

Methods

The constructor should be callable without any arguments

This will allow the use of the predefined atomic generators (see "Atomic Generator Transactor - rvm_atomic_gen" on page 357) and scenario generators (see "Scenario Generator Transactor - rvm_scenario_gen" on page 365).

All data object methods should be virtual

This will allow the functionality of methods to be extended in class derivatives.

All data object methods shall be nonblocking

Methods in data objects should only be concerned with the immediate state of the object. There should not be any need for the simulation time to advance or for the execution thread to be suspended within data object methods.

Transaction objects should have a method returning completion status information

Once the processing of a transaction is complete, a *status* property should be available to qualify the correctness of the transaction. The status indication depends on the error detection and injection capability of the transactors and the error recovery mechanisms in the protocol.

For example, the return value from a status method could indicate if a transaction was retried, how many times it was retired, and whether or not it was ultimately successful.

All classes derived from the bu_data class shall provide implementations for the "display", "allocate", "copy" and "compare" virtual methods

Do not rely on the built-in `object_print()`, `object_copy()` and `object_compare()`. They may not be available (for example, in SystemVerilog) and may not provide a suitable user interface. A user-defined method provides for a more consistent and reliable interface. The implementation of these methods may choose to use the built-in methods if relevant.

All classes derived from the bu_data class should provide implementations for the "byte_size", "byte_pack" and "byte_unpack" virtual methods

Do not rely on the OpenVera built-in pack/unpack procedures and methods (for example, `pack()`, `unpack()`, `vera_pack()` and `vera_unpack()`. They may not be available (for example, in SystemVerilog) and may not provide a suitable user interface or correct results. Furthermore, they may impose a specific layout of the data properties that will complicate the constraint and data usage model.

User-defined methods provide for a more consistent and reliable interface. The implementation of these methods may choose to use the built-in methods if relevant.

It is necessary to implement these methods if a data model needs to be transmitted to across a physical interface or between different simulations (for example, from Vera to SystemC).

The implementation of the byte_pack() method shall only pack the relevant properties based on the value of discriminant properties.

Not all properties may be valid or relevant under all possible variances of an object. The packing methods must check the value of discriminant properties to determine which property to include in the packed data, in addition to their format and ordering.

Example:

```
class ethernet_frame {
   rand reg [47:0] da;
   rand reg [47:0] sa;
   rand reg        has_vlan; // Discriminant property
   rand reg        cfi;      // if has_vlan == 1
   rand reg [ 2:0] priority; // if has_vlan == 1
   rand reg [11:0] vlan_id;  // if has_vlan == 1
   rand reg [15:0] len_typ;
   rand integer    len;
   rand reg [7:0]  data[*] dynamic_size len;
   rand reg        fcs;

   virtual function integer byte_pack(var reg [7:0]
bytes[*]) {
      integer i;
      bytes = new[this.byte_size()];

      ...
      if (has_vlan) {
         ‘{bytes[i], bytes[i+1]} = 16’h8100;
         i += 2;
         ‘{bytes[i], bytes[i+2]} =
            {this.cfi, this.pri, this.vlan_id};
         i += 2;
      }
      ...
   }
}
```

The byte_unpack() method shall interpret the packed data and set discriminant properties appropriately

Often, discriminant properties are logical properties, not directly packed into bit-level data nor directly unpacked from it. However, the information necessary to identify a particular variance of a data object is always present in the packed data. For example, the value 0x8100 in the “len_typ” in bytes 12 and 13 of an Ethernet mac frame stream indicate that the VLAN identification fields are present in the next two bytes.

The unpacking methods must interpret the packed data and set the value of the discriminant properties accordingly. Similarly, it must set all relevant properties to their interpreted values based on the interpretation of the packed data. Properties not present in the data stream should be set to unknown or undefined values.

A method shall be provided to compute the correct value of each data protection property

Because the data protection property is encoded simply as being valid or not, it must be possible to derive its actual value by other means when necessary.

The packing method is responsible for corrupting the value of a data protection property if it is modeled as invalid, not the computation method.

Provide a method to check internal consistency of related properties

The user can modify the values of public properties at any time. If properties have related values, it is possible for a user to modify one property without properly modifying the others. For example, a property used to define the length of a payload array may have a different value than the actual length of the array.

A method should be provided to ascertain the integrity of the values of all the properties in the object. This method should be called whenever the values of the properties are about to be used – before packing the object or computing a CRC value, for example.

Constraints

A constraint block shall be provided to ensure the validity of randomized property values

Some properties may be modeled using a type that can yield invalid values. For example, a "length" property can be modeled using an integer. This constraint would ensure that any length value is positive. Note that "valid" is not the same thing as "error-free". "Validity" is a requirement of the model used, not of the object being modeled.

This constraint block must never be turned off nor overridden hence it is a good idea to use a unique name, such as “class_name_valid”.

Constraint blocks should be provided to produce better distributions size or duration properties

Size and duration properties do not have equally interesting values. For example, short or back-to-back and long or drawn-out transactions are more interesting than average transactions. Randomized properties modeling size, length, duration or intervals should have a constraint block that distributes their value equally between limit and average values.

A distribution constraint block shall constrain a single property

Use one constraint block per property to make it easy to turn off or override without affecting the distribution of other properties.

Discriminant properties should be solved before dependent properties

A conditional constraint block does not imply that the properties used in the expression are solved before the properties in the body of the condition. If a property in the body of the condition is solved with a value that implies that the condition cannot be true, this will further constrain the value of the properties in the condition. If there is a greater probability of falsifying the condition, this makes it less likely to get an even distribution over all discriminant values. For example, the following example is unlikely to produce CONTROL packets because there is a low probability of the length property to be solved as 1:

```
class some_packet {
   enum kind_t = DATA, CONTROL;
   rand kind_t kind;

   rand integer length;
   rand reg [7:0] data[*] dynamic_size length;

   constraint some_packet_valid {
      length >= 0;
      if (kind == CONTROL) {
         length == 1;
      }
   }
}
```

This problem can be avoided, and a better distribution of discriminant properties obtained, by forcing the solving of the discriminant property before any dependent property:

```
class some_packet {
   enum kind_t = DATA, CONTROL;
   rand kind_t kind;

   rand integer length;
   rand reg [7:0] data[*] dynamic_size length;

   constraint some_packet_valid {
      length >= 0;
      if (kind == CONTROL) {
         length == 1;
      }
      solve kind before length;
   }
}
```

Constraint blocks shall be provided to avoid errors in randomized values

Error can be randomly injected by selecting the invalid value for error protection properties. A constraint block should keep the value of such properties to valid by default.

An error-prevention constraint block shall constrain a single property

Use one constraint block per error injection property to make it easy to turn off or override without affecting the correctness of other properties.

Testcase Configuration Descriptor

The concept of randomly generating data or transactions supplied to a design is a concept that is easily understood. The same principle can be applied to the configuration of the DUT and the testcase. Even though a DUT is typically configured only once during a simulation, the particular configuration used can be randomly selected. For example, a DUT could be configured to allocate the bandwidth of 16 pins between one to sixteen different channels. Instead of building a verification environment for a limited number of configurations (for example, 16x1 and 1x16), the configuration and associated verification environment can be randomly selected.

Constraints can be used to select a particular subset of the possible configuration (maybe because the DUT or the verification environment does not yet support some configuration features) or limit the random values to interesting configurations. Functional coverage can be used to measure and record the configurations that have been verified.

The configuration descriptor shall assume a constant DUT footprint

The DUT is instantiated in Verilog or VHDL. Because the DUT model is compiled and elaborated separately from the OpenVera verification environment, it is not possible to modify the physical interface of the DUT model based on a randomly generated configuration. The configuration descriptor must thus assume a fixed, invariant physical interface for the DUT.

The configuration descriptor shall model the configuration, not the register values

The configuration descriptor should model the DUT configuration at a high-level of abstraction, not the values of the registers used to encode the configuration. First, the configuration will be easier to interpret by the scoreboard to determine the expected response. Second, the configuration will be easier to constrain.

For example, the baud rate in a UART may be encoded using sequential binary values for the baud rates 9200, 14400, 28800 and 56k. The configuration should be modeled using an integer value constrained to be equal to one of the valid baud rate values, not a 2-bit register:

```
class dut_config {
   integer baud_rate;

   constraint dut_config_valid {
      baud_rate in {9600, 14400, 28800, 56000};
   }
}
```

Configuration of all variable aspects external to the DUT shall be randomized in the rvm_env::gen_cfg() method

Certain aspects of the verification environment can be randomized, independently of the configuration of the design under test. For example, the number of master and slave devices on a bus can be randomized. Similarly, the MAC addresses of network devices attached to the DUT can also be randomized.

Any aspect of the verification environment that is arbitrarily chosen and not fixed by the DUT requirement must be randomized. The verification environment is then built and configured accordingly.

The default value of the configuration descriptor shall match the default DUT configuration

All designs, after completing a reset cycle, come up with a specific default configuration. The default value of the configuration descriptor, as returned by the constructor, must match this configuration.

The method that will compile the value of the configuration descriptor and download it into the design under test will likely be incrementally implemented to parallel the functionality available in the verification environment. The configuration description must thus, by default, match the design configuration that will not be modified.

The generated value of the configuration descriptor shall be downloaded into the DUT in the rvm_env::cfg_dut_t() method

The value of the (usually randomly selected) configuration must be downloaded into the DUT. This will usually require the interpretation or compilation of the high-level device configuration description into the required register settings.

This may seem like a daunting task. But it is only a formal description of the intellectual process you would have to go through to configure the design to a particular state. Should the device configuration model change or need to be debugged, it will be much easier to maintain such a method than reverse-engineering a series of write cycles of seemingly arbitrary values to arbitrary addresses.

It is not necessary to develop the device configuration download method all at once. Just as the implementation of the verification environment and self-checking structure usually proceeds in incremental steps, supporting ever-increasing functionality, the device configuration method can be similarly evolved to modify the device configuration from its default value. A constraint block must be used to prevent the currently unsupported configuration features from being randomly selected.

A possible implementation strategy would be to leverage the device driver code. The device configuration code, usually written in C or C++, can be linked in the testbench simulation (Pioneer-NTB, Vera, or VCS-NTB) and use callbacks to the command-layer transactors in the testbench simulator (Pioneer-NTB, Vera, or VCS-NTB) to perform register read and writes. By passing to the C code a description of the configuration to be download into the DUT, as generated by the testcase, the actual device driver code can be developed and verified in parallel with the RTL verification.

A reference to the testcase configuration descriptor shall be passed to the constructor of the self-checking structure in the rvm_env::build() method

The self-checking structure needs to know the configuration of the design and surrounding components to determine the expected response. A constraint block should be used to prevent any currently unsupported configurations features from being randomly selected.

A constraint block named “limitations” shall be provided to eliminate currently unsupported configurations

A verification environment and a design usually evolve during a project. Not all functionality or configuration options may be available. Do not model these temporary limitations in the configuration descriptor itself. Instead, model the intended functionality and use a constraint block to prevent the unsupported features or configuration from being randomly selected. This will provide a better documentation of the intent of the verification environment and its current limitations. Constraints can be removed as new features are added to the design or the verification environment.

At the end of the project, this constraint block should be empty.

5

Stimulus and Generation

Introduction

The generation of data objects (packets, frames, instructions) or transactions is modeled separately from the objects themselves because of the different dynamics of their respective lifetimes. In a typical simulation, there will be thousands of data objects or transactions created, flowing through transactors, recorded and compared in the self-checking structure. On the other hand, there will be only a handful of object and transaction sources that need to exist at the beginning of the simulation and remain in existence until the end.

This section gives guidelines on how to write autonomous generators that will create a stream of random data or transactions. They will be designed to be easily externally constrained, without requiring modifications of their source code. Constrained random tests are then written, not by writing a completely new or slightly modified generator, but by adding constraints to the reusable generators that already exist.

Generation can also be a manual (directed) process, where transactions and data are individually created and passed to the appropriate transactor. This process is covered in the directed testcase section.

Generator Components

The following directives will help you to model object-oriented, reusable and externally-constrainable generators. All of these guidelines are demonstrated in the example generators found in:

```
$VCS_HOME/doc/examples/nativetestbench/rvm/atm

$VERA_HOME/examples/rvm_examples/scenario

$PIONEER_HOME/examples
```

The `rvm_atomic_gen` and `rvm_scenario_gen` macros can be used to automatically generate generator transactors for any user-defined class. These generators follow the guidelines outline in this chapter. See “Atomic Generator Transactor - rvm_atomic_gen” on page 357 and “Scenario Generator Transactor - rvm_scenario_gen” on page 365.

A generator shall be modeled as a transactor

As such, all guidelines applicable to transactors are applicable to generators, unless explicitly superseded in this section. See “Transactors” on page 95 for more details.

A generator shall generate a single stream of instances of a specific class

A generator must be concerned with generating only a single stream of a particular type of objects. The generated stream could represent a stream of data packets, processor instructions, or video frames.

The synchronization of concurrent data streams is a testcase issue.

A generator shall be implemented in a class named "bu_xyz_dataclass_gen"

Where "xyz" is the name of the protocol this generator belongs to and "dataclass" is the name of the class generated by the generator. This will make identifying generator classes and the generated stream type easier and document to purpose of a generator.

A generator shall have a single output channel

This is a consequence of the "single stream of instances" guideline. A single stream requires only one output channel.

The reference to the generator output channel shall be in a public property named "out_chan".

For example:

```
class abc_eth_mac_frame_gen extends abc_xactor {
   abc_eth_mac_frame_channel out_chan;
}
```

This will allow several important operations required to implement testcases or build verification environments:

- Directed stimulus can be directly injected into the channel

- The channel can be queried, controlled or reconfigured
- The channel can be referenced as the input channel for a downstream transactor
- The channel can be replaced if dynamic environment reconfiguration is required.

A reference to a pre-existing output channel instance shall be optionally specifiable to the generator constructor

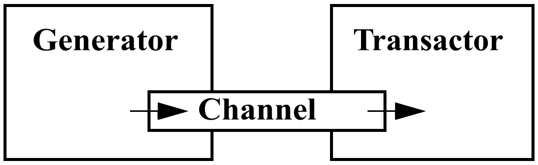
If no channel instance is specified, then the output channel is instantiated in the constructor.

```
class abc_eth_mac_frame_gen extends abc_xactor {
   abc_eth_mac_frame_channel out_chan;

   task new(string                    instance,
            integer                   stream_id = -1,
            abc_eth_mac_frame_channel out_chan  = null)
   {
      super.new("ABC Ethernet MAC Frame Generator",
                instance, stream_id);
      if (out_chan == null) {
      out_chan = new({name, " Output Channel"}, instance);
      }
      this.out_chan = out_chan;
   }
}
```

Connecting two transactors - such as a generator to a driver - requires that the output channel of the upstream transactor be the input channel of the downstream transactor. This can only be accomplished if they share references to a single channel instance.

Figure 5-1 Connecting a Generator to a Transactor



The steps to connect two transactors are to let the first one to internally instantiates its channel, then pass a reference to that channel to the constructor of the last one to be instantiated.

```
task dut_env::build(){
   abc_eth_mac_frame_gen gen = new("G0", 0);
  abc_eth_mac_xactor   mac = new("M0", 0, gen.out_chan);
}
```

Alternatively, a stand-alone channel can be instantiated then passed to the constructor of both transactors.

```
task dut_env::build(){
   abc_eth_mac_frame_channel gen_to_mac =
                        new("Gen to Mac Channel", "U0");
   abc_eth_mac_frame_gen gen = new("G0", 0, gen_to_mac);
   abc_eth_mac_xactor    mac = new("M0", 0, gen_to_mac);
}
```

A generator shall randomize a single instance, located in a public property, then copy the final value to a new instance

This is called a *factory pattern* and yields the most controllable generator:

```
class abc_eth_mac_frame_gen extends abc_xactor {
   ...
   abc_eth_mac_frame randomized_fr;

   task new(...) {
      ...
      this.randomized_fr = new;
   }
   ...
      while (1) {
         abc_eth_mac_frame fr;
         if (!this.randomized_fr.randomize()) ...
         cast_assign(fr, this.randomized_fr.copy());
         ...
         out_chan.put_t(fr);
      }
   ...
}
```

See “Modifying Constraints” on page 249 for the various constraint control mechanisms that can be used to control this generator pattern.

The name of the property containing the randomized instance shall have the prefix "randomized_"

This will make it easier to identify the location, name and type of all randomized instances in a verification environment. It also clearly identifies the purpose of the property.

The return value of the randomize() method shall be checked and an error be reported if it is FALSE

If a contradiction in a set of constraints make it impossible for the solver to find a solution, the `randomize()` method returns FALSE. It is important that an error be reported to indicate the problem with the constraints in the status of the simulation and to prevent a partial solution from being used.

```
if (!this.randomized_fr.randomize()) {
   rvm_error(this.log, "Unable to find a solution");
   continue;
}
```

Also, after reporting a message with an ERROR severity, the simulation will eventually abort once a maximum number of contradictions have been identified. Otherwise, a simulation may be stuck trying to find a solution to a problem where none exist.

The "stream_id" property of the randomized object shall be assigned the value of the "stream_id" property in the generator before each randomization

To ensure that the user does not accidentally modify the stream identifier in the randomized instance, they should be set before every randomization attempt:

```
while (1) {
   ...
   this.randomized_fr.stream_id = this.stream_id;
   if (!this.randomized_fr.randomize()) ...
   ...
}
```

Stream identifiers are defined in all data model classes. They are used to specify stream-specific constraints when constraints are added using out-of-body constraints blocks or aspect additions. For example, to constrain transactions in the first two streams:

```
extends my_test(ahb_transaction) {
   constraint interesting_scenario {
      if (stream_id < 2) {
         ...
      }
   }
}
```

Atomic Generators

Atomic generation is the process of generating a single instance of a random data object or transaction at a time. It may be sufficient if the validity of the protocol or reaching all of the functional coverage space does not depend on specific or complex sequences of data objects or transactions.

The rvm_atomic_gen macro can be used to automatically generate an atomic generator transactor for any user-defined class. The atomic generator follows the guidelines outline in this section. See "Atomic Generator Transactor - rvm_atomic_gen" on page 357.

Generation shall automatically stop after a configurable number of objects have been generated

One of the means for controlling the duration of a simulation is to limit the number of input data items to process. Rather then having to implement a count-then-stop procedure in every verification environment, it is much simpler to implement a configurable number of objects to generate in the generator itself.

A number of object equal to 0 shall be interpreted as infinity.

Because the generation thread is located in the `rvm_xactor::main_t()` method, the generation process can be reset and restarted after the configured number of objects has been generated.

```
class abc_eth_mac_frame_gen extends abc_xactor {
   ...
   task new(..., integer generate_n = 0, ...)
   {
      ...
      this.generate_n = generate_n;
   }
   ...
      while (this.generate_n == 0 ||
             this.object_id < this.generate_n) {
      ...
      this.randomized_fr.object_id  = this.object_id++;
         if (!this.randomized_fr.randomize()) ...
         ...
      }
   ...
}
```

A ON/OFF triggered event named DONE shall be indicated when the specified number of objects has been generated

This will allow the verification environment to determine when all generators have completed their stream generation.

```
class abc_eth_mac_frame_gen extends abc_xactor {
   ...
   task new(...)
   {
      ...
      this.DONE =
         this.notify.configure(*,
           this.notify.ON_OFF_TRIGGER);
   }
   ...
      while (...) {
         ...
      }
      this.notify.indicate(this.DONE);
   ...
}
```

The "object_id" property of the randomized object shall be assigned an incrementing value before each randomization

The generator must maintain an internal *object_id* counter and assign the value of that counter to the randomized object's *object_id* property:

```
while (1) {
   ...
   this.randomized_fr.object_id   = this.object_id++;
   if (!this.randomized_fr.randomize()) ...
   ...
}
```

Object identifiers are defined in all data model classes to enable the specification of object-specific constraints within a stream. For example, to constrain every fourth transactions to be a WRITE cycle:

```
extends my_test(ahb_transaction) {
   constraint interesting_scenario {
      if (object_id % 4 == 0) {
         kind == WRITE;
      }
   }
}
```

The "object_id" counter shall be in a local property

The value of the counter must not be externally modified and managed strictly by the generator itself:

```
class abc_eth_mac_frame_gen extends abc_xactor {
   ...
   local integer object_id;
   ...
}
```

The "object_id" counter shall start at 0 and be reset to 0 whenever the generator is reset

Resetting the generator is like starting the stream to the beginning and thus the *object_id* counter must be reset. This will also allow multiple runs of the configured number of objects to be generated, every time the generator is reset then restarted.

```
class abc_eth_mac_frame_gen extends abc_xactor {
   ...
   task new(...){
      ...
      this.object_id = 0;
   }
   ...
   virtual task reset_xactor(integer rst_type = 0){
      super.reset_xactor(rst_type);
      this.object_id = 0;
   }
}
```

The "scenario_id" property of the randomized object shall be assigned the value of the generator's scenario_id property before each randomization

The generator must maintain an internal scenario_id counter and assign the value of that counter to the randomized object's `scenario_id` property:

```
while (1) {
   ...
   this.randomized_fr.scenario_id = this.scenario_id;
   if (!this.randomized_fr.randomize()) ...
   ...
}
```

The "scenario_id" counter shall be in a local property

The value of the counter must not be externally modified and managed strictly by the generator itself:

```
class abc_eth_mac_frame_gen extends abc_xactor {
   ...
   local integer scenario_id;
   ...
}
```

The "scenario_id" counter shall start at 0 and be incremented whenever the generator is reset

Resetting the generator is like starting the stream for the beginning and thus creating a new random scenario:

```
class abc_eth_mac_frame_gen extends abc_xactor {
   ...
   task new(...){
      ...
      this.scenario_id = 0;
   }
   ...
   virtual task reset_xactor(integer rst_type = 0){
      super.reset_xactor(rst_type);
      this.scenario_id++;
      ...
   }
```

The generation thread shall call the rvm_xactor::wait_if_stopped_t() method before randomizing the next instance

This will allow the introduction of delays in a stream. It also guarantees that the randomized instance can be safely substituted while the generator is stopped.

```
class abc_eth_mac_frame_gen extends abc_xactor {
   ...
   while (1) {
      ...
      super.wait_if_stopped_t();
      this.randomized_fr.stream_id   = this.stream_id;
     this.randomized_fr.scenario_id = this.scenario_id;
     this.randomized_fr.object_id   = this.object_id++;
      if (!this.randomized_fr.randomize()) ...
      ...
   }
   ...
}
```

A ONE_SHOT event, named GENERATED, should be indicated immediately before adding a new instance to the output channel

This will allow other components in the verification environment or tests to synchronize themselves with the generated output stream.

```
class abc_eth_mac_frame_gen extends abc_xactor {
   ...
   integer GENERATED;

   task new(...){
      ...
      this.GENERATED =
         this.notify.configure(*,
this.notify.ONE_SHOT_TRIGGER);
   }
   ...
      while (1) {
         ...
         this.notify.indicate(this.GENERATED, fr);
         this.out_chan.put_t(fr);
      }
   ...
}
```

The newly generated instance should be used as the status information for the indicated GENERATED event

This will allow other components in the verification environment or tests to gain access to the generated data without having to extend a callback method. See the previous guideline for an example.

A callback method shall be called after each instance is generated

This will allow other components in the verification environment or tests to record the generated data for response checking, sample the generated data in a functional coverage model or modify the generated data with directed stimulus or inject errors.

```
class abc_eth_mac_frame_gen_callbacks
   extends rvm_xactor_callbacks {

   task post_gen_t(abc_eth_mac_frame_gen who,
                   abc_eth_mac_frame     what,
                   var reg               drop) {}

}

class abc_eth_mac_frame_gen extends abc_xactor {
   ...
   task post_gen_t(abc_eth_mac_frame fr,
                   var reg           drop) {}
   ...
   while (1) {
      ...
      {
         abc_eth_mac_frame_gen_callbacks cb;
         reg                             drop = 0;
         this.post_gen_t(fr, drop);
         foreach (this.callbacks, i) {
       if (!cast_assign(cb, this.callbacks[i], CHECK)) {
               continue;
            }
            cb.post_gen_t(this, fr, drop);
         }
         if (drop) continue;
      }
      this.out_chan.put_t(fr);
   }
   ...
}
```

Scenario Generators

Atomic generation is unlikely to generate most of the interesting transaction or data sequences. For example, randomly generating CPU instructions is unlikely to generate a valid loop structure – much less a nested loop structure. Using a million monkeys for a million years is not a suitable strategy to write Shakespearean plays.

The structure of a random sequence generator is similar to that of an atomic generator. Instead of generating individual data objects, it generates an array of data objects. The following additional guidelines address the challenges of implementing reusable random sequence generators where new scenarios can be defined without modifying the generator itself.

The `rvm_scenario_gen` macro can be used to automatically generate a scenario generator transactor for any user-defined class. This scenario generator follows the guidelines outlined in this section. See “Scenario Generator Transactor - rvm_scenario_gen” on page 365.

A scenario descriptor shall be randomized

A sequence generator has the same structure as an atomic generator. The only difference is that the former randomizes a scenario descriptor, which contains an array of data objects, instead of a single object directly.

```
class apb_scenario {
   ...
   rand apb_transaction items[$];
}
```

Scenario descriptors shall have a random "kind" property

Scenarios will be defined conditionally based on the (randomly selected) *kind* property. New scenarios are defined as new values of the *kind* property.

```
class apb_scenario {
   rand integer kind;
   rand apb_transaction items[$];
   ...
}
```

The default scenario shall be a single, unconstrained item

The default scenario reduces to an atomic generator.

```
class apb_scenario {
   static integer NULL = 0;
   ...
   constraint null_scenario {
      if (kind == NULL) items.size() == 1;
   }
}
```

Scenario descriptors shall have a reference to the generator output channel

The reference, in a *protected* property, is used to send the content of the scenario to the generator output channel.

```
class apb_scenario {
   protected apb_transaction_channel out_chan;

   rand integer kind;
   rand apb_transaction items[$];
   ...
}
```

Scenario descriptors shall have a virtual apply_t() method to send the generated scenario to the output channel

This method can be extended to implement scenarios that are better described using procedural code or that need run-time interaction with the environment or the DUT. The default implementation of this method simply forward a copy of the content of the random data or transaction array to the output channel.

```
class apb_scenario {
   protected apb_transaction_channel out_chan;

   rand integer kind;
   rand apb_transaction items[$];
   ...
   virtual protected task apply_t() {
      foreach (this.items, i) {
         this.out_chan.put_t(this.items[i].copy());
      }
   }
}
```

Directed Generation

Generators should provide a procedural interface to create directed transactions

For many, having a procedural interface where transactions are specified by calling different procedures remains desirable. The generator structure is compatible with this usage model by providing a set of procedures to create transactor objects.

Whenever a directed procedural interface method is called, a new instance of a transaction object is allocated. Its properties are set according to the parameter values specified in the procedure call. The final transaction object is then subjected to the callback methods before being added to the generator output channel. The procedure returns when the `rvm_channel::put_t()` method returns.

```
class bu_apb_generator extends bu_xactor {
   bu_apb_transaction_channel out_channel;
   ...
   task write(reg [ 7:0] sel,
         reg [31:0] addr,
         reg [31:0] data) {
      bu_apb_transaction tr = new;
      tr.kind = bu_apb_transaction::WRITE;
      tr.addr = addr;
      tr.data = data;
      if (call_callbacks(tr)) {
         this.out_chan.put_t(tr);
      }
   }
}
```

Directed stimulus must not be directly added to the public output channel

Directed stimulus can be easily introduced in the output stream of the generator by directly putting instances of transaction descriptors in the output channel. This is accomplished by calling the `rvm_channel::put_t()` method directly. But such stimulus would not be subjected to the callbacks methods of the generator and must be avoided.

The reference to the output channel of a generator is public to allow for dynamic reconfiguration of an environment and to connect it to a downstream transactor.

Directed scenarios should be implemented as extensions of the scenario descriptor's apply_t() method

Directed sequences should be included in random tests as another available scenario. That way, they will be surrounded by other random or directed scenarios and may uncover an unexpected problem.

A directed scenario may require run-time interaction with the environment to be fully specified or may be simpler to describe procedurally than declaratively (such as using a *for* loop to generate a long sequence of identical objects). The data may be a priori generatable but may also depend on run-time feedback information from the environment. It is implemented by overloading the `rvm_scenario::apply_t()` method.

```
extends
my_procedural_scenario(bu_eth_mac_frame_scenario) {
   static integer MY_PROCEDURAL;

   constraint my_directed_scenario {
      if (kind == MY_PROCEDURAL) items.size() == 0;
   }
```

```
      before protected task apply_t() {
         if (kind == MY_PROCEDURAL) {
            ...;
            return;
         }
      }
   }
```

Extensions of the scenario descriptor's apply_t() method shall not execute the default implementation

The default implementation of the `rvm_scenario::apply_t()` method simply forwards the content of the generated scenario to the generator's output channel. If a procedural directed scenario leaves the scenario array non-empty and the default implementation is called, some items may be added twice to the output channel.

The default implementation of the method can be skipped by either not calling `super.apply_t()` method in a object-oriented class extension or by returning from the method in an aspect-oriented class extension.

```
extends
my_procedural_scenario(bu_eth_mac_frame_scenario) {
   static integer MY_PROCEDURAL;

   after protected task register_scenarios() {
      this.register_scenario(MY_PROCEDURAL, "My
Procedural");
   }

   constraint my_directed_scenario {
      if (kind == MY_PROCEDURAL) length == 0;
   }

   before protected task apply_t() {
      if (kind == MY_PROCEDURAL) {
         ...;
         return;
      }
```

```
        }
    }
```

Embedded Generators

As data flows through layers of transactors on their way to be applied to the design under verification, additional data may be required to correctly format the data according to the requirements of the lower layers. For example, a link-layer header may need to be added to a transport-layer packet. This additional information should be generated within the transactor where it is added. It should not be added to the higher-layer transaction descriptor to avoid making it dependent on the lower protocol layers (which it may not traverse in a transaction-level environment) or to have to include information for all possible lower-layers that the transaction may potentially traverse.

Transactors also need to deviate from standard behavior in preferably random ways. Deviations could be answering with a negative acknowledge or corrupting the CRC field or waiting for some delays before or during the execution of a transaction. These deviations should be introduced by randomly generating a deviation descriptor within the transactor. Of course, default constraints should prevent (or limit the kind of) the introduction of deviations by default. Deviations can be introduced by removing or modifying those default constraints when they become the focus of a set of tests.

The following directives will help you to model externally-constrainable generators embedded in transactors.

The randomized instance shall be in a public property

This is the same guideline used for implementing generator components. This will allow the same constraint control mechanisms to be applied to the embedded generator.

The randomized instance should be a scenario descriptor

A scenario descriptor can be used to perform atomic generation by default. But it provides the necessary features to perform scenario generation if required by the user. Since scenario generation is the more general solution, it should be preferred to atomic generation.

The randomized instance shall contain a reference to the transaction it is generated with

Constraints can only be expressed within the scope of the object being randomized. It is desirable to be able to express constraints on the deviations to be introduced with respect to the transactions they are to be applied to. For example, it should be possible to express a constraint stating that a CRC error can only be injected on packets of length greater than 100 bytes. This can only be specified if the object being randomized has a reference to the transaction object.

For example:

```
class bu_eth_mac_deviations {
   bu_eth_mac_frame frame;
   ...
}
```

6

Transactors

Introduction

The term "transactor" is used to identify components of the verification environment that interface between two levels of abstractions for a particular protocol or generate protocol transactions. The lifetime of transactors is static to the verification environment: they get created at the beginning of the simulation and stay in existence for the entire duration. They are structural components of the verification components, similar to modules or entities in a design.

Traditional bus-functional models are called physical-level transactors. These transactors are the ones used in the command layer. Physical-level transactors have a transaction interface on one side and a physical interface on the other. Transactors in the functional layer only have transaction interfaces and do not directly interface to physical signals.

This chapter specifies guidelines designed to model transactors that are reusable, controllable and extendable.

Transactor Models

Transactors shall be modeled as a class

This allows an environment to instantiate as many copies of the transactors as required. The current state of each transactor is maintained in local properties and the execution threads are implemented in local methods.

Data and transactions are also modeled as objects. Why are transactors also objects? Why not include the transaction procedures in the data or transaction objects? The difference is in the respective lifetime of these two varieties of object.

Data and transactions are created as required to supply stimulus to the DUT. They flow through the verification environment and are transformed and recorded by the self-checking structure. They are eventually destroyed when their corresponding output transaction has been observed. Data and transaction objects contain methods to transform and operate on the data or transaction.

Transactors, on the other hand, are created at the beginning of the simulation, according to the needs of the verification environment. Their number remains constant throughout the duration of the simulation and they are destroyed only upon exit. They contain execution threads that can be started, stopped and reset. Data and transaction objects flow through transactors to and from the DUT. Transactor objects contain methods to transfer a data object to another transactor or the DUT or to execute a transaction.

Transactors shall be implemented in classes derived from the rvm_xactor

This base class contains standard properties and methods to configure and control transactors. To ensure that all transactors have a consistent usage model, they must be derived from a common base class.

All threads shall be started in the extension of the rvm_xactor::main_t() method

For the `rvm_xactor::start_xactor()` and `rvm_xactor::reset_xactor()` to work as intended, all threads that implement autonomous behavior for a transactor must be forked in the body of the `rvm_xactor::main_t()` task.

No threads shall be started in the constructor

This is a corollary of the previous guideline. Threads started in the constructor cannot be controlled by the `rvm_xactor::start_xactor()` and `rvm_xactor::reset_xactor()` methods.

Extensions of the rvm_xactor::main_t() method shall fork a call to super.main_t()

Transactors may be implemented as successive derived classes all based on the `rvm_xactor` class. Each inheritance layer may include relevant autonomous threads started in their extension of the `rvm_xactor::main_t()` method. The implementation of this method in the `rvm_xactor` base class is necessary for the proper operation of the control methods and must be eventually called.

```
class bu_eth_mii extends bu_xactor {
   ...
   virtual protected task main_t() {
      fork
         super.main_t();
      join none
      ...
   }
}
```

The rvm_xactor::start_xactor(), rvm_xactor::stop_xactor() and rvm_xactor::reset_xactor() may be extended to add protocol or transactor-specific functionality

These methods are virtual to enable the addition of functionality specific to the implementation of a transactor or a protocol to be executed when a transactor is started, stopped or reset.

Please refer to the documentation of the virtual task `reset_xactor(integer rst_type = 0)` on page 324 method for the interpretation of the reset type argument and what needs to be reset according to the specified type.

Extensions of the rvm_xactor::start_xactor(), rvm_xactor::stop_xactor() and rvm_xactor::reset_xactor() shall call their implementation in the base class using the super prefix

The implementation of a virtual method in a base class that has been overloaded in a derived class is only invoked when implicitly called using the `super` prefix. If a transactor extends these methods to perform transactor or protocol-specific operations, they must invoke the implementation of these virtual methods in the base class for proper operation.

Layers of a protocol shall be modeled as separate transactors

Protocols are often specified using a layering concept, each with different levels of abstraction. The transactors implementing these protocols should follow a similar division. The functional layer of the verification environment is built using sub-layers of relevant transactors. For example, a USB functional layer could be composed of USB transaction (host) and USB transfer (host controller) sub layers.

Transactors shall be identified (or configurable) as active, reactive or passive

Active transactors initiate transactions. Reactive transactors respond to transactions. A passive transactor will simply observe the interface in both directions, reporting observed data as it flows by and any protocol violation it observes. The verification environment must be able to control the timing of transactions initiated by active transactors but it has no control over the timing of transactions observed by reactive or passive transactors.

When modeling reactive and passive transactors, care must be taken that no data is lost if the transactor is executing user-extension code while a significant event occurs on the upstream interface.

For every active or reactive transactors, there should be a passive transactor

Active and reactive transactors are used when direct interaction with an interface is required to complete or initiate a transaction. When the design under verification is embedded into a system, that interface may no longer be controllable and instead controlled by another block in the system.

A passive transactor should be available to monitor the transactions that used to be under the control of the block-level environment to be able to reuse the block-level functional coverage model or self-checking structure.

All messages issued by a transactor objects shall use the message service instance in the rvm_xactor::log property

This will ensure that all messages from that transactor have a consistent format and can be controlled as a single set of messages.

Transactor objects should indicate events in the rvm_xactor::notify property upon occurrence of significant events

These notifiers can be used by the verification environment to synchronize with the occurrence of a significant event in a transactor. For more information on notifiers, refer to the section "Event Notification Class - rvm_notify" on page 333.

When relevant, status information about the reason of the event occurrence should be supplied by the transactor and attached to the event indication.

Transactors shall assign the value of their rvm_xactor::stream_id property to the rvm_data::stream_id property of the data objects flowing through it

The stream identifier is used to set the stream identifier in objects as they flow through the transactor or are randomized in the transactor and are reported to user-defined code extensions in callback methods. This identifier may be used to differentiate between multiple instantiations of the same transactor.

Transactors shall have a rich set of callback methods

The behavior of a transactor has to be controllable as required by the verification environment and individual testcases. These requirements are often unpredictable when the transactor is first written. By allowing the execution of arbitrary user-defined code in callback methods, transactors can be adapted to the needs of an environment or a testcase. For example, callback methods can be used to monitor the data flowing through a transactor to check for correctness, inject errors or collect functional coverage metrics.

The actual set of callback methods that must be provided by a transactor is protocol dependent. Subsequent guidelines will help design a suitable set in most cases. Additional callback methods should be provided as required by the protocol or the transactor implementation.

Transactors should call a callback method after receiving data, allowing the user to record, modify or drop the data

Whether it is a transaction descriptor or sampling a byte on a physical interface, the new input data must be reported to the user to be recorded in or checked against a scoreboard, modified to inject an error or collect functional coverage metrics.

Transactors should call a callback method before transmitting data, allowing the user to record, modify or drop the data

Whether it is a transaction descriptor or driving a byte on a physical interface, the new output data must be reported to the user to be recorded in or checked against a scoreboard, modified to inject an error or collect functional coverage metrics.

Transactors should call a callback method after generating any new information, allowing the user to record or modify the new information

Whenever a transaction requires locally generated additional information, the additional information must be reported to the user to be recorded in or checked against a scoreboard, modified to inject an error or collect functional coverage. A reference to the original transaction should be provided to convey context information.

For example, a transactor prepending a packet with a preamble should call a callback method with the generated preamble data before starting the transmission process.

Transactors should call a callback method after making a significant decision but before acting on it, allowing the user to modify the default decision

Whenever a transactor makes a choice between several alternatives, the choice and available alternatives must be reported to the user to be recorded in or checked against a scoreboard, modified to select another alternative or collect functional coverage.

For example, a transactor selecting traffic from different priority queues must call a callback method after selecting a queue based on the current priority selection algorithm but before pulling the next item from the selected queue. The user can then modify the selection.

All callback methods for a transactor shall be implemented as virtual tasks in a single class derived from rvm_xactor_callbacks

This creates a *facade* for all available callback methods for a particular transactor. These callback methods are extended using the Object-Oriented programming model. Restricting callback methods to tasks avoids difficulties with handling a return value from a function when multiple callback extensions are registered and cascaded in a transactor. If status information my be returned from the task (such as a flag to indicate whether to drop the transaction or not), use a `var` argument.

The common base class is required to be able to register the callback extension instances using the predefined methods and properties in the `rvm_xactor` class.

```
class bu_eth_mii_callbacks extends bu_xactor_callbacks {
   virtual task pre_frame_tx_t(...) {}
   virtual task post_frame_rx(...) {}
}
```

For each OOP callback method in the callback facade, there shall be an identical non-virtual protected method in the transactor class

This creates a set of callback methods that can only be extended using the Aspect-Oriented programming model. By having an AOP callback for each OOP callback, users can choose the appropriate mechanism for customizing the behavior of a transactor.

```
class bu_eth_mii extends bu_xactor {
   ...
   protected task pre_frame_tx_t(...) {}
   protected task post_frame_rx(...) {}
   ...
}
```

Note that the use of AOP callback extensions is restricted to implementing testcases, not verification environments.

The AOP callback method shall be called before the corresponding OOP callback method)

The OOP callbacks will be used to integrate the scoreboard and sample functional coverage. These features are part of the verification environment and thus must be implemented using the OO programming model. The AOP callback, which may be used to implement an error-injection testcase, may modify the content of the transaction. Any such modification must be recorded in the scoreboard and functional coverage model. Therefore, AOP callback methods must be called before OOP callback methods.

For example:

```
task bu_eth_mii::main_t() {
   ...
   while (1) {
      ...
      this.pre_frame_tx_t(...);
      rvm_OO_callback(bu_eth_mii_callbacks,
pre_frame_tx_t(...));
      ...
   }
}
```

Transactors shall be configured using a randomizable configuration object

The configuration of a transactor must be specified as an object. All the properties in the configuration object must be randomized to allow the generation of a random configuration – both to verify the transactor itself under different conditions and to make it usable as a component of the random test configuration descriptor.

Transactor configuration shall be passed via the constructor

A transactor must be configured before being used. The best way to ensure this is to require the configuration object to be provided as a constructor argument. The transactor may chose to keep a reference to the original configuration object or make a copy of it.

It should be possible to dynamically reconfigure a transactor

It should be possible to modify the configuration of a transactor during the simulation. Modifying the original configuration object may not be sufficient as the transactor has no means of efficiently detecting such a change, or it may have an internal copy different from the original.

A method accepting a new configuration description should be provided.

Reconfiguring a transactor that is not idle may yield unexpected results.

Physical-Level Interfaces

Command-level transactors and bus-functional models are components of the command layer. They translate transaction requests from the higher layers of the verification environment to physical-level signals of the DUT. In the opposite direction, they monitor the physical signals from the DUT or between two DUT modules. They also notify the higher layers of the verification environment of any transactions initiated by the DUT.

Virtual ports shall be declared in a VRI file

Virtual port declarations are not included in the header file generated from a source file, regardless if the `-h` or `-H` option is used. If the `-H` option is used to generate the header file, it will contain a suitable `#include` directive to make the virtual port declaration visible. If the `-h` option is used, it will be necessary to manually include this `vri` file or embed its content in the generated header file in a post-processing step. See “A top-level source file shall include all ancillary source files.” on page 10.

In `bu_eth_mii_port.vri`:

```
port bu_eth_mii_port {
   txd;
   tx_en;
   rxd;
   rx_dv;
   col;
   crs;
}
```

In `bu_eth_mii.vri`:

```
#include "bu_eth_mii_port.vri"
class bu_eth_mii extends bu_xactor {
   ...
}
```

Physical interfaces shall be specified using a virtual port binding as an argument to their constructor

This allows each instance of a transactor to be connected to specific interface signals, without requiring a particular naming convention or interfacing mechanism. The signal layer can choose to use static or dynamic signal connections to create a port bind.

```
#include "bu_eth_mii_port.vri"
class bu_eth_mii extends bu_xactor {
   ...
   task new(bu_eth_mii_port sigs, ...) {
      ...
   }
}
```

The virtual port binding shall be stored in a public property

This will allow testcases to access the physical signals if required.

```
class bu_eth_mii extends bu_xactor {
   bu_eth_mii_port sigs;
   ...
   task new(bu_eth_mii_port sigs, ...) {
      this.sigs = sigs;
      ...
   }
}
```

Virtual ports shall not include clock signals

OpenVera's physical interface model separates timing and synchronization from the behavior. The `interface` construct or the `signal_connect()` procedure define the timing and sampling relationships between data and clock signals. You can wait for the next edge of the clock by using an `expect` statement.

```
@1 this.sigs.data == void;
```

If the sampling clock signal is included in the virtual port, it then becomes possible – and tempting since it is the natural coding style in Verilog – to explicitly wait for a specific clock edge using the `@posedge` statement. This violates the separation of function and timing. It would also create a potential for functional failures if the signal layer binds data signals to interface signals synchronized with a different edge or clock signal. And these kinds of interface timing failures are quite difficult to debug.

A callback method should be called before driving values, allowing the values about to be driven to be modified

This will enable a user extension to introduce physical-level errors.

A callback method should be called after sampling values, allowing the sampled values to be modified

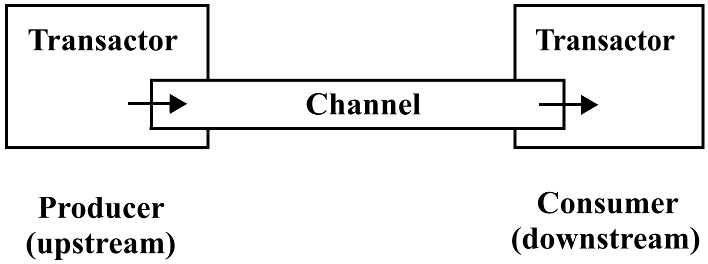
This will enable a user extension to emulate the effect of physical-level errors.

Transaction-Level Interfaces

This section applies to the transaction-level interface of transactors. The transaction-level interface allows the higher layers of the verification environment to have tests performed by specifying which transactions should be executed or be notified of which transactions have been observed at a given point in time. The transaction-level interface removes the higher-level layers from the physical interface details.

The transaction-level interface is an object designed to connect two transactors, one generating transactions for the other. The connection is established by having two transactors refer to the same transaction interface object, as illustrated in Figure 6-1. The connection can be made by instantiating any one of the endpoints in any order to allow the building of verification environments in a bottom-up or top-down fashion, or to replace the design with a transaction-level model. The transaction interface allows a transactor – whether upstream or downstream – to be connected to any other transactor with a compatible transaction-level interface, without requiring any source code modifications.

Figure 6-1 Transaction Interface Object



A transaction interface is implemented using the `rvm_channel` class. See "Transaction Interface Class - rvm_channel" on page 304 for more details.

An instance of a rvm_channel object shall be used to pass transactions between two transactors

Because transactions are modeled as objects, executing a transaction on a transactor or have a transactor report the occurrence of a transaction is accomplished by exchanging a reference to a transaction object instance.

This could be accomplished using a procedure. But invoking a procedure in an object instance requires having a reference to that object in the first place. This requires that verification environments be built in a bottom-up fashion, with the higher-layers having a reference to the lower-level transactor instances so they can call methods in them. This creates the following difficulties:

- You cannot build a verification environment on top of the physical layer using a procedural transaction-level interface that can then be retargeted, without modifications, to a different physical layer implementation.
- It is impossible to build a verification environment on a transaction-level model that can be reused, unmodified, on an implementation with physical-level interfaces.

By encapsulating the transaction exchange mechanism into an object, the transactors are considered as endpoints that can be replaced easily, without any knowledge required by or of the other endpoint.

A natural extension of the data notifier object is the possibility of locating the endpoints in different implementation languages. Thus a random generator or functional-level transactor written in OpenVera could exchange transactions with a physical-level transactor written in Verilog, a transaction-level model written in SystemC or a synthesized bus functional model running on an emulator – without requiring any modifications of the verification environment.

References to channel instances shall be stored in public properties suffixed with "_chan"

This allows the connection between two transactors to be made in arbitrary order. The first one creates the instance of the channel object then the second grabs a reference to the channel object in the first one.

Furthermore, this allows directed portions of tests to put manually created transaction objects into a channel by suspending the execution of the upstream transactors and accessing the channel’s `put_t()` method.

```
class bu_eth_mii extends bu_xactor {
   bu_eth_mac_frame_channel tx_chan;
   bu_eth_mac_frame_channel rx_chan;
   ...
}
```

Channel instances may be specified as constructor arguments

Connecting two transactors requires that they be endpoints on the same channel instance. One way would be to always allocate new instances in each transactor when they are constructed, then manually replace one of them with a reference to the other transactor's data notifier instance:

```
upstream   = new(...);
downstream = new(...);
upstream.data_out = low.data_in;
```

Alternatively, it should be possible to specify channel instances as optional constructor arguments. If none are specified, new instances are internally allocated.

```
upstream   = new(...);
downstream = new(..., high.data_out);
```

This requires that a channel object instance be allocated if none is specified w\in the constructor argument list.

```
class bu_eth_mii extends bu_xactor {
   bu_eth_mac_frame_channel tx_chan;
   bu_eth_mac_frame_channel rx_chan;
   ...
   task new(...
            bu_eth_mac_frame_channel tx_chan = null,
            bu_eth_mac_frame_channel rx_chan = null) {
      ...
      if (tx_chan == null) tx_chan = new(...);
      this.tx_chan = tx_chan;
      if (rx_chan == null) rx_chan = new(...);
      this.rx_chan = rx_chan;
   }
}
```

A transactor shall not hold an internal reference to a channel instance while it is stopped or reset

If a transactor holds a copy of the reference to a channel instance in an internal variable, the channel instance cannot be substituted with another one to dynamically redirect the output or input of a transactor. While unavoidable during normal operations, a reset or stopped transactor should "release" all such internal references to allow the channel instance to be replaced.

Active transactors shall have a mechanism to insert delays if the protocol allows it

If upstream transactors or testcases supply transactions as fast as downstream transactors can process them, this will only create tests with maximum transaction densities. For fixed-rate protocol, such as SONET, there is no concept of delays between transactions. But packet-based and bus-based protocols must be verified with different transaction densities.

An active transactor may have a method to hold the flow of transactions being processed and a method to restore the normal flow. To insert delays, a testcase would simply call the first method, wait for an appropriate amount of time or synchronization signal, and then call the second method. The transactor may provide delay method services to specify delays in terms of protocol events (e.g. number of clock cycles or data beats) rather than absolute simulation time.

It may also be possible to insert delays between or in the middle of a transaction by extending the appropriate callback methods with a blocking implementation. As long as the callback method is blocked, the execution of the transaction is delayed. Callback methods allowed to have a blocking implementation without breaking the protocol must be clearly identified.

Alternatively, the predefined `rvm_data::EXECUTE` event can be used to prevent the execution of a transaction until it is triggered ON.

Reactive and passive transactors shall allocate new transaction instance from a factory instance using the rvm_data::allocate() method

It is often desirable to add user-defined or testcase-specific information to a transaction object. This should be done via class inheritance, not in the original class, to avoid proliferating unrelated application-specific information into a generic definition – thus lowering its reusability. The only challenge is that these additional properties are located in a different object type.

This challenge is present in reactive and passive transactors that monitor and report transactions observed on a physical interface. Transaction instances are created internally when a new transaction is detected. Because these transactors are written in terms of the original base class, they will allocate an instance of the original, generic class without the required additional information if a call to `new` is used. This problem can be solved by creating the new instances by copying from a factory instance. The `rvm_data::allocate()` method, being virtual, will allocate an instance of the derived class found in the factory instance, not the original class.

Example: Reusable slave transactor:

```
class bu_ahb_slave {
   bu_ahb_transaction         factory;
   bu_ahb_transaction_channel tr_out;
   ...
      while (1) {
         bu_ahb_transaction tr;

         ...
         tr = this.factory.allocate();
         ...
         this.tr_out.put_t(tr);
      }
   ...
}
...
```

Example: Adding user-defined transaction information

```
class my_ahb_transaction extends bu_ahb_transaction {
   ...
}

program my_test {
   verif_env env = new(...);
   my_ahb_transaction my_tr = new(...);
```

```
      env.build();
      env.ahb_slave.factory = my_tr;

      env.run_t();
   }
```

See the “virtual function rvm_data allocate()” on page 344 for an implementation example.

A transactor shall not be both a producer and a consumer for a channel instance

Channels objects cannot enforce which transactor endpoint is the producer and which one is the consumer. Transaction objects "flow" from the `rvm_channel::put_t()` method to the `rvm_channel::get_t()` method. A producer is defined by the simple fact that it calls the `put_t()` method, whereas a consumer is defined by the fact that it calls the `get_t()` method. A transactor cannot be both a consumer and producer for the same channel (unless the channel is used internally and not as a transaction interface).

If a bidirectional interface is required, two channel instances must be used, one for each direction.

Output channels on reactive or passive transactors shall use the rvm_channel::sneak() method to put objects in the channel

A reactive or passive transactor will block on the execution of the `rvm_channel::put_t()` method if the channel is full. This may break the implementation of the protocol and cause data to be missed or checks not to be performed. Using the `rvm_channel::sneak()` method prevents these problems from occurring.

Because reactive and passive transactors are regulated by the interface (physical or transaction-level) they are monitoring, using the `rvm_channel::sneak()` method should not cause an infinite execution loop in the monitoring thread.

Completion and Response Models

Transactions are provided to and reported by transactors via a channel object. It is usually important for the higher layer transactors to know when a transaction has been completed or how to respond to a reactive transactor. Furthermore, it must be possible for an active or reactive transactor to output status information about the execution of the transaction.

The following guidelines will help choose a completion or response model suitable for the transactor and protocol being implemented. A completion model is used by active and reactive transactors to indicate the end of a transaction execution. A response model is used by a reactive transactor to request, from the higher layers of a verification environment, additional data or information required to complete a suitable response to the transaction being reacted to.

Transactors shall clearly document the completion model used by input channels

The completion model used by a transactor to indicate the completion of a transaction is crucial to its proper usage. Each transactors must document the completion model used.

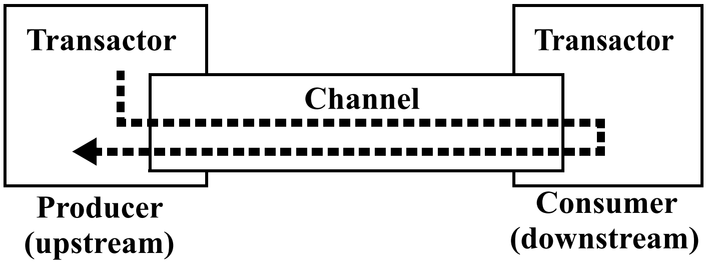
Reactive transactors shall clearly document the response model used by output channels

The response model used by a reactive transactor to request additional response information to complete a transaction is crucial to its proper usage. Each reactive transactors must document the response model used.

In-Order Atomic Execution Model

Transactors with an in-order atomic execution model perform transactions in the same order as they were submitted. Each transaction is executed only once and complete in a single execution attempt. Such transactors use a blocking completion model. As illustrated in Figure 6-2, the execution thread from the upstream transactor (depicted as a dotted line) is blocked while the transaction flows through the channel and is executed by the downstream transactor. It remains blocked until the execution of the transaction is completed.

Figure 6-2 Blocking Completion Model



From the upstream transactor's perspective, the blocking completion model is embodied in the `rvm_channel::put_t()` method. When this method returns, the transaction is completed.

```
class upstream_xactor extends bu_xactor {
   ...
   virtual tasl main_t() {
      ...
      while (1) {
         transaction tr;
         ...
         out_chan.put_t(tr);
         rvm_trace(this.log, "Transaction is completed");
         ...
      }
   }
}
```

The suitability and proper implementation of this completion model requires adherence to the following guidelines:

Input channel instances shall be reconfigured with a full level of 1

The channel object is responsible for blocking the execution of the `rvm_channel::put_t()` method, not the downstream transactor. That can only happen if the channel is considered full as soon as a transaction is put into the channel itself. Any other configuration would create a nonblocking interface.

To ensure that input channels have a full level of 1, downstream transactors must explicitly reconfigure the input channel instances. Otherwise, externally created instances with incompatible configurations may be used.

```
class downstream_xactor extends bu_xactor {
   transaction_channel in_chan;
   ...
  task new(..., transaction_channel in_chan = null, ...) {
      ...
      if (in_chan == null) in_chan = new(...);
      in_chan.reconfigure(1);
      this.in_chan = in_chan;
   }
}
```

Downstream transactors shall peek transactions out of the channel

To keep the `rvm_channel::put_t()` method blocked for the upstream transactor, the channel must not be emptied while the transaction is being executed. Therefore, the `rvm_channel::peek_t()` or `rvm_channel::activate_t()` method must be used to obtain the next transaction to be executed from the input channel.

```
class downstream_xactor extends bu_xactor {
   virtual task main_t() {
      ...
      while (1) {
         transaction tr = this.in_chan.peek_t();
         ...
         void = this.in_chan.get_t();
      }
   }
}
```

Downstream transactors shall get transactions out of the channel only when the transaction execution is completed

This is a corollary to the previous guideline. A transaction is removed from a channel by using the `rvm_channel::get_t()` or `rvm_channel::remove()` method.

Downstream transactors should indicate the rvm_data::STARTED and rvm_data::ENDED events

An upstream transactor may choose to use a nonblocking model by forking the thread that puts the transaction into the input channel. Providing built-in indication of the execution of the transaction will eliminate the need for additional synchronization infrastructure in the upstream transactor.

```
class downstream_xactor extends bu_xactor {
   virtual task main_t() {
      ...
```

```
        while (1) {
           transaction tr = this.in_chan.peek_t();
           tr.notify.indicate(tr.STARTED);
           ...
           tr.notify.indicate(tr.ENDED);
           void = this.in_chan.get_t();
        }
     }
  }
```

Downstream transactors should use the rvm_channel:: activate_t(), rvm_channel::start(), rvm_channel::complete() and rvm_channel::remove() methods to indicate the progress of the transaction execution

The `rvm_data::STARTED` and `rvm_data::ENDED` event require that the upstream transactor maintain a reference to the transactor descriptor while it flows through the channel and is executed by the downstream transactor. The active slot interface allows an upstream transactor to query the execution progress of a transaction directly from the channel itself.

```
class downstream_xactor extends bu_xactor {
   virtual task main_t() {
      ...
      while (1) {
         transaction tr = this.in_chan.activate_t();
         ...
         tr.notify.indicate(tr.STARTED);
         void = this.in_chan.start();
         ...
         tr.notify.indicate(tr.ENDED);
         void = this.in_chan.complete();
         void = this.in_chan.remove();
      }
   }
}
```

Downstream transactors may add completion status information to the transaction object

If the transaction object has properties that can be used to specify completion status information, these properties may be modified by the downstream transactor to provide status information back to the upstream transactor.

```
class downstream_xactor extends bu_xactor {
   virtual task main_t() {
      ...
      while (1) {
         transaction tr = this.in_chan.peek_t();
         tr.notify.indicate(tr.STARTED);
         ...
         tr.status = ...;
         tr.notify.indicate(tr.ENDED);
         void = this.in_chan.get_t();
      }
   }
}
```

Downstream transactors may attach completion status information to the rvm_data::ENDED event

If the transaction object does not have properties that can be used to specify completion status information, the downstream transactor can provide status information back to the upstream transactor via the `rvm_data::ENDED` event.

The additional status information is provided as a separate object, derived from `rvm_data`, and attached to the `rvm_data::ENDED` event when it is indicated. It is not necessary to overload all of the virtual methods in the status information class.

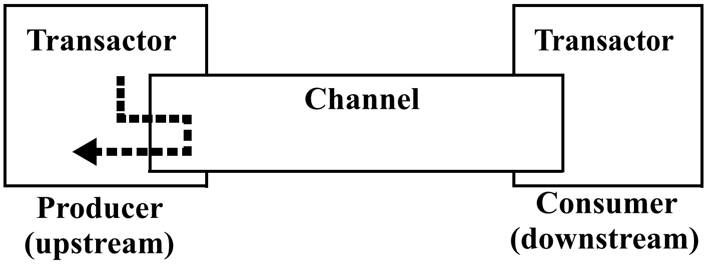
```
class transaction_status extends bu_data {
   ...
}

class downstream_xactor extends bu_xactor {
   virtual task main_t() {
      ...
      while (1) {
         transaction tr = this.in_chan.peek_t();
         tr.notify.indicate(tr.STARTED);
         ...
         {
            transaction_status status = new;
            ...
            tr.notify.indicate(tr.ENDED, status);
         }
         void = this.in_chan.get_t();
      }
   }
}
```

Out-of-Order Execution

Transactor with an out-of-order execution model execute transaction in a potentially different order than they were submitted. The order in which transaction are selected for execution is protocol specific and outside the scope of this document. Such transactors use a nonblocking completion model. As illustrated in Figure 6-3, the execution thread from the upstream transactor (depicted as a dotted line) is not blocked while the transaction flows through the channel and is executed by the downstream transactor. It is blocked only when the channel is full and unblocks as soon as it is empty, regardless of the completion of the transaction.

Figure 6-3 Nonblocking Completion Model



The nonblocking completion model allows several transactions to be submitted to the downstream transactor to be completed in the future. It is up to the upstream transactor to detect the completion of a transaction according to a mechanism defined by the downstream transactor.

The suitability and proper implementation of this completion model requires adhesion to the following guidelines:

Input channel instances should be reconfigured with a full level greater than 1

The channel object is responsible for blocking the execution of the `rvm_channel::put_t()` method, not the downstream transactor. That only happens if the channel is considered full. More than one transaction must be available in the channel to allow out-of-order execution to occur. If a full level of 1 is used, a blocking interface is created and out-of-order execution is only possible if the downstream transactor implements additional transaction buffering internally.

Transactors shall use the rvm_channel::activate_t(), rvm_channel::start(), rvm_channel::complete() and rvm_channel::remove() methods to indicate the progress of the transaction execution

The `rvm_data::STARTED` and `rvm_data::ENDED` event require that the upstream transactor maintain a reference to the transactor descriptor while it flows through the channel and is executed by the downstream transactor. With an out-of-order execution model, it is a complex task to manage these references to all pending transactions and identify the next one that will be executed. The active slot interface allows an upstream transactor to query the execution progress of a transaction directly from the channel itself.

```
class downstream_xactor extends bu_xactor {
   virtual task main_t() {
      ...
      while (1) {
         ...
         tr = this.in_chan.activate_t(i);
         ...
         void = this.in_chan.start();
         ...
         void = this.in_chan.complete();
         void = this.in_chan.remove();
      }
   }
}
```

Downstream transactors should indicate the rvm_data::STARTED and rvm_data::ENDED events

An upstream transactor may track individual transaction by maintaining a reference to the transaction descriptors as they flow through the channel and are executed by the downstream transactor. Using the built-in indication of the execution of the transaction will eliminate the need for additional synchronization infrastructure in the upstream transactor.

```
class downstream_xactor extends bu_xactor {
   virtual task main_t() {
      ...
      while (1) {
         ...
         tr = this.in_chan.activate_t(i);
         ...
         tr.notify.indicate(tr.STARTED);
         void = this.in_chan.start();
         ...
         tr.notify.indicate(tr.ENDED);
         void = this.in_chan.complete();
         void = this.in_chan.remove();
      }
   }
}
```

Downstream transactors may add completion status information to the transaction object

If the transaction object has properties that can be used to specify completion status information, these properties may be modified by the downstream transactor to provide status information back to the upstream transactor.

```
class downstream_xactor extends bu_xactor {
   virtual task main_t() {
      ...
      while (1) {
         ...
         void = this.chan.start();
         ...
         tr.status = ...;
         tr.notify.indicate(tr.ENDED);
         void = this.in_chan.complete();
         ...
      }
   }
}
```

Downstream transactors may attach completion status information to the rvm_data::ENDED event

If the transaction object does not have properties that can be used to specify completion status information, the downstream transactor can provide status information back to the upstream transactor via the `rvm_data::ENDED` event.

The additional status information is provided as a separate object, derived from `rvm_data`, and attached to the `rvm_data::ENDED` event when it is indicated. It is not necessary to overload all of the virtual methods in the status information class.

```
class transaction_status extends bu_data {
   ...
}

class downstream_xactor extends bu_xactor {
   virtual task main_t() {
      ...
      while (1) {
         ...
         tr.notify.indicate(tr.STARTED);
         void = this.in_chan.start();
         ...
         {
            transaction_status status = new;
            ...
            tr.notify.indicate(tr.ENDED, status);
         }
         void = this.in_chan.complete();
         ...
      }
   }
}
```

Concurrent, Split or Recurring Transaction Execution

Non atomic transactors execute transactions in parallel, through multiple attempts, or multiple partial sub transactions or execute a transaction repeatedly at regular intervals. Such transactors use a nonblocking completion model. As illustrated in Figure 6-3, the execution thread from the upstream transactor (depicted as a dotted line) is not blocked while the transaction flows through the channel and is executed by the downstream transactor. It is blocked only when the channel is full and unblocks as soon as it is empty, regardless of the completion of the transaction.

The nonblocking completion model allows several transactions to be submitted to the downstream transactor to be completed in the future. It is up to the upstream transactor to detect the completion of a transaction according to a mechanism defined by the downstream transactor.

The suitability and proper implementation of this completion model requires adherence to the following guidelines:

Input channel instances should be reconfigured with a full level greater than 1

The channel object is responsible for blocking the execution of the `rvm_channel::put_t()` method, not the downstream transactor. That only happens if the channel is considered full. More than one transaction must be available in the channel to allow out-of-order execution to occur. If a full level of 1 is used, a blocking interface is created and out-of-order execution is only possible if the downstream transactor implements additional transaction buffering internally.

Transactors shall only use the rvm_channel::get_t() to immediately remove a transaction from the channel

Transactions in the channel are assumed to be available for execution. As soon as a transaction is selected for execution (either concurrently, partially or as the first instance of a recurrence), it must be immediately removed from the channel to prevent it from being selected again.

```
class downstream_xactor extends bu_xactor {
   virtual task main_t() {
      ...
      while (1) {
         ...
         tr = this.in_chan.get_t(i);
         ...
      }
   }
}
```

Downstream transactors should indicate the rvm_data::STARTED and rvm_data::ENDED events

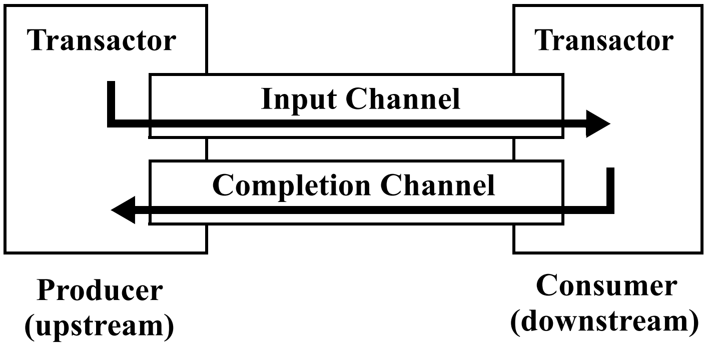
An upstream transactor may track individual transaction by maintaining a reference to the transaction descriptors as they flow through the channel and are executed by the downstream transactor. Using the built-in indication of the execution of the transaction will eliminate the need for additional synchronization infrastructure in the upstream transactor.

```
class downstream_xactor extends bu_xactor {
   virtual task main_t() {
      ...
      while (1) {
         ...
         tr = this.in_chan.get_t(i);
         tr.notify.indicate(tr.STARTED);
         ...
         tr.notify.indicate(tr.ENDED);
      }
   }
}
```

An output "completion" channel should be used to send back (partially) completed transactions

An upstream transactor may require information about the various intermediate completions of a transaction execution -- each execution attempts, each sub-transactions, each occurrence of a recurring transaction. Since a transaction may have more than one completion indication, an output channel should be used to return completion information back to the upstream transactor, as illustrated in Figure 6-4.

Figure 6-4 Completion Channel



```
class downstream_xactor extends bu_xactor {
   transaction_channel in_chan;
   transaction_channel compl_chan;

   virtual task main_t() {
      ...
      while (1) {
         ...
         tr = this.in_chan.get_t(i);
         tr.notify.indicate(tr.STARTED);
         ...
         tr.notify.indicate(tr.ENDED);
         this.compl_chan.sneak(tr);
      }
   }
}
```

A copy of the original transaction descriptor should be sent back through the completion channel

A single transaction descriptor may result in multiple completion responses back through the completion channel. If the same instance is used, subsequent responses may modify the content of prior responses before the upstream transactor has had time to process them. Using separate instances for each response will ensure that an accurate history of the transaction execution will be reported via the completion channel.

```
class downstream_xactor extends bu_xactor {
   transaction_channel in_chan;
   transaction_channel compl_chan;

   virtual task main_t() {
      ...
      while (1) {
         ...
         tr = this.in_chan.get_t(i);
         tr.notify.indicate(tr.STARTED);
         ...
         tr.notify.indicate(tr.ENDED);
         this.compl_chan.sneak(tr.copy());
      }
   }
}
```

Downstream transactors shall use the rvm_channel::sneak() method to add completed transaction descriptors to the completion channel

This will avoid the downstream transactor from stalling on a full completion channel, should the upstream transactor fail to drain it. No data is loss even if the channel becomes full.

Upstream transactors should drain completion channels

This will avoid the downstream transactor from stalling on a full completion channel, and prevent the accumulation of data in the completion channel.

Downstream transactors may add completion status information to the transaction object

If the transaction object has properties that can be used to specify completion status information, these properties may be modified by the downstream transactor to provide status information back to the upstream transactor.

```
class downstream_xactor extends bu_xactor {
   virtual task main_t() {
      ...
      while (1) {
         ...
         void = this.chan.start();
         ...
         cast_assign(compl, tr.copy());
         compl.status = ...;
         this.compl_chan.sneak(compl);
         ...
      }
   }
}
```

Downstream transactors may use a different descriptor to return transaction completion information

If the transaction object does not have properties that can be used to specify completion status information, the downstream transactor can provide status information back to the upstream transactor via a different object supplied through the completion channel.

The additional status information is provided as a separate object, derived from `rvm_data`. A reference to the original transaction should be provided in the completion object. It is not necessary to overload all of the virtual methods in the status information class.

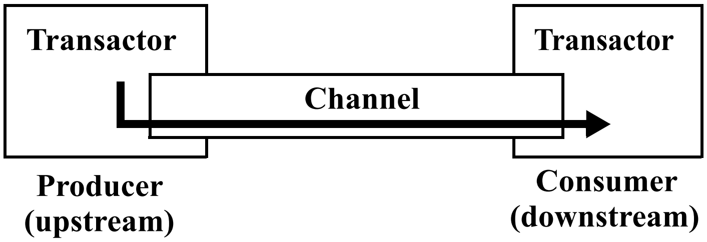
```
class transaction_compl extends bu_data {
   ...
}

class downstream_xactor extends bu_xactor {
   virtual task main_t() {
      ...
      while (1) {
         ...
         tr.notify.indicate(tr.STARTED);
         void = this.in_chan.start();
         ...
         {
            transaction_compl compl = new;
            ...
            compl.tr = tr;
            this.compl_chan.sneak(compl);
         }
         ...
      }
   }
}
```

Passive Response

Passive transactors monitor the transactions executed on a lower-level interface and report to the higher layers descriptions of the observed transactions. The passive transactor should report any protocol-level errors it detects but the higher level transactors will be responsible for checking the correctness of the data carried by the protocol. As illustrated in Figure 6-5, passive transactors use an output channel to report transactions. Each observed transaction is reported using a new instance of the transaction descriptor object.

Figure 6-5 Passive Response Model



Note that the passive response model is not limited to passive transactors. It can be used to report on observed transactions in any other transactors. A reactive transactors may use the passive response model to report on the observed transactions that were actively replied to. An active transactor may use a passive response model to report on the received transactions as observed on a half-duplex interface.

The suitability and proper implementation of this response model requires adhesion to the following guidelines:

The passive transactor shall put transaction descriptor instances in the output channel using the rvm_channel::sneak() method

The channel will block the execution thread of the passive transactor if it ever becomes full. This may break its implementation or cause data to be lost. The `rvm_channel::sneak()` method ignores the channel’s full level and never blocks the execution thread of the upstream transactor. Because the passive monitor is observing the proper execution of a protocol, its execution should be regulated by the time required to execute a complete transaction.

Upstream transactors should put an incomplete transaction descriptor instance in the output channel as soon as the start of a transaction has been identified

A downstream transactor may need to know when a transaction has started executed on an interface. For example, a half-duplex higher-level transactor would need to know if the transport medium is busy before attempting to execute its own transaction. Waiting until the end of the transaction to put it in the output channel may make the information available only too late.

```
class passive_xactor extends bu_xactor {
   virtual task main_t() {
      ...
      while (1) {
         ...
         tr = new;
         this.out_chan.sneak(tr);
         tr.notify.indicate(tr.STARTED);
         ...
         tr.notify.indicate(tr.ENDED);
      }
   }
}
```

Passive transactors should indicate the rvm_data::STARTED and rvm_data::ENDED events

This is a requirement of the previous guideline. If an incomplete transaction descriptor instance is put into the channel, the higher-level transactor on the other side of the channel will need to know when the transaction has been completed. Using the built-in transaction completion notification events eliminates the need for additional synchronization infrastructure or mechanisms.

Passive transactors shall add completion status information to the transaction object

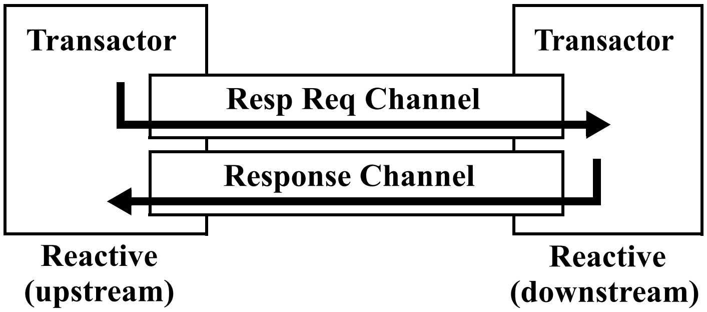
The transaction must have the necessary properties to indicate to the downstream transactors how the transaction was completed.

```
class downstream_xactor extends bu_xactor {
   virtual task main_t() {
      ...
      while (1) {
         ...
         tr = new;
         this.out_chan.sneak(tr);
         tr.notify.indicate(tr.STARTED);
         ...
         tr.status = ...;
         tr.notify.indicate(tr.ENDED);
      }
   }
}
```

Reactive Response

Reactive transactors monitor the transactions executed on a lower-level interface and may have to request additional data or information from higher layers transactors to complete the transaction. The reactive transactor should report any protocol-level errors it detects and locally generate protocol-level answers. But the higher level transactors will be responsible for providing correct data content to be carried by the protocol. As illustrated in Figure 6-6, reactive transactors use output channels to request transaction responses. A second input channel is used to receive the transaction response to be applied to the lower-level interface. Each response request is reported using a new instance of a transaction response descriptor object.

Figure 6-6 Reactive Response Model



Note that the reactive response model is only used to obtain higher-level data carried by the protocol. Where the entire set of possible responses are fully defined by the protocol, the response is internally generated by the reactive transactor. For example, deciding to reply to a USB transaction with a ACK, NACK or a STALL packet (or not replying at all) can be entirely decided internally. However, the content and length of a DATA packet in reply to an IN transaction should be provided by a reactive response model. Note that the response must be provided within sufficient time to avoid breaking the protocol.

The suitability and proper implementation of this response model requires adhesion to the following guidelines:

The downstream reactive transactor shall use a nonblocking model for the response request channel

The implementation of the protocol may require that the reactive transactor perform additional operations while the response is being "composed". Using a nonblocking model, the downstream transactor is free to continue its execution thread, if only to immediately wait for a response via the response channel.

```
class reactive_xactor extends bu_xactor {
   ...
   virtual protected task main_t() {
      ...
      while (1) {
         ...
         resp_req = new;
         ...
         this.resp_req_chan.sneak(resp_req);
         resp = this.resp_chan.get_t();
         ...
      }
   }
}
```

Upstream reactive transactors shall check that a response is provided within required time interval.

The time required to respond to a transaction is usually limited by the lower-level protocol specification. However, the time required to "compose" the response is controlled by the downstream reactive transactor. Thus the upstream reactive transactor can only check that the response comes back when required.

```
class reactive_xactor extends bu_xactor {
   virtual task main_t() {
      ...
      while (1) {
         ...
         resp_req = new;
         ...
         this.resp_req_chan.sneak(resp_req);
         fork
            resp = this.resp_chan.get_t();
            {
                ...
           rvm_warning(this.log, "No response in time");
            }
         join any
         terminate;
         ...
      }
   }
}
```

Upstream reactive transactors should issue a warning message if no response is received after the maximum allowable time interval

The higher-level transactor may have wished to inject this particular error or the consequences of not continuing with the transaction response may be irrelevant for the device under verification. Nonetheless, a message should be issued to inform the unwary user of a potential problem with the verification environment.

Transaction response request objects should provide a random response when randomized

This simplifies the composition of a default answer when the higher-level data is carried by the protocol. A downstream reactive transactor can provide a random – but valid – response by simply randomizing the response descriptor.

```
class downstream_xactor extends bu_xactor {
   virtual task main_t() {
      ...
      while (1) {
         req = this.resp_req_chan.get_t();
         req.stream_id = this.stream_id;
         req.object_id = this.response_id++;
         void = req.randomize();
         this.resp_chan.put_t(req);
      }
   }
}
```

Protocol-level response shall be randomly generated using an embedded generator

Protocol-level response are fully defined by the protocol and can be selected by the upstream reactive transactor without any input required from the higher level transactors. The response should be generated using an embedded generator. By default, the generator is constrained to produce the best possible response but it can be unconstrained or modified to respond differently or inject errors.

See “Embedded Generators” on page 92 for more details on embedded generators.

Upstream reactive transactors may randomly generate a default response using an embedded generator

To ease the creation of verification environment, a reactive transactor may be configurable to generate the complete protocol response internal instead of deferring the higher-level data to higher-level reactive transactors. A transactor that detects that a response was not provided within acceptable time and determines that the response request is still in the request channel could assume that there are no higher level transactor and choose to compose a default response on its own. The response should be generated using an embedded generator. By default, the generator is constrained to produce adequate responses but it can be unconstrained or modified to respond differently or inject errors.

See “Embedded Generators” on page 92 for more details on embedded generators.

7

Design for Verification (DFV)

Introduction

Design for Verification (DFV) is an integral part of the overall RVM. It is the rational response to the problems encountered when verifying the current (and future) complex microelectronics devices. Conceptually, it follows a similar path as the changes that preceded it, namely, Design For Synthesis (DFS) and Design For Test (DFT).

When the design of chips reached a point where schematics-based methodology was becoming a hindrance to designing larger devices as permitted by the advances in technology, HDL-based DFS methodology was introduced that included the following key elements: Higher abstraction level (RTL), restrictions on the allowed HDL constructs supported by verifiable rules, and a set of tools supporting the methodology – RTL-to-logic synthesis, rule checking, RTL simulation.

The rapid increase in the size and complexity of the so designed devices made them more difficult to test after manufacture. Functional tests derived from design verification sequences were not sufficiently effective in detecting the structural defects in the devices. DFT and later Built-In-SelfTest (BIST) separated structural defect testing from the functional aspects of the devices. Again, abstracting from the function and also from the structure (random tests), improved controllability and observability lead to enormous gains. The DFT methodology imposed restrictions on the forms of RTL and the registers implementing memory elements, and is supported by verifiable rules and CAD tools.

Design verification is now in a similar situation in which methodological changes supported by verifiable rules and CAD tools must come to the rescue. The RVM methodology is based on the following cornerstones:

- Constrained random stimulus generation raises the abstraction level in testbench code.
- DFV improves the observability of design errors by providing means for an abstract specification of the device behavior using assertions.
- Reusable testbenches and reusable assertion-based checkers reduce the effort needed to verify complex devices and interface protocols.

- Supporting language and software tools: the Hardware Verification Language OpenVera that supports abstract data types and constrained random simulation, the assertion language OpenVera Assertions (OVA), the fast integrated coverage-driven simulation of design, testbench and assertions supported by VCS and Vera, and the efficient block-level hybrid-formal verification in Magellan for assertion proofs and coverage-driven automatic test sequence generation. All of these tools are integrated in the Discovery ™ Verification Platform.

In this chapter, we concentrate on Design for Verification. The chapter covers the following aspects of the methodology:

- How to use assertions in a design: assertions on internal signals and on interfaces; use of prepackaged checkers such as the OVA standard checker library.
- OVA coding guidelines in general and some specific rules for use with (hybrid-)formal tools. The coding guidelines provide hints on how to model certain behaviors using assertions.
- Guidelines for developing reusable assertion-based checkers.
- Guidelines on using assertions in OpenVera testbenches.
- Information on what constitutes a reusable assertion-based checker IP. That is, what code and documentation should be provided with such objects.
- Guidelines on using OVA assertions in OpenVera testbenches as sampling events and coverage points.

Using assertions with a design

The three main sources of functional flaws in taped-out designs are design errors, specification errors and errors in re-used modules and IP (either internal errors or incorrect usage). Most of these errors are due to ambiguous or changing specifications, and due to unwritten or unverified assumptions on the behavior of surrounding blocks.

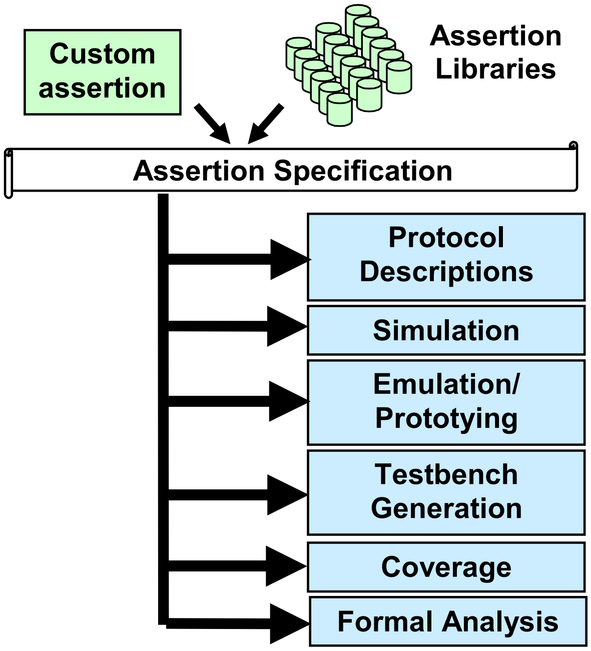
Assertion languages like OVA provide an efficient means for system architects to complement design specifications by non-ambiguous and executable statements in the form of assertions that precisely express the intent of the specification. Such assertions reduce ambiguity and thus the chance of misinterpretation. Since assertions are a different, more abstract description of the required behavior than the design RTL, they increase the likelihood of detecting a design error during simulation or emulation. Moreover, formal and hybrid (a combination of formal engines and simulation) tools can prove that the design does not violate the assertions under any legal input stimulus.

When creating various blocks in the RTL model of the design, the designers often make assumptions on the behavior of the surrounding blocks. These assumptions are usually unwritten and not verified during simulation. Any change in the behavior of the surrounding blocks may violate these assumptions, leading to errors that may cause failures only much farther in the information flow through the design. This makes the detection and the identification of the source of failure much more difficult and time consuming. Assertions that express the interface assumptions can detect such violations right at the source, making debugging of the design that much easier.

A similar situation but even more dangerous exists when re-using an existing module or IP block. If the assumptions on the usage of the module are not precisely stated and verified, errors may be difficult to identify due to the black-box nature of such imported modules or even undetected if that module is not properly exercised during verification. Assertions play an important role here in specifying the rules under which the module can be used (the assumptions on the use environment) and the expected behavior of the module. Finally, assertions can describe the input sequences that exercise the range of behaviors embodied by the IP block and that should be covered during verification. In the case of standard protocols, their specifications often include such “compliance statements”.

It follows from the above discussion that assertions can influence many parts of the verification process, as shown in Figure 7-1.

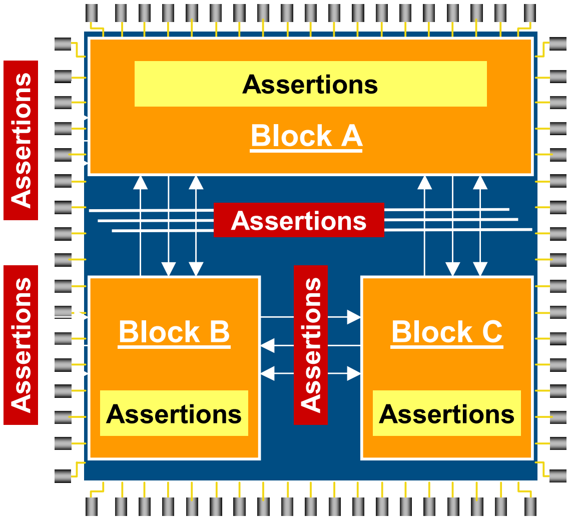
Figure 7-1 Assertions are a central part of verification



The usage of assertions can be divided into the following categories (Figure 7-2):

- Assertions on internal signals of the design, including inter-block interfaces.
- Assertions on external interfaces, standard or custom.
- Coverage statements.

Figure 7-2 Assertions on internal and external interfaces and internal signals of the design - block to system-level verification.



In addition, assertions for typical behaviors are packaged for reuse in a standard checker library and assertion for standard protocols such as PCI, PCI Express, AMBA AHB, Utopia, etc. can be packaged as reusable checker IP units. Both kinds can be used in either of the above contexts, although checker libraries usually cover more low-level behaviors suitable for checking local behaviors inside a design, while the larger checker IP is for complete checks of standard protocol on interfaces (external or internal).

As mentioned, assertions play a vital role when verifying a design block using formal or hybrid-formal tools. There, assertions for interface protocols can play a dual role: A check on the behavior of the block and assumptions (or constraints) on the behavior of the surrounding environment. Therefore, such assertions must be subdivided into disjoint subsets, one for each signal direction on the interface. We discuss such a subdivision in the section on reusable checkers.

A question arises: When to choose an assertion-based checker and when to use a testbench monitor? There is no definite answer, the choice depends on many factors:

- If the behavior involves transactions consisting of large data blocks and the objective is to verify their correct routing or numerical transformations, a testbench monitor may be more appropriate.
- If the property to verify involves individual signals inside the design, assertions are more appropriate because they can more easily access such signals than a testbench.
- If the behavior to verify is an interface protocol it is more appropriate to write the monitor using assertions.
- If the behavior is to be verified with the help of (hybrid-) formal tools, then assertions should be used.
- A monitor may consist of both assertions on the lower-level signaling protocol and testbench code for the higher-level, transaction-oriented, properties.

The following sections describe rules governing the three categories of assertions, including some best-practice coding guidelines. This is followed by rules that are specific to reusable checkers and their packaging. A discussion on testing assertions and assertion-based checkers concludes this chapter.

Assertions on internal design signals

Assertions on internal signals are inserted by the designer when advancing in the detailed implementation of the specification. The assertions serve as succinct comments describing the expected behavior while at the same time, during simulation or formal analysis, they check that the actual behavior is as intended. The following guidelines should be followed (in addition to the best practice coding guidelines in Section “Assertion coding guidelines” on page 164.)

Assertions on internal signals shall be implemented using inlined OVA

Since these assertions also serve as part of the code documentation and they are inserted by the designer, the assertion statements shall appear in the RTL code where they apply. Hence, the following OVA pragmas shall be used as specified in the VCS User Guide:

```
/* ova first_part_of_pragma
...
last_part_of_pragma
*/
```

or

```
//ova_begin
// pragma_statement
//...
//ova_end
```

or for a single-line statement

```
// ova pragma_statement;
```

Standard OVA checkers shall be used wherever possible

Using predefined checkers has a number of advantages as compared to writing custom assertions:

- To reduce the amount of coding.

- Custom checkers may not be easy to write and debug by an RTL designer.
- The standard checkers have been thoroughly verified

As long as there is one matching the required behavior, checkers provide a quick way to add assertions to the design.

There are some 80 checkers in the OVA standard checker library, located in $VERA_HOME/ova/etc/ov. The library includes checkers that are the OVA equivalents to the well known Accellera Open Verification Library (OVL); the names of these checkers are prefixed by “assert_”.

The OVA checkers characterize behaviors from very simple boolean invariant properties like `ova_check_bool` or `assert_always` to medium complexity time window and handshake checks like `window` or `ova_valid_id`, and finally there are checkers that verify typical hardware blocks such as fifo, stack, memory and arbiter. For more detail please refer to the quick reference list in “OVA Checker Library Quick Reference” and to the *OpenVera Assertions Checker Library Reference Manual*.

Even if the desired behavior is not covered by a single checker in the library, it is often possible to decompose the required behavior into a collection of properties that together imply the behavior. Each such property being implantable using a standard checker.

In the case of simple standard checkers like `ova_check_bool`, a single-line `//ova` pragma is all it takes to insert it, while the more complex checkers that have many parameters may be best inserted using the other two forms of pragmas. If local rules require uniform pragma starts using // form of comments, then the `//ova_begin` and `//ova_end` delimiters should be used.

Every FSM shall have assertions that verify the state encoding and transitions

The checks shall include

- invalid states are not reached
- reset state is correct

and may include

- valid transitions (present state – valid next states)
- correct outputs on transitions

There are a number of OVA standard checkers that are suitable for verifying FSM behavior:

```
ova_next_state
ova_value
ova_sequence
ova_one_cold
ova_one_hot
ova_code_distance
assert_next
assert_transition
assert_cycle_sequence
assert_one_hot
assert_one_cold
assert_zero_one_hot
```

Every internal block interface shall have assertions that verify the assumed interface protocol

Unverified assumptions used in the design of an interface controller are often a source of errors if the neighboring block violates these assumptions. Therefore, each interface should have assertions on the complete protocol. There are numerous OVA standard checkers available that can be used to build the appropriate assertions for an interface protocol. For example,

```
ova_hold
ova_hold_value
ova_req_ack_unique, ova_req_resp
ova_valid_id
ova_reg_loaded, ova_req_requires
ova_window
ova_even_parity
ova_odd_parity
ova_driven
ova_no_contention
ova_tri_state
assert_frame
assert_handshake
assert_win_change
assert_win_unchange
assert_window
assert_even_parity
assert_odd_parity
```

Every fifo, stack or memory shall have assertions on its proper use

Incorrect use of fifo and stack (overflow, underflow, data corruption) as well as memory accesses (overwriting without reading, reading before writing, and so on) are common kinds of errors. Each such block should have assertions verifying its use protocol. There are numerous checkers for these structures in the OVA standard checker library:

```
ova_fifo
ova_dual_clk_fifo
ova_multiport_fifo
ova_stack
ova_memory
ova_memory_async
assert_fifo_index
```

Assertions shall be used to verify that arbitration for access to resources is following the appropriate rules

Errors in arbitration may lead to starvation of some requestors. Assertions are that verify rules of the specific arbitration algorithm can be very effective in detecting such problems. The OVA standard checker library contains the unit

```
ova_arbiter
```

that is specifically designed to verify different arbitration algorithms: *priority* alone or with (as secondary selection criterion) *round-robin* (fairness), *fifo* or *least-recently-used* (LRU).

There shall be no assertion on the system clock only

The assertion language is primarily targeting synchronous systems where all signals are updated synchronously with some clock. An assertion verifying such a clock would have to run on simulation time as its sampling clock which is inefficient. Also, such assertions cannot be used with formal or hybrid-formal tools.

There shall be no assertion to monitor combinational signal glitches or asynchronous timing

The assertion language is primarily targeting synchronous systems where all signals are updated synchronously with some clock. Therefore, to state assertions over asynchronous signals would require using simulation time as the sampling clock of the assertion. This is inefficient and also cannot be used with formal or hybrid-formal tools.

There shall be no assertion that verifies the correctness of the HDL language or on know-to-be-good components

For example, verifying that a sequence of arithmetic operations produces a correct sum is not useful:

```
i = a + b;
j <= i + 1;

and the assertion
      event e: if (!reset) then j == past(a) + past(b);
```

Assertions shall verify that arithmetic operations do not overflow and / or the target registers do not change value by more than some +/- delta

In the case where wrapping over is not permitted, overflow, underflow and too large changes in value are often sources of errors. The following OVA standard checker library units may help formulating those checks:

```
ova_arith_overflow
ova_overflow
ova_underflow
ova_dec
ova_inc
ova_delta
ova_range
assert_increment
assert_decrement
assert_no_overflow
assert_no_underflow
assert_range
```

Decoding / selector logic shall have assertions to verify mutual exclusion

Violation of mutual exclusion may lead to contention on busses and other interference between blocks targeted by the selection / decode logic. The following OVA standard checkers may be useful to verify such logic:

```
ova_bits
ova_code_distance
ova_one_hot, ova_one_cold
ova_mutex
ova_asserted
ova_deasserted
ova_no_contention
ova_value
assert_one_hot
assert_one_cold
assert_zero_one_hot
```

Whenever a signal is to hold for some time or until some condition occurs, such behavior shall be verified using assertions

For example, “once asserted, *s* must remain so for 3 clock cycles” and “once asserted, *load* must remain so until *eop* is asserted” are situations targeted by this rule. The OVA standard checker library contains a number of units that can be used in this context:

```
ova_asserted
ova_deasserted
ova_hold
ova_hold_value
ova_reg_loaded
ova_req_resp
ova_timeout
ova_window
assert_time
assert_unchange
assert_width
assert_win_change
assert_win_unchange
assert_window
```

Any time-bounded well defined relationship between signals should be checked using assertions

Such assertions should verify the cause-effect relationships of such signals. For example,

- *data_valid_in* on one interface causes within a certain amount of time *data_valid_out* on another interface, or
- signal *a* asserted implies that condition *c* holds within 1 to 3 clock cycles.

There are many OVA standard checkers that can be used to verify such relationships. For example:

```
ova_check_bool
ova_follows
ova_forbid_bool, ova_const
ova_req_resp
ova_req_requires
assert_always
assert_always_on_edge
assert_change
assert_frame
assert_implication
assert_never
```

Reset conditions on control signals and shared busses following a reset shall be verified using assertions

This assumes resets that span at least one clock cycle and can thus be sampled by the assertion. For example, such checks may verify that a reset lasts at least some number of cycles, that bus drivers are tri-stated (verifying that a bus carries the value Z may only be verified in simulation) or equivalently that enable signals on bus drivers are de-asserted. The latter requires that these internal signals to the DUV are accessible. The OVA standard checkers mentioned in the preceding rule “Any time-bounded well defined relationship between signals should be checked using assertions” on page 156 are likely to be useful in this context.

No assertion shall be used to verify the behavior of known-to-be-good components

In particular, this includes memorization in synchronous latches or flip-flops.

Assertions on external interfaces

Unless inserted inside the DUV as inlined assertions by the designer as discussed in the preceding section, the assertions on external interfaces are provided and used by verification engineers. The design under test is then considered as a black box. Exception may be the access to enable signals of shared bus drivers, as explained in the rules governing reusable assertion-based checkers “Reusable OVA Checkers” on page 186.

Assertions on interfaces packaged in one or more OVA units shall be attached to the DUV using the OVA bind module or bind instances statements and ports will be mapped to module ports

Since the DUV is treated as a black box, internal signals are not visible and no code shall be inserted in the design by verification engineers. The OVA bind statement provides non-intrusive means for attaching checkers without any design code modification.

Assertions shall be divided into two categories - assertions on local interface protocols and assertions on signals from two or more interfaces

The first category is concerned with local behavior of the interface protocol. It makes sure that communication follows the specified rules. Often these protocol assertions are suitable for verification by formal or hybrid-formal tools.

The second category is concerned with the function of the DUV as it relates some conditions on one interface with conditions on another interface in a causal way. Such assertions may cover a very small portion of the behavior and be amenable to formal proof in formal or hybrid-formal proofs, but more often the portion of the design determining the behavior to be verified by the assertion is too big for the formal tool. The assertions covering the global behavior may be suitable only for simulation or for searching for bugs using hybrid-formal techniques.

Custom assertions verifying local protocols on interfaces shall follow rules for constructing reusable assertion-based checkers

Even if the interface protocol is not standard and may not be reused on another design, the reuse rules assure that the checker can be used both for block-level verification with (hybrid-)formal tools (some assertions become assumptions on the environment) and at block or system-level simulation (all interface assertions verify the communication on interconnections between blocks).

Assertions involving two or more interfaces may reuse definitions and variables from the local interface protocol checkers

Assertions that verify some global behavior of the DUV may deal with more coarse information units like bits assembled into bytes, or bytes into words, and so on. OVA events and variables that mark in time the assembly of the information in OVA variables may already exist in the local interface checkers.

Checks over global behavior of the DUV at the transactional level should not be implemented using assertions

At the transaction level, the behavior is often expressed in terms of information exchanges using potentially long blocks of data – packets. The events that determine the validity of the generated or received packets are often on a different time scale than the system clock of the design. Also, their decoding, checking and routing may involve complex algorithms. These are neither easily expressible by OVA assertions nor they can be effectively proven by formal tools due to their complexity.

In general, if the check involves extensive data structures, algorithms and spans a large portion of the design, checking by using testbench monitors as described in the section "Passive Response" on page 132 and the section "Verification Environment" on page 20 is more appropriate.

OVA assertions and cover statements may be used to detect significant events

Assertions on local interfaces or those on structures like FIFO's, stacks, arbiters, etc. often detect the occurrence of signal value sequences that mark significant events related to the behavior of the protocol or object. These assertions may be used to signal these events to the testbench. The code of the testbench can thus be simplified.

However, if OVA `check` assertions are used for this purpose and they contain `if-then` clauses, only the non-vacuous successes signal the occurrence of the significant event associated with the assertion. Therefore, the OVA code should be compiled with the `-ova_filter` option. OVA `cover` statements do not usually contain `if` clauses without an `else` and are generally better suited for reporting significant events and for gathering functional coverage.

For information on how to connect OVA assertions in Vera testbench code, see the chapter on Temporal Assertions and Expressions in the *OpenVera User Guide*, and see “DUT state coverage points may be implemented using OVA event coverage” on page 226.

Coverage statements

Functional coverage using OVA cover and assert statements can provide valuable information about the progress of verification on a design.

Coverage can be gathered on all assertions described in the two preceding sections. These provide information about what functionality has been exercised, but may not indicate in what combinations that happened.

Therefore, it is useful to provide additional OVA statements using the directive `cover` that is defined specifically for this task.

Coverage can be subdivided into two basic categories – data/ stimulus coverage and protocol/activity coverage. The former is concerned with the coverage of input sequences, the latter with the coverage of the actual functional aspects of the DUV.

OVA Cover statements shall be used to describe (compliance) test sequences on all interfaces whenever such compliance test are defined

The specifications of standard protocols such as AMBA AHB, PCI, etc. often contain the so called compliance sequences that describe what transactions and in what combinations must occur for the controller in the DUV to be fully exercised. Such compliance sequences when encoded in OVA cover statements provide useful information about the progress of the local interface controller verification. If the tests involve more than one interface, the coverage will indicate to what extent the possible behaviors of the use environment of the DUV have been applied.

OVA coverage gathering shall be turned on in regression tests for all assertion and cover statements each time the design or the testbench has been modified

Every time the design has changed significantly or the testbench was modified, the previous coverage data is not valid anymore and has to be reconstituted. However, if only new tests are added, coverage should be turned on only for these new tests and the results then merged with the existing coverage data.

The -ova_filter option should be used to eliminate vacuous successes

Assertions of type `check` that contain `if-then` clauses without `else` will report success even when the condition in the `if` is false, even though the sequence in the consequent is not matched. This is the so-called vacuous success. The option distinguishes non-vacuous successes in the coverage database.Vacuous successes are not reported in textual assertion reports (on screen or in a file) if the `-ova_success` option is used.

OVA cover statements may be used to characterize combinations of significant events on two or more interfaces

These statements provide information about the occurrence of combinations of events in the design that may not be covered by assertions. This type of coverage is often done in Vera testbenches, hence the OVA statements can be used as sampling events or coverage points for Vera coverage groups.

OVA cover statements shall not contain “if - then” clauses

OVA cover statements are used to detect specific sequences, that is, they must match on that sequence and not anything else. Therefore, unlike in `check` assertions, only a simple unconditional OVA sequence is needed.

For example, if we wish to verify that after req is asserted, ack arrives within 3 to 6 clock cycles, this can be coded using an `if-then` clause as follows:

```
event e1: if (posedge req) then #[3..6] ack;
...
assert c1: check(e1);
```

However, to cover that the sequence req asserted was followed by ack asserted within 3 to 6 clock cycles should be coded in the following way:

```
event e2: (posedge req) #[3..6] ack;
...
cover c2: (e2);
```

$display statements may be attached to cover and assert directives to provide more detailed information in a separate report from the OVA coverage database

The `$display` statement output is part of the OVA assert statement failure message in the on-screen report or in a file. In `cover` statements `$display` executes each time there is a match of the covered OVA event. The `$display` statement can contain a formatting string as well as OVA variables and unit ports as arguments. Sampled values of ports and variable values (before `<=` assignments take place) are output.

In the following example when the req-ack sequence is detected, the fact is logged in the OVA coverage database and at the same time the data value supplied with the ack (for example, a read operation) is placed in the OVA report.

```
unit u( logic clk, logic req, logic ack, logic [7:0] data
);
clock posedge clk {
      event e2: (posedge req) #[3..6] ack;
...
}
cover c2: (e2) $display ("\n data value %x received,   \
\n req sampled at time %0t, ack and data at time %0t",  \
data, $ova_start_time, $time);
...
endunit
```

Assertion coding guidelines

This section lists general rules to follow when writing OVA assertions. They help to avoid efficiency problems during compilation and simulation, and incompatibility with formal or hybrid-formal tools and hardware emulators, the assertions are translated to RTL HDL code using the OVA Compiler. Many of the rules are also checked by the OVA linter in VCS.

General rules

Open-ended interval #[n ..] should not be used without other qualifiers

Failures in assertions involving a time shift with an open-ended interval to model eventuality cannot be detected using simulation. Since most commercial formal tools only can prove safety or bounded liveness properties, such unbounded liveness property cannot be falsified by these tools either.

For example, let

```
clock posedge clk {
      even e:  if (c) then #[1..] b;
}

assert c: check(e);
```

The assertion cannot be proven false, and, unless a b is sampled true after c is sampled true, the evaluation attempt will remain active in a simulation till the end of the run.

Nevertheless, if such an assertion is used in simulation, an indication that b is missing can be deduced from the fact that the evaluation attempt(s) of the assertion has (have) not completed by the end of simulation. Looking at the start time of the attempt and the time of the end of the simulation might reveal that something is amiss.

Similarly, the following `forbid` assertion cannot succeed (it is the dual of the preceding case) and unless it fails all evaluation attempts will remain active till the end of simulation. If many such attempts are created, severe performance problems could arise.

```
clock posedge clk {
      even e:  a #[1..] b;
}

assert c: forbid(e);
```

Most formal and hybrid-formal tools can detect a failure of the above `forbid` assertion, but cannot do the same in the case of the dual check assertion mentioned earlier.

Note that open-ended intervals are useful when constrained by the `length` or `istrue` operators. The former is particularly useful in cases exemplified by the following OVA event:

```
event e: if (y) then #1 length [10..20] in (c #[1..] b
#[1..] d);
```

It states that after y, c must be followed by b that must be followed by d, and the total extent of the sequence c ->> b ->> d is at least 10 and at most 20 clock ticks.

Open-ended # delays shall not be used in antecedent sequences without other constraints

It is often the case that the trigger condition of a `check` assertion on a sequence s2 is a sequence s1 that detects that one boolean condition b1 true is eventually followed by another boolean condition b2 true. The temptation may be to write this assertion in the following way:

```
clock posedge clk {
      event s1: b1 #[1..] b2;
      event s2: if (ended s1) then s2;
}
assert c: check(s2);
```

The problem with this form is that an evaluation attempt of s1 is started anytime b1 is true, but that attempt will remain active, searching for b2 to be true till the end of simulation. Each such match of s1 re-triggers the check on s2. Yet, the probable intent was to detect only the 1st match of b2 after b1. This can be accomplished in the following way:

```
clock posedge clk {
      event s1: b1 #1 (!b2)*[0..] #1 b2;
      event s2: if (ended s1) then s2;
}
assert c: check(s2);
```

Notice how s1 was modified using the `*` operator to match only on the first occurrence of b2 strictly after b1.

Time shift and repetition operators should not involve large bounded intervals

Use as tight timing bounds as is expected from the protocol, and if the bound is large (> 20) then consider rewriting the property in some other way. For example, time span can be measured using OVA variables.

For example,

```
event e: xyz #[1..4600] abc;
```

This could take a long time to compile.

The “past” operator shall not be over a large number of clock cycles (>100)

Large number of clock cycles may reduce verification performance because the past operator resembles a shift register with as many stages as the delay argument.

Consider reformulating the same property without the use of a deep look into the past using that operator. Often this can be achieved by identifying a condition that marks the instant when the value is valid. Use that condition to store the value in an OVA variable for use later. Note, however, that overlapped transactions (for example, a pipeline) cannot be handled in this way because the single variable would be overwritten.

Only clocked events and assertions shall be used

Unclocked events used with ended or matched, and assertions over unclocked events use simulation time as the clock. This is inefficient during simulation and usually not needed when verifying synchronous designs. Also, it cannot be used with OVA Compiler.

For example,

Correct clocked use:

```
clock posedge clk {
      event e1 = ...;
}
assert a: check/forbid (e1);
```

Inefficient simulation time clock and not usable with OVA Compiler:

```
event e1 = ...;
assert a: check/forbid (e1);
```

A large time window shall be delimited using OVA variables

Sometime a property should span a large time window (a large number of clock ticks after a trigger condition). As mentioned earlier, using large repeat * or delay intervals # may lead to long compilation times. A better way is to express such properties with the help of OVA variables.

For example, suppose that after start the signal stop should be asserted within 1 to 4096 clock ticks. A direct but inefficient way to code this would be to use #[1..4096] in an OVA event:

```
clock posedge clk {
      event e: if (start) then #[1..4096] stop;
}
assert c: check(e);
```

If the protocol is such that no start can be issued before a stop is received (that is, there are no overlapping start-stop transactions), the property should be coded more efficiently as follows:

```
logic [12:0] timer = 0;
clock posedge clk {
      timer <= (reset || stop) ? 0:
               start ? 1 :
                    (timer > 0) &&
                    (timer <= 13'd4096) ? timer + 1 :
                                      timer;
      event e: if (stop && !reset) then
                    (timer > 0) && (timer <= 13'd4096);
}
assert c: check(e);
```

Assertions should be added to verify that while timer > 0 then there is no start issued, that there is no stop while timer == 0 and that start and stop cannot happen simultaneously.

To detect a change in a bit-vector value, edge operators (posedge, edge and negedge) shall not be used

If applied to a bit-vector then only the least significant bit is tested. To detect a change of value on a bit-vector expression <*exp*>, use the following boolean expression:

```
bool change(exp): past(exp) != exp;
```

`change(...)` can be instantiated like `edge`.

Verilog reduction and word-level operators shall be used to simplify boolean expressions in OVA sequences

Often it is necessary to test for specific values of bits in a bit vector. Instead of conjoining comparisons of each individual bit, it is often possible to use Verilog reduction operators and word-level operators to simplify the boolean expression and thus improve verification tool performance.

For example:

- To test that all bits are set to 1 on a word W, use (&W) instead of (W[0] == 1) && (W[1] == 1) && ...
- To test that there is an odd number of bits in W, use (^W), even number, use !(^W)
- To verify that there is at least one bit set in the same position in words W1 and W2, use |(W1 & W2)

The use of level expressions shall be given preference over edge expressions

Edge expressions add overhead during verification and in many cases edge detection is not needed.

For example, to check that a pulse on signal x is only one clock cycle wide can be written as:

```
if (posedge x) then #1 negedge x;
```

However, a simpler equivalent check can be stated as:

```
if (x) then #1 !x;
```

Similarly, instead of using

```
if (posedge request)then #[2..10] posedge ack;
```

it is often possible to state it as follows:

```
if (posedge request)then #[2..10] ack;
```

In this case however, the meaning has been changed. The modified sequence is correct as long as the detection of the edge on ack after the edge on req is not required. An assert statement on the modified version will succeed on the first occurrence ack after the rising edge of req.

Top-level “if” statements shall be used to allow filtering of vacuous successes

If a check assertions with a top-level `if` clause succeeds because the condition in the `if` is false, it is said that the assertion succeeded vacuously. In that case the consequent – the sequence that should happen when the condition is true – has not been verified. If all evaluation attempts of the assertion succeed in this way, there is no failure reported, yet nothing has been verified either. Therefore, it is useful to detect and filter out vacuous successes so that the effectiveness of the assertion during a simulation run could be assessed.

VCS and OVAsim do support automatic vacuity filtering over conditions in top-level nested `if` statements (without `else` clauses) in check assertions. Therefore, implications should be coded using `if` statements rather than hidden in a boolean expression.

For example, “ack cannot be asserted unless req is asserted” can be coded as follows

```
clock posedge clk {
      event e: req || !ack;
}
assert c: check(e);
```

However, if we are verifying the component that generates ack, it is better to express this event as follows:

```
event e: if !req then !ack;
```

In the latter case the situation when req is never de-asserted during simulation and thus ack de-asserted is never tested can be detected by observing that there is no real success on the assertion in the OVA coverage report.

When an assertion involves signals that have no causal relationship between them (signals of the same direction or internal to the design), there is no advantage to using the `if-then` form over the boolean form. For example, this is the case of verifying that only one bit is asserted in a vector of enable signals.

Note that in the case of OVA `forbid` assertions it is not possible to detect vacuous firing in this way. The reason is that the trigger condition cannot be automatically identified.

An “if” statement without an “else” clause should not be used in a forbid assertion

For example, suppose it is necessary to verify that when c occurs then it is not followed by b in the next clock cycle. This could be coded in the following way:

```
clock posedge clk {
      event e1: if (c) then #1 !b;
}
assert c1: check(e1);
```

This property could also be expressed using the violating sequence and a forbid assertion. The temptation may be to write it as follows:

```
clock posedge clk {
      event e2: if c then #1 b;
}
assert f2: forbid(e2);
```

The problem is that whenever c is false then e2 succeeds and hence forbid will fail. This is clearly not the intent. It is required that only the sequence a followed by b should fail and it is exactly that way the OVA event should be written:

```
clock posedge clk {
      event e3: a #1 b;
}
assert f3: forbid(e3);
```

Here, *f3* will report a failure only when the sequence *a* followed by *b* is detected in the simulation trace.

To summarize: `if (bool) then good_sequence` is usually in a `check` assertion, while the form `bool && bad_sequence` is placed in a `forbid` assertion (`&&` can be replaced by the time shift operator `#`).

“ended” or “matched” should not be used on an event that contains an “if” without an “else”

Example:

```
event e1: if c then #1 b;
event e2: if ended e1 then #1 c;
```

Whenever c is false, event e1 will match because the “if - then” implication is satisfied. This means that e2 will also trigger and try to match on c at the next clock tick. This may not be the intent, however.

Consider changing event *e1* as follows:

```
event e1: c #1 b;
```

“Ended” or “matched” should be used when an event is followed by another event

Example:

```
event e: ...;
event e1: e #M e11;
event e2: e #N e21;
etc.
```

To improve performance, consider replacing it by

```
event e: ...;
event e1: ended e #M e11;
event e2: ended e #N e21;
etc.
```

The performance optimization comes from the fact that in the original form event e is expanded in e1, e2, ..., while in the modified version only one copy of e is evaluated and only the outcome of matching e is used in e1, e2, ...

The above situation often arises when creating coverage events.

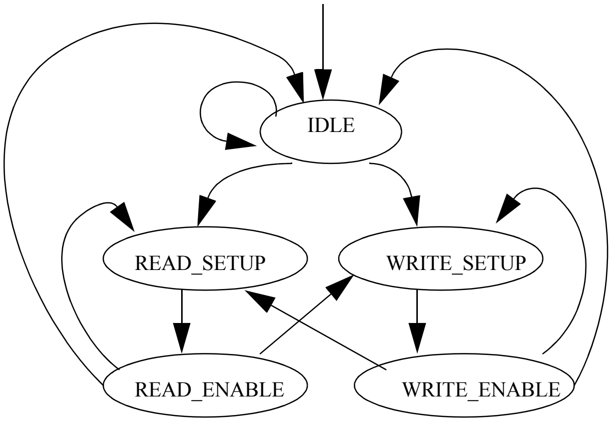
State machines shall be coded using OVA logic variables assigned NBA

The specification of a bus protocol is often described with the help of a state machine. This state machine can be useful in simplifying the OVA assertions. It can be easily implemented using OVA variables.

For example, the state machine Figure 7-3 of the AMBA APB bus specification can be modeled as follows:

```
logic [2:0] machine_state = IDLE_STATE;
//machine state transitions
clock posedge clk {
      machine_state <=
           (!resetn) ? IDLE_STATE:
           (idle_idle) ? IDLE_STATE:
           (idle_readsetup) ? READ_SETUP:
           (idle_writesetup) ? WRITE_SETUP:
           (readsetup_readenable) ? READ_ENABLE:
           (writesetup_writeenable) ? WRITE_ENABLE:
           (readenable_idle) ? IDLE_STATE:
           (readenable_readsetup) ? READ_SETUP:
           (readenable_writesetup) ? WRITE_SETUP:
           (writeenable_idle) ? IDLE_STATE:
           (writeenable_writesetup) ? WRITE_SETUP:
           (writeenable_readsetup) ? READ_SETUP:
                                     machine_state;
}
```

Figure 7-3 A state-machine for the APB Bus protocol



Each line in the above definition corresponds to one transition in Figure 7-3. The conditions in the `?:` statements are booleans representing the enabling conditions of the corresponding transitions. For instance, idle_idle could be defined as

```
bool idle_idle: (machine_state == IDLE) && !psel;
```

A rule that verifies that PENABLE is not active while in the IDLE state can then be written in the following way:

```
event SnpsApb_e_no_enable_in_idle:
          if (machine_state == IDLE) then !penable;
```

Similarly for other signal values that must have a specific value in a state of the protocol. Note that if a signal value is used to make a choice between two or more transitions in the state machine then (of course) it cannot be constrained as shown above. If a rule requires this to happen, then there may be something wrong either with the rule or the state machine.

Macro definitions shall not be used to define constants

The reason is that macros are global in compilation and thus could interfere with definitions from another module or unit. Use parameters or OVA untyped expressions to define constants. For example, the state values of AMBA APB bus can be defined as untyped OVA expressions as follows:

```
IDLE_STATE:        3'b001;
READ_SETUP:        3'b010;
WRITE_SETUP:       3'b011;
READ_ENABLE:       3'b100;
```

OVA variables should be used to store expected results for data checking

OVA variables can be used to keep / compute expected value that will be compared later with monitored signals.

For example, to check that incoming parity is the same as outgoing parity can be achieved as follows:

```
logic v_parity = 0;
clock posedge clk {
      v_parity <= in_ack ? ^ ad_in[7:0] : v_parity;

      event e: if (out_ready) then ^ad_out[7:0] ==
v_parity;
}
assert c: check(e);
```

If multiple transactions overlap and may carry different data values, the above simple scheme is not sufficient because the same variable is used and shall be overwritten before used. In that case, a more elaborate data storage scheme such as a FIFO should be implemented using OVA variables.

Note also that if a large piece of data is to be checked across a complex DUV using assertions then formal tools may have difficulty proving or disproving the assertions.

Assertion shall be disabled upon reset condition

When an assertion evaluation is triggered because, for example, the antecedent condition in a check assertion is sampled true, the property expressed may not hold if during the sequence evaluation a reset occurs. Since OVA does not have a direct way to specify a disabling condition, incorporating a reset into the assertion depends on the specific form of the property. Please see also the rule regarding detection of a reset condition in Magellan "A time_0 variable should be provided if the initial reset is not visible" on page 182.

The following example shows how it can be done in a rather typical situation where the antecedent and the consequent in a check assertion are sequences, and the consequent is an "until" like operation on booleans. Let b1 and b2 be boolean expressions:

```
clock posedge clk {
      // antecedent sequence
      event e_ante: istrue !reset_condition in
                    main_antecedent_sequence;
      event e_property1: if (ended e_ante) then
                    b1*[1..] #1 (b2 || reset_condition);
}
assert c: check(e_property1);
```

Notice that the antecedent will not match if reset_condition becomes false, which then causes e_property to have a vacuous success, while if reset_condition becomes true while evaluating the consequent, then it matches on the reset and terminates with success (this time not vacuous, however.)

If the antecedent is a boolean expression, say b3, then the antecedent is simply a conjunction with !reset_condition:

```
event e_property1: if (!reset_condition && b3) then
                   b1*[1..] #1 (b2 || reset_condition);
```

If the assertion is of the `forbid` kind, then disabling the assertion is achieved using `istrue` as in the antecedent of the case above:

```
clock posedge clk {
      event e_property2: istrue !reset_condition in
                              forbidden_sequence;
}
assert f: forbid(e_property2);
```

Note that:

- The reset_condition should also be used to reset auxiliary variables to an initial state.
- The reset_condition should also include detection of abort situations, not just resets. For instance, a STOP being issued by a target during a PCI transaction.

Place top-level "if" statements into the assertion to simplify reuse of events in other event definitions

For example, suppose that we wish to verify that: *When a occurs then the sequence b #1 c must hold*, and also *if a is followed by b #1 c, then some other sequence e2 must hold*.

One way to code it is as follows:

```
clock posedge clk {
      event e2: ... ;
      event e_if: if (a) then #1 b #1 c;
      event e_a_e1: a #1 b #1 c;
      event e_next: if (ended e_a_e1) then e2;
}
assert c_if: check ( e_if );
assert c_next: check ( e_next );
```

A simpler and possibly easier form to understand is as follows:

```
clock posedge clk {
      event e2: ... ;
      event e_abc: a #1 b #1 c;
}
assert c_if: check ( if a then e_abc );
assert c_next: check ( if (ended e_abc) then e2 );
```

Note that the clock from e2 is inferred for the if (ended e_abc) part in the c_next assert statement.

Control of assertions in simulation may be enhanced by using category and severity arguments

The `severity` argument in an assertion enables controlling whether the assertion failure causes the simulation to abort, stop or continue.

The category argument provides fine control over which assertion should be enabled or stopped, and allows to sort failure report by category value. This finer control is further enhanced by the use of a mask in the control system tasks

For example, suppose that the 24 bits of the category argument are subdivided as follows:

24 - 23	: Block number (out of four top-level blocks)
22 - 19	: Not used
18 - 15	: Sub-block number
14 - 5	: Not used
4 - 1	: Function number
0	: Interface or internal assertion

The system task call

```
$ova_category_stop  (24'h30_0000,24'h30_0000);
```

will stop all assertions in block 3, while

```
$ova_category_stop(0,24'h00_0001);
```

will stop all assertions internal to the design.

Similarly, specific assertions can be started using the `$ova_category_start` task.

Guidelines specific for OVA Compiler

Case equality shall not be used in expressions

This is non-synthesizable. Comparisons with Z and X values are also not possible for the same reason.

An uninitialized OVA variable should not be used

An uninitialized variable may cause a simulation-formal mismatch due to differences in interpreting the initial unknown value X.

A multidimensional array shall not be used as a unit port

This may not be synthesizable.

A time_0 variable should be provided if the initial reset is not visible

In many protocols, there are properties that must hold from the moment reset is de-asserted to the occurrence of some condition. Since the reset sequence is handled in a separate initialization sequence, the verification engines may not see the de-assertion of the reset signal. The problem then is that the assertions may not be triggered and if used as assumptions, they may not be activated possibly leading to under constraining and dead ends.

A solution is to provide an OVA variable (for example, called time_0) that is initialized to 1 and then reset to 0 on the first clock tick, retaining 0 thereafter. Assertions that must hold between a reset (time 0) and some other condition can be conditioned by either the de-assertion of reset or by the fact that time_0 == 1. For example,

```
logic time_0 = 1’b1;
clock posedge clk {
      time_0 <= 1'b0;
      event e_no_data_valid:
      if ((negedge reset) || time_0) then
           istrue !DATAVALID in
                #[0..] ((GRANT != 0) || reset );
}
```

A property (assertions) should not be stated solely over an OVA variable value if it is to be used as an assumption on DUV inputs

Example:

```
logic [bw-1:0] tmp = bw'b0;
clock posedge clk {
      tmp <= c1 ? port_A :
                c2 ? port_B: tmp;
        event e: (| tmp) == 1'b1;
}
```

```
assert c: check(e);
```

This type of an assertion cannot be used effectively as an assumption because it requires constraining the signals driving the OVA variable tmp in the past clock tick. Depending on the tool and the complexity of the assumptions, this may not be possible when doing random simulation constrained by OVA assumptions. The result would be that no constraint is imposed at clock tick t leading to a potential inconsistency and dead end at tick t+1 if tmp == 1 does not hold. Consider modifying the assertion as follows:

```
clock posedge clk {
      event e: if (c1) then (| port_A) == 1'b1
               else if (c2) then (| port_B) == 1'b1;
}
assert c: check(e);
```

The constraint on the ports is now applied in the current clock cycle rather than in the past one.

Signals that are inputs to the design should not appear only as arguments of the "matched" or "ended" operator in an assertion that is to be used as an assumption on DUV inputs

The problem is that this form of an assertion cannot be used effectively in random simulation under OVA assumptions because depending on the tool used it may require constraining inputs in past clock cycles.

Example:

Let d and b be design inputs to be controlled by the assumption.

```
clock posedge clk {
        event e1: d #1 b;
        event e2: if c then #1 (ended e1);
}
```

```
assert c: check(e2);
```

Try to rewrite event e2 without the use of `ended`. In the above example the solution is simple due to the `#1` delay before `ended`:

```
clock posedge clk {
//      event e1: d #1 b;
        event e2: if c then d #1 b;
}
assert c: check(e2);
```

In general the transformation may not be as simple as that. It may be preferable to approach the problem differently right from the start rather than trying to rewrite the assertion later.

Signals that are inputs to the design should not appear only as arguments of the "past" operator in an assertion that is to be used as an assumption on DUV inputs

Example:

Let *a* and *b* be design inputs.

```
clock posedge clk {
      event e: if c then past(a && b);
}
assert ch: check(e); // to be an assumption
```

This form is not causal and cannot effectively be used as an assumption because it requires constraining inputs in past clock cycles. Most likely it should be an assertion on c rather than an assumption on a && b. That is,

```
clock posedge clk {
        event e: if !(a && b) then #1 !c;
}
assert ch: check(e);
```

When writing OVA assumptions, the sequence expressions should be projecting values on inputs forward in time using `#` and `*` operators rather than using the `past` operator.

In general, the form of an assertion used as an assumption shall have the following form:

There shall be at least one DUV input that can be constrained at every clock tick of the consequent.

```
event e: if (expr( past, state,
               ended, matched,
               present outputs) )
                              then
                        sequence(inputs, perhaps
outputs);
```

Example:

```
event e:
      if (past(in_x) && out_y && !reset && var_z) then
                                 in_x*[1..] #1 !out_y;
```

In the example when the `if` condition is satisfied the input in_x must hold asserted until out_y is deasserted.

Note:

The dead-end avoidance algorithm in the hybrid-formal verification tool Magellan resolves the issues related to constraining values in the past. However, the complexity of the assertions may limit the usability of the algorithm. Therefore, whenever possible it is preferable to respect the above three rules.

‘ifdef conditional compilation shall be used to eliminate assertions not compatible with OVA Compiler

For example, formal or hybrid-formal verification tool works with two states only, hence case equality checks which require four states (for example, Hi-Z detection using case equality) can be conditionally compiled under user control only for simulation as follows:

```
‘ifdef SNPS_FORMAL
   // do nothing
‘else
      bool SnpsPci_b_Par_not_tristated : par !== 1’bz;
      bool SnpsPci_b_Data_driven_during_read :
           ((ad[31:0] !== 32’bz) &&
           ( SnpsPciRead_bit &&
               ( SnpsPciRead_dataPhase_bit ||
                   SnpsPci_trdy_asserted)
           ));
clock posedge clk {
      event SnpsPci_e_Par_not_tristated: if  ... ;
...
}
assert ...
‘endif
```

When the macro `SNPS_FORMAL` is not defined, the booleans, events and assertions related to checking for Hi-Z are only included in simulation and are eliminated otherwise for the formal verification tool.

Reusable OVA Checkers

A reusable checker is usually developed as a self-contained monitoring entity for standard or commonly used bus protocols, such as PCI, Utopia, SPI, AMBA, etc. If integrated with a testbench, it represents a passive transactor which only performs monitoring. It can also do logging of coverage points, and a testbench can use some or all the assertions in the checker for detecting specific sequences as sampling events or coverage points in coverage groups.

Some checkers can also be used as abstract models of the environment, that is to say, as constraints for random test generators and assumptions for formal verification engines. The current level of sequential constraint solvers and formal engines may not yet handle entirely the assumptions needed by more complex multi-layer protocols such as PCI Express or HyperTransport. Proper organization of the checkers is thus needed for use as assumptions.

This chapter contains a collection of guidelines for the development of Reusable Assertion-Based Checkers using the OpenVera Assertion language (OVA). These guidelines are based on the experience of implementing OVA checkers for various protocols such as AMBA AHB, PCI, PCI Express, etc. The guidelines will evolve with time, as more experience with the use of the checker is obtained as well as with improvements in constraint solving technology.

The development of a reusable checker must take into account many factors, such as

- protocol options
- parameterization of signal widths, resource requirements
- assertions and / or assumptions (constraints)
- use in simulation and / or formal tools
- 4-valued and / or 2-valued evaluation of expressions
- system and / or block-level verification

- reporting of parameter errors
- failure reporting and failure message contents
- verification assertions vs. coverage of sequences
- efficiency
- packaging for Verilog and VHDL design environments

The guidelines are subdivided into subsections that address the above issues from the following angles:

- Property rules
- Architecture
- Naming convention
- Coding style
- Testing
- Documentation and Release items for Assertion-based Checker IP.

Property extraction guidelines

The first question which arises when faced with the development of a checker for a bus protocol is often “Where do I start?” The reason is that the protocol description is spread over many pages in different forms: textual, timing diagrams, state machines, perhaps some algorithms and even logic diagrams. Yet, for an assertion-based protocol checker we need a list of precise statements that can then be expressed in the assertion language.

A set of rules that represent the required behavior shall be extracted from the specification document

The rules shall have each a succinct name, a brief description and a cross-reference to the section(s) (and paragraph numbers) in the protocol specification document.

More constraints on the behavior than the specification imposes should not be introduced because it may result in false failures of assertions, and it may lead to false positive results or inconsistency if used as assumptions. This can especially happen when interpreting timing diagrams in which actual causal relations are not clearly identified. For example, event B may be shown between 2-6 cycles after event A, and event C between 3-8 cycles after event A. Unless there is a timing constraint between B and C or the textual description implies such a constraint, no temporal precedence between events B and C should be imposed. The over constraint may be detected when testing the rule as an assertion.

Similarly, a less strict rule will be more permissive and may lead to a false positive outcome if used as assertions and to false negative results if used as assumptions (constraints). Such a situation may arise when we ignore some protocol requirements that are spread over multiple diagrams or spread over disparate pages. The less constraining rules are often best detected if used as assumptions (constraints) because they may lead to failures on other assertions or lead to a dead end in random simulation constrained by such “loose” assumptions.

The rules shall be subdivided according to the signal direction they control

It may often be difficult to decide (especially for novice verification engineers) what signal is to be constrained by a specific rule. Fortunately, with careful reading, the specifications are usually quite clear as to what signal is concerned.

The reason for such subdivision is that depending on whether we use the checker in a system or a block-level test. In a system-level test, all the rules over the bus should be used. In a block-level test, only those rules that pertain to the block output signals may be used as assertions, while the other rules can be used as assumptions controlling the inputs of the block or, in simulation, to check that the testbench generates legal inputs.

For example, in a single Master - single Slave system, there are signals from Master to Slave (M-S) and from Slave to Master (S-M). The rules should be subdivided into two subsets according to the direction they characterize, M-S and S-M. Let FRAME be a signal from Master to Slave and READY a signal from Slave to Master. If the rule says “Whenever FRAME is asserted then within 1 to 8 cycles READY should be asserted”, it indicates a required behavior of READY, assuming that FRAME is asserted and thus it belongs to the S-M subset. Note that the “whenever” or “if” condition may refer to signals from either set, it is the signals in the conclusion that determine the subset. In OVA, this could be described by the following event:

```
event e_READY_on_time: if (posedge FRAME) then
                       #[1..8] (posedge READY);
```

Similarly, the statement “When FRAME is asserted it must remain so until READY is asserted” is a rule that controls the behavior of FRAME and not that of READY. Therefore, the rule belongs to the M-S subset. It can be expressed in OVA as follows:

```
event e_FRAME_stable: if (posedge FRAME) then
                    FRAME*[1..] #0 (posedge READY);
```

Notice that there is no time constraint involved here, only the stability of FRAME until READY is detected (inclusive). The timing aspect constrains READY.

The following example shows a situation where the signal to be constrained is even less obvious from the specification:

“FRAME must not be asserted if DATA==eof is going to appear on the output within the next four cycles.”

In this case, it seems that it is FRAME that is to be constrained by some future signal value. However, such a situation is non-causal and can be resolved in two ways:

Either there is another signal (possibly internal to the DUV) that does indicate that DATA==eof will occur in the future and that signal can be used to constrain FRAME, or the constraint is actually on DATA not being equal to eof if FRAME is asserted.

In the latter case it can be expressed as follows:

```
event e_DATA: if (FRAME) then #1 (DATA != eof)*[4];
```

The distinction of assertions by signal direction is extremely important if the checker is to be used in formal or hybrid-formal tools for block-level verification such that some property rules act as checks and others as assumptions.

Rules may refer to internal signals in the device

Such rules may not be always verifiable by just observing the bus. The rules may be usable only in block-level white-box verification. These rules should be put in a separate sub-category that cannot be used as assumptions on the design. The inclusion of the rules should be controlled by a unit parameter.

Rules may refer to internal driver signals of external busses

For example, a rule may state: “Master never starts a transaction cycle unless GNT# is asserted”. This means that we should check that when there is no grant to a master then that master must not start a transaction. However, if there are other masters on the same bus then it is not possible to identify which master started a transaction on the bus by just observing the bus (unless no grant is given at all). In this case, by referring to the internal driver enable signal in the design it is possible to verify that the particular master did not drive the bus.

For example, the signal `frame` on a PCI bus is such a shared signals between a number of devices. Suppose that OVA unit ports of a Master checker are

```
frame_out_n, frame_in_n, frame_en_n
```

In a white-box verification as a Master checker with access to the internal signals, the OVA unit ports will be mapped to the internal signals. For example,

```
          .frame_out_n (DUV_frame_out_n),
          .frame_in_n (DUV_frame_in_n),
          .frame_en_n (DUV_frame_en_n),
```

In a black-box verification in which there is no access to the internal signals frame_in_n and frame_en_n, the port mapping may be:

```
          .frame_out_n (FRAME_n),
          .frame_in_n (FRAME_n),
          .frame_en_n (FRAME_n),
```

where FRAME_n is the shared bus signal. That is, the assertion that verifies that the specific device is driving is weakened or even virtually removed.

The set of rules should not contain redundant rules

Make sure that the rules are disjoint, that is, that one rule does not cover (parts of) behaviors characterized by another rule. The reason is not only the efficiency of the checker, but also easier problem identification in case of a failure of the assertions.

For example, consider the following case from the PCI protocol specification:

Rule 1: If PERR# is enabled and a data parity error is detected by a master during a read transaction, *the master must assert PERR# two clocks after a completion of a data phase in which a parity error occurs*. (Section 3.7.4.1 of PCI 2.2 Specification.)

Rule 2: Master always drives *PERR# (when enabled) for a minimum of 1 clock for each data phase* in which a parity error is detected (Section 3.8.2.1 of PCI 2.2 Specification.)

Rule 1 detects the first occurrence of PERR# and at that point Rule 2 is also verified because once PERR# is asserted it is for at least one cycle. Rule 2 is thus redundant. In the description, Rule 1 should refer to both items in the protocol specification, however.

Contradictions in the rules should be eliminated

Such contradiction may occur due to misinterpretation of the specification in an area only partially related to another rule. Identifying contradiction is a difficult task and it may be detected only when testing the entire checker, in particular if the properties are used as assumptions.

Compliance statements / functional coverage should be expressed using OVA cover statements

Often the protocol specification is accompanied by so called compliance statements that state what sequences of transactions or operations on the bus must be exercised during simulation so as to verify the design to an acceptable degree. These compliance statements can be easily expressed using OVA cover statements and then used in functional coverage evaluation.

For example, to observe if a Write followed by a Read to a different peripheral has occurred, the following cover statement can be included:

```
clock posedge clk {
      event SnpsApb_e_wr_rd:
           SETUP_WR_SELECT #1 ENABLE_WR_DSEL #1
           IDLE #1 SETUP_RD_COV #1 ENABLE_READ;
}
cover SnpsApb_cv_WriteAfterRead: (SnpsApb_e_wr_rd);
```

where each of the identifiers represents a particular phase of the protocol expressed using an OVA logic variable or event.

Since compliance statements may not be required in each verification case, these events and cover statements should be included selectively under the control of a macro.
For example,

```
'ifdef SNPS_OVA_PCI_COMPLIANCE_STATEMENTS
...
'endif
```

Configuration choices shall be selected by OVA unit parameters

As shown in the above examples, the inclusion of certain parts of the checker may be under the control of a macro. Its definition can be provided in a header file (see Architecture), or as part of the compilation command using `+define+`*`MACRO_NAME`*.

The control by a macro may be useful when it is to affect all instances of the checker in the verification environment. Yet, since macros are global in a compilation, the name of the macro should clearly distinguish its application.

A preferred form is the selection by parameters, especially if each instance needs to have a different form. The selection is implemented using OVA `for` loop constructs that can implement a form of generate statements like in Verilog 2001 or VHDL.

For example, suppose that the above compliance statement should be included in one instance of the checker but may not be in another one because we wish to check coverage only on the latter bus. We can include a parameter to control this selection as follows:

```
unit checker_name
      #(parameter snps_compliance_statement = 0; //
default not select
           ...  // other parameters
       )
      (logic ... // port decalrations
      );

...
```

```
for (i=snps_compliance_statement; i==1; i=i-1) {
      clock posedge clk {
         event SnpsApb_e_wr_rd:
               SETUP_WR_SELECT #1 ENABLE_WR_DSEL #1
               IDLE #1 SETUP_RD_COV #1 ENABLE_READ;
      }
      cover SnpsApb_c_WriteAfterRead:
(SnpsApb_e_wr_rd);
}
... // rest of checker
endunit
```

Macro names shall have a name that clearly distinguishes the origin, application and purpose of the macro

The name thus should consist of the following fields:

```
<company>_<product>_<component>_<specific_define>
```

For example:

```
‘define SNPS_OVA_PCI_BLOCK_LEVEL_CHECK
```

Reusable Checker Architecture

The following factors must be considered in structuring an assertion-based checker:

- protocol layers
- assume vs. assert
- sharing vs. local use of auxiliary variables
- reuse of common definitions
- configurability
- failure reporting

Global configuration macro definitions may be in a header file

Global configuration macros that affect all instances of the checker as discussed in the preceding section, should be placed in a separate header file, so that only this file needs to be edited when changing the global configuration. This file is then included wherever needed. The alternative is to use `+define+` compilation option. However, a large number of macros can make this approach harder to manage.

Common definitions and shared auxiliary variables shall be placed in one or more OVA templates

Checkers for any realistic bus protocol usually rely on auxiliary variables, both combinational and state, to facilitate protocol modeling and to make the checker easier to understand and in some cases more efficient. For example, variables are needed to store input data values that are later compared with those output by the design. Such variables may be used in several assertions and by other variables. Moreover, there may be sub-sequences in OVA events that appear in a number of assertions and thus should be defined once only and then reused by instantiation wherever needed. These common variables and event definitions should be packaged together in one or more OVA templates for easier maintenance and reuse.

Assertions shall be grouped by signal direction

As mentioned earlier, rules should constrain only signals of one direction. This separation by direction should also be reflected in the checker architecture. Different forms may be considered depending on the sharing of common definitions.

If the properties that deal with distinct signal directions do not depend on the same variables, then they can be placed in separate OVA units and instantiate their specific common definition templates.

If however they do share common variables and all the properties regardless the signal direction may have to be used at the same time, then it is preferable to place all the properties in one unit and to share a single copy of the common definitions. In this way there is no replication of the same objects. Since in some cases you may need only assertions related to one of the signal directions, the selection of the appropriate subsets should be placed under the control of a unit parameter (not a macro because each checker instance may need a different configuration.)

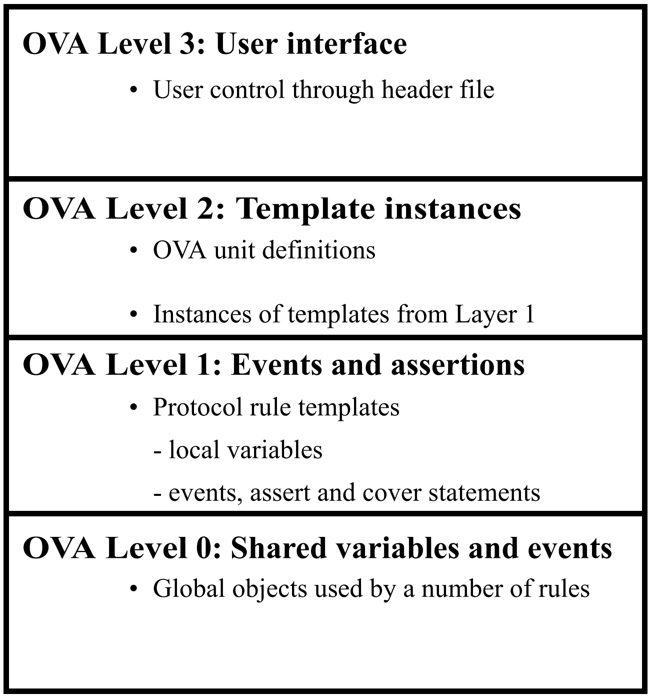
Assertions may be packaged in templates according to the rules they implement

Each rule as extracted from a protocol specification may result in one or more OVA assertions, possibly relying on both shared and local auxiliary variables and definitions. When the rules lead to such a more complex implementation (rather than a single event and assert definitions), it may be preferable to encapsulate the implementation of each rule in a separate template. The collection of such templates related to one signal direction should be contained in a single file for ease of maintenance. The templates are instantiated in the appropriate OVA units mentioned above.

Single-layer protocols shall use 2- or 3-level architecture

Many on-chip and PCB memory / system bus protocols may have a simple one-layer structure, most signalling is achieved out-of-band using separate control signals. For example, ARM AHB and APB busses would fall into this category. In this case, the architecture is as shown in Figure 7-4

Figure 7-4 OVA checker architecture for single-layer protocols



OVA Level 3: User interface

- User control through header file

OVA Level 2: Template instances

- OVA unit definitions
- Instances of templates from Layer 1

OVA Level 1: Events and assertions

- Protocol rule templates
  - local variables
  - events, assert and cover statements

OVA Level 0: Shared variables and events

- Global objects used by a number of rules

OVA Level 0: This level contains variable declarations and assignments, and event declarations that are shared by several rules. For example, it may contain a state machine that tracks the state of a transaction on a pipelined bus. Several rules may refer to the state variables to ascertain properties in different states of the transaction. The common template will be instantiated with a suitable name (see Naming conventions) at Level 2. Fully qualified names (for example, SnpsPci_<Variable/Event Name>) should be used to access the objects in the template instance from a rule template instance.

OVA Level 1 (optional): The rules of the protocol are coded in the assertion language and encapsulated in templates, one template for one or more rules. The template may contain local variables and event declarations for use by the assertions of the rule, and all of them can refer to the objects in the instance of the common template. In cases where the assertions related to rules consist of event and assert statements only and do not require many local auxiliary variables, it is possible not to use Level 1 encapsulation and place the definitions directly in the OVA unit(s) at Level 2.

OVA Level 2: At this level, templates are instantiated for each signal direction inside an OVA unit for a single signal direction or if for multiple directions then each set of template instances for a given direction can be selectively enabled or disabled under the control of a parameter value. The port names of the units are those from the protocol specification.

OVA Level 3: Finally, this Level contains a header file that defines global configuration macros and constants. This file is included (‘include) in all OVA files. The third level may also contain an example of a file containing OVA bind statements that links the checker to a sample design. The user only needs to edit this file for the specific design under verification.

Multi-layer protocols may use a 2- or 3-level hierarchical or flat architecture

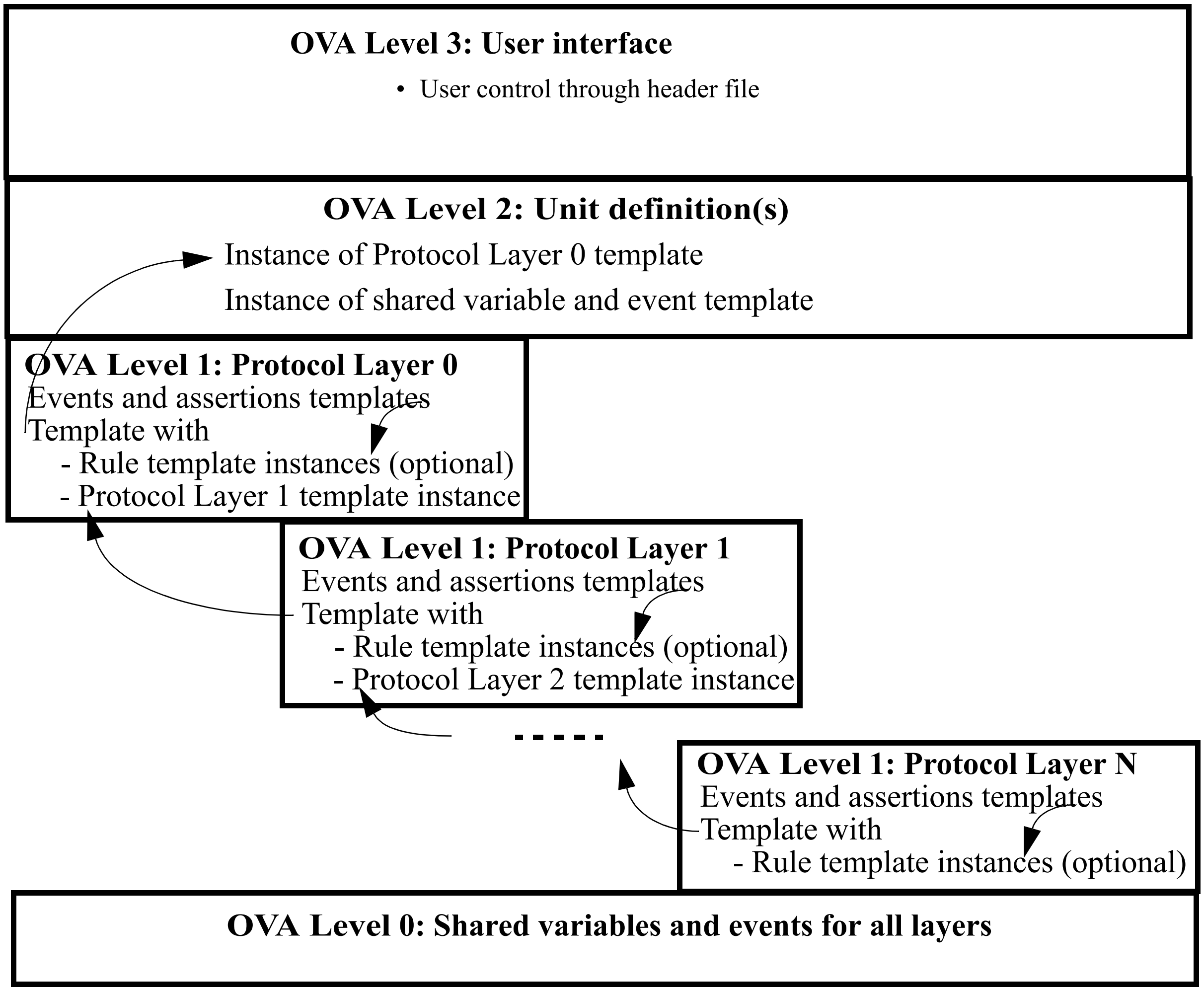
Peripheral and off-PCB bus protocols using fast serial communication (optical or electrical) with the help of SERDES devices may have most control communicated in-band, as part of control and data packets. Viewed externally, there may only be data and clock signals for each direction of communication. PCI Express bus is one such example. Protocols of this kind are layered, much like the ISO Open System Interconnect (OSI) protocol stack, lower layers providing services to the higher layers. For example, PCI Express consists of the Physical, Data Link and Transaction layers. Only the Transaction layer communicates with the user-defined Application layer and that upper-level interface need not be standardized.

In the case of layered protocols, it is useful to structure the rules and the actual checker also into the corresponding layers. Each layer has a similar structure as described for the single layer protocol. It may have its local common variables and definitions and rule templates. In addition it needs to access some definitions and variables from the lower-layer protocol that form the interface between the layers, and it has to provide similar services to the layer above. Since OVA units cannot be nested within units, the protocol stack of rules is implemented using templates.

The templates may be nested according to the protocol layers, the lowest protocol layer forming the outer layer as shown in Figure 7-5. OVA Level 1 now contains templates for the particular protocol layer rules and also a template that instantiates all the rule templates, variables and events that are needed by the next protocol layer. As in the case of Figure 7-4, Level 1 may be removed and all the rule events, assertions and variables could be placed in the OVA units at Level 2.

The units at OVA Level 2 instantiates the global shared variable and event template (if there is any needed beyond Protocol Layer 0) and the lowest protocol layer template. To have control over which protocol layer should be included, the rule template instances and any local variables to that layer should be under parameter control, so that protocol layer checks can be eliminated if needed. This architecture requires careful planning, but it resembles more closely the protocol architecture that it is to verify.

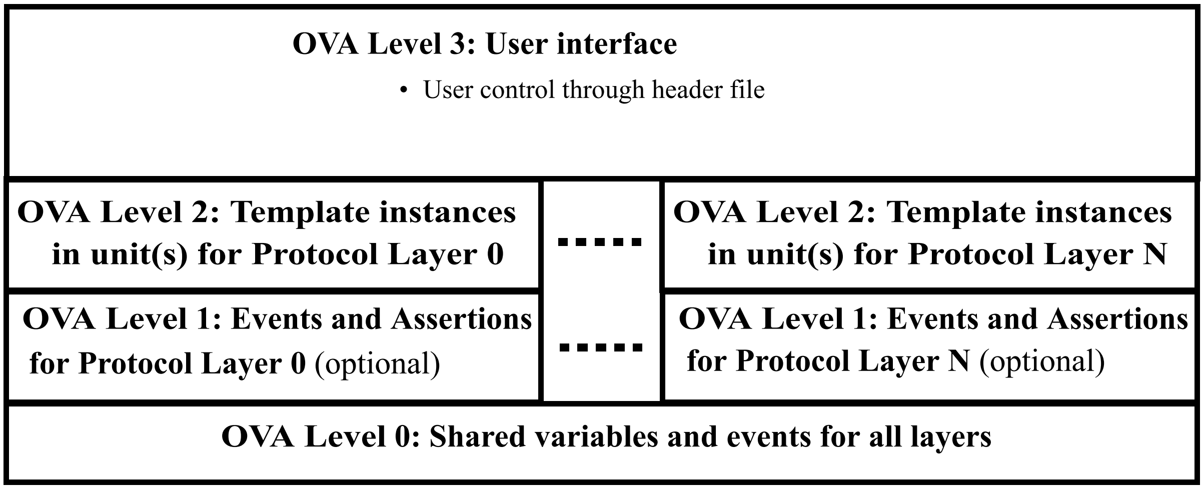
Figure 7-5 OVA checker architecture for multi-layer protocols - hierarchical structure


OVA Level 3: User interface
• User control through header file
OVA Level 2: Unit definition(s)
Instance of Protocol Layer 0 template
Instance of shared variable and event template
OVA Level 1: Protocol Layer 0
Events and assertions templates
Template with
- Rule template instances (optional)
- Protocol Layer 1 template instance
OVA Level 1: Protocol Layer 1
Events and assertions templates
Template with
- Rule template instances (optional)
- Protocol Layer 2 template instance

OVA Level 1: Protocol Layer N
Events and assertions templates
Template with
- Rule template instances (optional)
OVA Level 0: Shared variables and events for all layers

The protocol stack layers can also be instantiated in a flat structure in a unit, with all the global and local variable declarations and common definitions visible to all layers. This is illustrated in Figure 7-6. The specific protocol layer templates can now be instantiated independently in separate OVA units or all of them can be placed in one unit with selective instantiation under parameter control as in the hierarchical architecture.

Figure 7-6 OVA checker architecture for multi-layer protocols - flat structure


OVA Level 3: User interface
• User control through header file
OVA Level 2: Template instances in unit(s) for Protocol Layer 0
OVA Level 2: Template instances in unit(s) for Protocol Layer N
OVA Level 1: Events and Assertions for Protocol Layer 0 (optional)
OVA Level 1: Events and Assertions for Protocol Layer N (optional)
OVA Level 0: Shared variables and events for all layers

To date, it is this flat architecture that has been used due to its simplicity, however, better hiding of local information can be obtained with the hierarchical architecture.

The following section provides general guidelines for naming and coding OVA objects.

Naming conventions

Since reusable checkers with similar assertion names could be developed for different protocols by different creators, it is important to select a naming convention that clearly distinguishes them. Furthermore, the name should also identify the type of the object it refers to. This makes it easier to understand the meaning of object definitions and messages.

Naming conventions shall be used to prevent name conflicts

A simple rule is to name all assertions as follows:

```
<BusinessUnit><assertion_group>_<kind>_<name>
```

The kind of the object can be:

e: event
b: bool
c: assert check
cv: cover
f: assert forbid
l: logic variable
v: var variable

Note:

Variables of type `logic` should be used instead of `var`. The same functionality is provided. Logic variables must be declared outside a clock domain, however.

For example, a Synopsys generated PCI check assertion for rule "foo" could be named as follows:

```
assert SnpsApb_c_foo : check ( ... );
```

The naming convention also simplifies interpreting / sorting the simulation report.

Correspondence with the original specification shall be made clear by proper selection of names for booleans, variables and events

For example, to check last two bits of signal AD are zero as in linear burst ordering, the corresponding logic expression could be named as follows:

```
logic SynPci_l_linear_burst_ordering;
assign SynPci_l_linear_burst_ordering = (ad[1:0] ==
2'b0);
```

Each assertion shall output a short but meaningful message upon failure

The message output upon assertion failure should indicate the situation as in the rule description. The message is a quoted string passed as the second argument to the directives `check` and `forbid`. Or, preferably, a message can be output using `$display` statement following the directive. The latter form also permits to output the values of variables and signals at the time of failure. For example,

```
assert SnpsPci_c_MP39:
      check(SnpsPci_e_MP39)
           else
      $display ("Master does not ignore GNT# when RST#
is asserted");
```

Each file shall contain explanatory comments in the header

The comments shall include a brief description of the contents of the file, the name of the file, the date of creation and may include modification history, authorship and a copyright protection statement if appropriate.

Checkers shall include a severity level parameter and the corresponding argument in assert statements

If an assertion fails, the simulation could just report the error and continue or stop the simulation, etc. This behavior can be controlled by providing a severity parameter to the checker which is then passed to all assert statements that should be affected by the same behavior upon failure.

For example,

```
      unit SnpsPci_Master
      #( parameter integer severity = 0;
      ...
      )
      ( ... )

assert SnpsPci_c_name:
      check(SnpsPci_e_name,,severity)
           else
      $display( ... );
```

Checkers shall include a category level parameter and the corresponding argument in assert statements

The user may wish to have more refined control over starting, stopping and report sorting of assertions in the checker. This can be achieved by supplying a category argument to the assertions. The checker should thus have one or more category parameters that are passed to the assert statements. In the simplest case, there is only one category parameter for all the assertions, however, finer distinction could be obtained by providing different category parameters to different classes of assertions, for example, by signal direction.

For example,

```
unit SnpsPci_Master
      #( parameter integer severity = 0, category = 0;
      ...
      )
      ( ... )

assert SnpsPci_c_name:
      check(SnpsPci_e_name,,severity, category)
           else
      $display( ... );
```

Documentation and Release Items for Assertion-Based Reusable Checker Intellectual Property

The release of the OVA IP shall contain the following items

- The files containing the OVA templates and units.
- The header file with user configured macro definitions. The file may be included in all OVA files, or the macro definitions can be supplied on the compilation command line using `+define+MACRO` options, the header file just providing a list of the definitions. The choice depends on the number of such macro definition that would have to be typically included on the command line.
- All the tests used for verifying the IP.
- A README file describing the contents of the release.
- A text file with Release notes indicating any restrictions related to this particular release not documented in the main User Guide.
- A User Guide of the checker shall include the following information
- An indication of the checker documentation version and date.
- An indication of protocol version(s) and specifications covered by the checker.
- A summary of features as far as configurability, protocol subsets, and so on.
- A table of rules, separated by signal direction / device type.
- A list of macro definitions if any, indicating the default behavior and permissible values.

- A description of ports and parameters including defaults for each checker unit.
- A list of any restrictions and limitations of the checker.
- The global organization of the checker (details as far as allowed by required IP protection rules if any.)
- Layout of the directory structure in the checker distribution.
- Example of use, including sample binding statements, in block-level tests under assertion and assumption forms, and in system-level assertion only form.
- Instructions on how to compile and execute tests and how to interpret the results.

Testing

Assertions in general and reusable checkers in particular have to be thoroughly verified for compliance with the specification. The bugs can arise due to misinterpretation of the assertion language constructs, incorrect parameter values and port connections in the bind statements of checkers, typographical errors, and the like. An error in an assertion or in the deployment of a checker may result in false reporting of successful design verification. Yet, the testing problem is complex because not only the acceptance of correct protocol behavior has to be verified, but also that violations of the specification must be detected and reported by the assertion or checker.

In the case of reusable protocol checkers the space of possible violations may be quite large and a carefully prepared and reviewed verification plan is an absolute necessity for assuring acceptable quality of the checker.

There are a number of ways how to verify that a given assertion or checker complies with the requirements. Depending on the complexity of the checked behavior, the methods vary from simple to very complex involving hybrid formal tools. A possible classification of these methods is as follows:

- Visual inspection: This may be sufficient if the abstraction level of the assertion matches the natural language specification. Requires solid understanding of assertion language semantics! Typically used on simple assertions involving a few temporal operators.

  For example, the “until” assertion:

  ```
  if (c) then d*[1..] #1 e;
  ```

  It suffices to make sure that the boolean expressions `c`, `d`, and `e` are correct.

- Debugging in the DUV context: This is the easiest method to implement, but may only exercise a subset of all accepting sequences. In addition, error injection into the design (a temporary modification) is necessary to verify that the assertion rejects incorrect behaviors.

- Assertion testbenches: Write testbench code that exercises the assertion or the reusable checker. For any realistic protocol checker, the testcase development represents a major endeavor and should not be underestimated. A testcase scenario shall have the following components:

  - A typical passing sequence for the required behavior implemented as a generator in Verilog/VHDL or Vera. This represents a Basic testcase.

  - For each behavioral rule: Enumerate corner cases that should be accepted as the valid behavior. These include possible choices in timing or signal values. Write a Verilog/VHDL or Vera Positive testcase that generates a sequence covering all these corner cases. Often, such a sequence can be obtained as a variant of the Basic testcase.

  - For each behavioral rule: Enumerate the possible violations of the rule, in particular as related to the corner cases enumerated above. Write a Verilog/VHDL or Vera Negative test that generates a sequence covering all these failing corner cases. Often, such a sequence can be obtained as a variant of the Positive test.

- Constrained random stimulus generation: An interesting (yet still untested) alternative may be to use the assertion(s) in the checker as assumptions in a system model configured only using a set of wires and continuous assignments to the bus. Then, use Magellan to:

  - Generate random testcase sequences that satisfy these assumptions and observe them in a waveform viewer to check their validity. This will generate only the positive testcases.

  - Create OVA coverage statements that encode the negations of the assertions. Set these cover statements as goals in Magellan and ascertain that none of these goals is reachable, that is, the set of assertions used as assumptions cannot generate them.

- Create OVA coverage statements that represent the positive corner cases of each assertion or behavioral rule. Set these cover statements as goals in Magellan and ascertain that all of them are reachable. If not, then even though the original assertion contains the corner cases, due to other conflicting erroneous assertions (or an error in the specification itself), these corner cases cannot be reached.

The amount of work using the Magellan approach may not be any less than writing Verilog / OpenVera testbenches, however, the advantage is that it will also verify that the set of assertions are not contradictory in the ensemble and that they can be used effectively as assumptions. The limitation of this approach may be the performance limits of the tool on complex multi-layered protocols. Also, the boolean expressions and auxiliary variable assignments must be synthesizable.

8

Verification Environment

Simulation Flow

In general, the successful simulation of a testcase to completion involves the execution of the following major functions:

1. Generation of the testcase configuration.

   This includes a description of the verification environment configuration and the DUT configuration. It also includes a description of the testcase duration. It is used by the self-checking structure to determine the appropriate response to expect and by the verification environment to configure the DUT.

2. Building the verification environment around the DUT according to the generated testcase configuration.

The specific type and number of transactors that need to be instantiated around the DUT to exercise correctly may depend on the configuration that will be used. For example, a DUT may be configured with an Intel-style or a Motorola-style processor interface. Each will require a different command-layer transactor. Similarly, 16 GPIO pins may be configured as 16 1-bit interfaces or one 16-bit interface (or anything in between). Each configuration will require a different number of command-layer and functional-layer transactors and scoreboards in the self-checking structure.

3. Configuration of the DUT according to the generated test configuration.

   This may involve writing specific values to registers in the DUT or setting interface pins to specific levels.

4. Starting all transactors and generators in the environment.
5. Detection of the end-of-test conditions.
6. Stopping all generators in an orderly fashion.
7. Draining the DUT of any buffered data and downloading of accounting or statistics registers

   Any expected data left in the scoreboard is then assumed to have been lost. The value of accounting or statistics registers is compared against their expected values.

8. Reporting on the success or failure of the simulation run

Not all DUTs require all of those steps. Some steps may be trivial for some DUT. Others may be very complex. But every successful simulation follows these sequence of generic steps. Individual testcases intervene at various points in the simulation flow to implement the unique aspect of each test.

Building a Verification Environment

The `bu_env` class formalizes these simulation steps into well-defined virtual methods. These methods must be extended for each verification environment to implement the DUT-specific requirements. This class also instantiates and interconnects all transactors, generators and self-checking structures to create a complete layered verification environment around the DUT.

The following directives give guidelines to help you implement a reusable and configurable verification environment.

Extensions of the bu_env class shall implement the gen_cfg(), build(), cfg_dut_t(), start_t(), wait_for_end_t() and cleanup_t() virtual methods

These methods implement each of the generic steps that must be performed to successfully simulate a testcase. They must be overloaded to perform each step as required by the design under verification. Even if a method need to be extended for a particular DUT, it is should be extended anyway - and left empty - to explicitly document that fact.

Refer to “Environment Manager Base Class - rvm_env” on page A-387 for details on the intended semantics and purpose of each method.

Extensions of the gen_cfg(), build(), cfg_dut_t(), start_t(), wait_for_end_t() and cleanup_t() virtual methods shall call their base implementation first

The implementation of these methods in the base class manages the sequence in which these methods must be invoked. They make it unnecessary for each tests to enumerate all intermediate simulation steps. To ensure the proper automatic ordering of the simulation steps, each method extension must call their base implementation first.

```
class verif_env extends bu_env {
   ...
   virtual task wait_for_end_t() {
      super.wait_for_end_t();
      ...
   }
   ...
}
```

Extensions of the bu_env class shall not redefine the run_t() or pre_test_t() methods

These methods are not virtual because they are not intended to be specialized for a particular verification environment. They must not be redefined to prevent their semantics from being modified.

The extension of the rvm_env::gen_cfg() method shall use an embedded generator to generate the testcase configuration

This will allow tests to constrain the testcase configuration to ensure that a compatible configuration is generated, without requiring modifications to the environment or configuration descriptor.

The embedded generator will randomize an instance of the testcase configuration descriptor, as described in “Testcase Configuration Descriptor” on page 4-67.

Environment components shall be instantiated only in the rvm_env::build() method

Transactors, generators, scoreboards and functional coverage models must be instantiated according to the testcase configuration which is final only when the `rvm_env::build()` method is invoked. No transactors must be instantiated in the constructor or other methods.

The only object that is instantiated in the environment constructor is the default testcase configuration descriptor instance that will be randomize by the embedded testcase configuration generator.

All transactors and generators shall be instantiated in public properties

A testcase needs to be able to control transactors and generators as required to implement the objectives of the testcase. This can only be accomplished if the transactors and generators are publicly accessible.

Scoreboard integration callback instances shall be registered in the rvm_env::build() method

Integrating the scoreboard into the environment is part of the building process. All necessary references will exist and can be passed to the callback extension constructors if required.

Scoreboard integration callback instances shall be registered first

The scoreboard must be informed of the actual transactions that are applied to the DUT. Some callback extensions may modify the transaction descriptors (e.g. error injection) while other simply record the content of the transactions (e.g. functional coverage). By registering the scoreboard callback extensions first, it will be possible to register other extensions before or after the scoreboard callbacks depending on their effect on the transactions.

Callback extension instances that can modify or delay the transactions shall be registered before the scoreboard callback extension instances

Some callback extensions may modify or delay the transaction before it is processed by the transactor. For example, error injection callback extensions could corrupt a parity byte. By registering those callback extensions before the scoreboard callback extensions, it ensures that the scoreboard will see the actual transaction that will be executed.

Because the scoreboard callback extensions are registered first, these callback extensions are registered using the prepend argument of the rvm_xactor::register_xactor_callback() method. See “task append_callback(rvm_xactor_callbacks cb)” on page A-348.

Callback extension instances that do not modify the transactions shall be registered after the scoreboard callback extension instances

Some callback extensions do not modify the transaction and simply record its content. For example, functional coverage callback extensions may save some transaction parameters for later sampling my a coverage group. By registering those callback extensions after the scoreboard callback extensions, it ensures that the content of the transaction that was checked for correctness will be sampled.

Because the scoreboard callback extensions are registered first, these callback extensions are registered without the prepend argument of the rvm_xactor::register_xactor_callback() method. See "task append_callback(rvm_xactor_callbacks cb)" on page A-348.

The rvm_env::cfg_dut_t() method shall first reset the DUT

To ensure that the DUT is properly configured, it must be reset before register values and memory contents are modified.

The rvm_env::cfg_dut_t() method should have a fast implementation that writes to registers and memories via direct accesses

Configuring a DUT often takes a significant amount of simulation time because a relatively slow processor or serial interface is usually used to perform the register and memory updates. Once that interface has been verified to ensure that all registers and memories can be updated, it is no longer necessary to keep exercising that logic.

The DUT-specific extension of the `rvm_env::cfg_dut_t()` method should have a "fast mode" implementation, controlled by a parameter in the testcase configuration descriptor, that causes all register and memory updates to be performed via direct or API accesses, bypassing the normal processor interface.

The rvm_env::start_t() method shall start all transactors and generators

The environment should not require any additional external intervention to operate properly. All transactors and generators must be started in the extension of the `rvm_env::start_t()` method. If a testcase does not require the presence or operation of a particular transactor, it can be stopped afterward.

The rvm_env::start_t() method should not block the execution thread

Although the name of the method implies that it can block the execution thread, it should be avoided as much as possible. If the execution thread is blocked, it may be possible for a generator to start creating stimulus before a testcase has had the opportunity to shut it down.

The rvm_env::wait_for_end_t() method shall have configurable aspects

A test must be able to control how long it is going to run. It may be in terms of number of transactions to be executed or absolute time. There must be some properties in the testcase configuration descriptor, used by the `rvm_env::wait_for_end_t()` method, that control the duration of a simulation.

Functional Coverage Model

Whether or not functional coverage is used as the primary director of the verification process, functional coverage needs to be added to the verification environment to measure the thoroughness of the verification process. A functional coverage model is composed of several functional coverage groups and assertion coverage points. The bulk of the functional coverage model for a particular design under verification will be implemented as a functional aspect of the verification environment. Some test-specific functional coverage used to confirm that the objective of a test have been achieved may be implemented in the testcase itself.

This section outlines some guidelines to help implement and integrate a functional coverage model in a verification environment built according to the guidelines outlined up to now. They also help implement a functional coverage model that can be easily analyzed and correlated to the functional specification and verification plan.

A coverage model shall include stimulus coverage points

Stimulus coverage points identify all of the interesting and relevant input stimulus conditions and patterns that must be applied to the DUT. It includes concepts such as the types of transactions, the length of data packets, the presence of CRC errors, the interval between two consecutive transactions, the type of acknowledgement and the relative position of transactions on concurrent interfaces.

Stimulus coverage points can easily be correlated to generation constraints or directed testcases.

A coverage model shall include DUT response coverage points

Response coverage points identify all of the interesting and relevant responses conditions and patterns that have been produced by the DUT. It includes concepts such as the types of acknowledgement, the length of data packets, the reordering of packets, the dropping of corrupted data, the interval between two consecutive transactions and the relative position of transactions on concurrent interfaces.

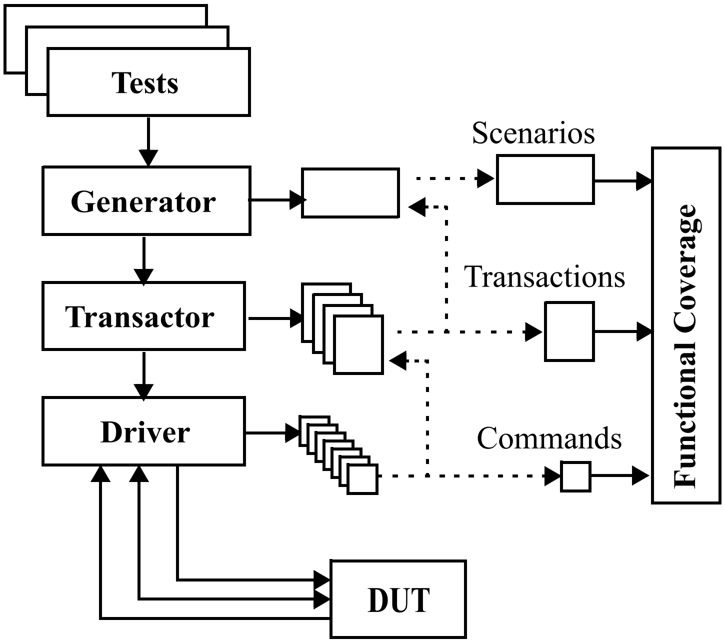
A coverage model shall include DUT state coverage points

State coverage points identify all of the interesting and relevant internal state conditions or transformation paths that have been reached by the DUT. It includes concepts such as the occupancy level of internal buffers, request and grant patterns on an arbiter, detection and handling of an exception condition or the proper segmentation or reassembly of data packets.

Stimulus coverage shall be sampled after submission to the DUT

It is not because a sequence of transaction has been generated that it will necessarily been applied, as-is, to the DUT. A transactor execution the scenario may have filtered some transactions out of the sequence or injected errors that caused some transactions to be ignored or rejected by the DUT.

Figure 8-1 Collecting Stimulus Coverage

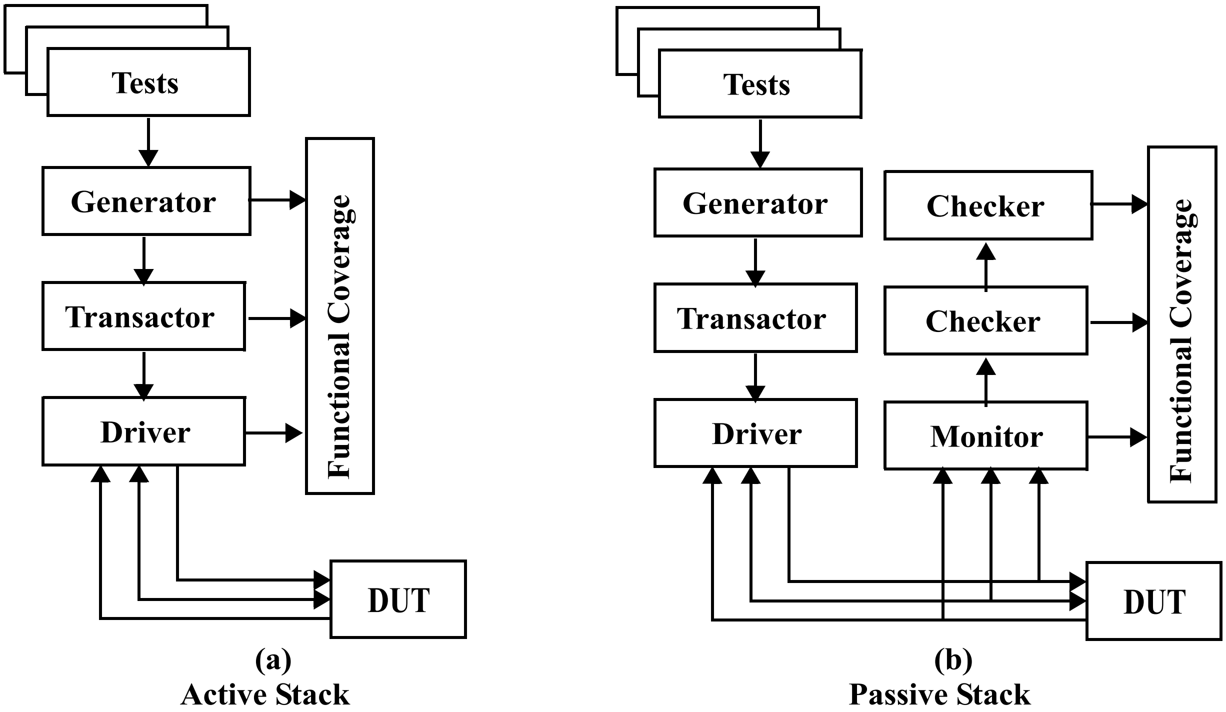


Scenarios are composed of high-level transactions which are composed of command-level transactions. As transactors execute transactions, they can relate low-level transactions back to the higher-level transaction that caused it to provide some context for the low-level transaction. As high-level transactions and scenarios are generated, they must be buffered until they have been completely implemented by the lowest layer of the environment. Low-level transactions are covered first. Using the context information in the low-level transaction, the coverage model can determine if it is the (successful) termination of the higher-level transaction or scenario it is part of. If it is, the higher level transaction is also recorded in the coverage model. If any low-level transaction is missing or indicates an improper implementation of a scenario or high-level transaction, the scenario or high-level transaction is dropped and never included in the coverage model.

Stimulus coverage should be sampled via a passive transactor stack

Figure 8-2 shows two alternatives for collecting stimulus functional coverage in a verification environment. In Figure 8-2(a), the functional coverage is collected in the generation and active transactor stack. In Figure 8-2(b), it is collected in a passive transactor stack. The latter is portable to a different verification environment that utilizes a different stimulus generation structure (such as directed testcases) or in a system-level environment where the stimulus is generated by another design block.

Figure 8-2   Collecting Stimulus Coverage



Using a passive transactor stack to collect functional coverage directly from the DUT interface ensures that only what has been observed by the DUT will be recorded in the functional coverage model. This implementation approach satisfies the guideline on only sampling stimulus coverage after the stimulus has been submitted to the DUT.

DUT state coverage points should be implemented using coverage groups instantiated in the self-checking structure

Some states of the DUT can be inferred from the operations and transformations that are necessary to perform to verify the correctness of the DUT response. If no errors were detected, it is implied that the DUT has performed similar operations or transformations. By sampling data that is representative of the relevant DUT state, state functional coverage can be implemented in the self-checking structure. For example, not adding a packet to the scoreboard because it is invalid can be sampled in the state functional coverage to infer that the DUT has rejected an invalid packet. If that packet is indeed never observed on the output (i.e. no errors are detected), the inferred state functional coverage is valid.

The self-checking structure is not a reusable component of a verification environment. Because it is so specific to a DUT, it could only be reused on a very similar design. Because of its DUT-specific nature, it will always be developed by the verification team and the source code will always be available. Since the source code is available, it can be modified to insert the DUT state functional coverage model.

DUT state coverage points may be implemented using OVA event coverage

If the DUT state information to be covered is not readily inferred from the self-checking structure, then it is necessary to sample the state information directly on the DUT. For example, the occupancy level of a buffer cannot be inferred from the scoreboard as each likely use different data structures. That information must be sampled directly on the buffer itself.

OVA events can be functionally covered to determine if they have had the opportunity to fire and if they indeed fired sometimes during a simulation. Each state coverage point is described using an OVA event. The event is not designed to perform checking and report an error if some conditions are seen or fail to materialize. It is designed to simply report to occurrence of an interesting condition in the design. By covering these events, the obtained information can form part of the DUT state functional coverage model.

Coverage groups should not be added to bu_data class extensions

The `bu_data` base class is designed to help model data and transaction descriptors. Although it may be seem natural to put the data and transaction-related coverage groups in those same classes, it creates several limitations.

There are hundred of thousands of data and transaction descriptors created and garbage collected in the course of an average simulation. A corresponding number of coverage group instances will therefore need to be created and tracked. Each coverage group will only contain coverage data for a single data or transaction instance. Individual coverage metrics will be meaningless and only cumulative coverage will be useful. It would be more efficient to collect hundreds of thousands of coverage samples in a single instance of a coverage group.

If a verification environment has multiple streams of the same data or transaction descriptors, it will be necessary to cross cover all sampled data with a stream identifier to differentiate the coverage metrics on a per-stream basis. If each stream has its own set of coverage group instances, coverage metrics for each stream will be individually collected for the thousands of data or transaction descriptor in each stream, without having to use cross coverage. Cumulative coverage can also be used to report coverage metrics regardless of the stream where the data was collected.

Functional coverage should be associated with transactors, generators, the scoreboard or the design under verification

To avoid creating a large number of coverage group instances with few coverage data in them, they should be associated with objects with a static lifetime during the simulation. Transactors, generators, scoreboards and the DUT are all created at the beginning and live until the end. By associating functional coverage groups with these verification environment components, data and transaction descriptors can be sampled as they flow through them. It will thus be possible to measure coverage for individual transactor instances and cumulative coverage for all instances of a particular transactor.

Coverage group shall be designed to produce meaningful reports, not based on the sampled data

The value of the sampled data may be an implementation convenience for a higher level concept that is being verified. The coverage group must be designed based on the higher level concept, not the data being sampled because the analysis will have to be related to that concept.

For example, a 1k-deep circular buffer needs to be functionally covered to ensure it has been thoroughly verified. It is implemented using a register file with 1024 entries, a read pointer and a write pointer. The pointer values are the only data available to be sampled.

A coverage group based on the sampled data would have 1024x1024 coverage points, one for each possible combinations of the pointer values.

The higher-level concept is the exercising of the circular buffer. To be thoroughly verified, the circular buffer must has been observed as empty, full, and neither empty nor full. Each of the occupancy levels must also be verified around the boundary conditions where the pointers wrap from the maximum address value to the lowest address value. The coverage group must therefore be coded based on buffer occupancy and the position of the pointers with respect to addresses 0x000 and 0x3FF.

```
coverage_goup circ_buffer {
   sample_event = ...;

   sample level ((’h400 + wr_ptr - rd_ptr) % ’h400) {
      state empty    (0);
      state full     (0’h3FF);
      state occupied (not state);
      cov_weight = 0;
   }
   sample rd_offset (rd_ptr) {
      state at_min (0);
      state at_max (0’h3FF);
      state middle (not state);
      cov_weight = 0;
   }
   cross goal (level, rd_offset);

   sample relation (rd_ptr) {
      state ahead  (not state) if (wr_ptr < rd_ptr);
      state behind (not state) if (wr_ptr > rd_ptr);
   }

   cov_weight = 2;
}
```

The data sampled in the verification environment or the DUT may not be the data sampled by the coverage groups

The raw data available in the verification environment may not be in a form that is suitable for creating high-level coverage points and coverage groups that will yield meaningful report. Additional transformation, combination or statistics may need to be performed to transform the raw data into something that can be classified into relevant coverage points.

For example, the coverage model for the circular buffer shown earlier could have the following data sampling interface:

```
class circ_bfr_cvr {
   local reg [9:0] rd_ptr;
   local reg [9:0] wr_ptr;
   event cover_it;

   coverage_group occupancy {
      sample_event = sync(ALL, this.cover_it) async;
      ...
   }

   task new_cvr_point() {
      do {
         @(coverage_if.bfr_ptrs) async;
      while ((^coverage_if.bfr_ptrs) === 1’bx);

      this.rd_ptr = coverage_if.bfr_ptrs[ 9: 0];
      this.wr_ptr = coveraga_if.bfr_ptrs[19:10];
      trigger(this.cover_it);
   }
}
```

Coverage groups should be implemented in coverage objects

Instantiating the coverage groups in coverage objects, it will allow the encapsulation of the transformation of the data as sampled on the verification environment or the DUT into a form that can be sampled by the functional group themselves to fill coverage points.

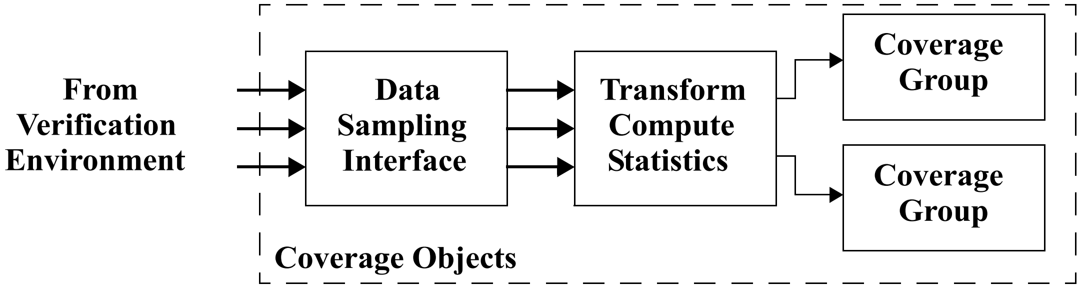
These objects may also be derived from a common base class (such a class may be added to the RVM library in the near future) to enforce standard practices such as mechanisms to turn the sampling coverage data on or off globally or partially.

The data sampling interface of the coverage object shall be designed to match the verification environment

Inserting functional coverage in a verification environment must be as unobstrusive as possible and require the minimum number of extensions. By matching the data sampling interface of the coverage object to the verification environment, it makes the integration of the coverage model in the environment that must easier.

As illustrated in Figure 8-3, any discrepancies or gap between the data sampling interface and the sampled values in the functional coverage group is bridged by transformations and computations inside the coverage object itself.

Figure 8-3 Structure of Functional Coverage Objects



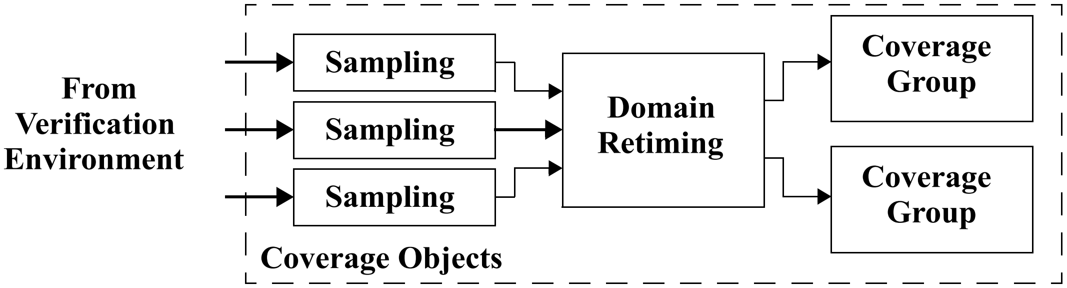
The data sampling interface should be designed to match the available data as reported by the various transactors via callback methods or in public properties.

The coverage object shall reconcile sampling domains when crossing variables sampled in different sampling domains.

The sampling mechanism built into the functional coverage groups only allows for a single sampling event to cause the sampling of all sampled variables. Because cross coverage is not allowed across coverage groups, all variables to be cross-covered must be sampled by the same coverage group - hence the same sampling event. With variables in different sampling domains, that are valid at different (and potentially asynchronous) points in time, this may yield false coverage points.

Cross covering data across sampling domains requires resolution of the sampling domain differences by the user. This can be done using arbitrary OpenVera code in a coverage object to concurrently sample, each in their own domain, all required variables then, when an appropriate window is identified, sample the intermediate values into a coverage group to be crossed. This is illustrated in Figure 8-4.

Figure 8-4 Cross Coverage Across Sampling Domain



The coverage weight of samples involved in a cross coverage should be set to 0.

If the cross coverage goal is met, it usually implies that the coverage goal of the individual samples in the cross coverage have been met. By setting their coverage weight to 0, it prevents the overall coverage score of the coverage group from being artificially raised by having some (or all of the) components of the cross coverage fully covered but not the cross coverage itself. Because the cross coverage is more representative of the true coverage goal, it alone should contribute to the coverage group score.

The coverage weight of a coverage group shall be set to the number of samples and cross with a non-zero coverage weight

By default, the coverage weight of a coverage group is the same whether the coverage group contains a single sample or dozens of large cross coverages. The latter is much more difficult to fill than the former and thus should carry a greater weight in the overall coverage rating. This guideline weighs coverage groups proportionally to the number of relevant coverage samples and crosses it contains.

The sampling frequency of a coverage group shall be minimized

Every time the sampling event of a coverage group occurs, a potentially large amount of data is sampled, expressions are computed and assigned to state bins and a database is updated. This consumes simulation bandwidth and reduces performance.

Although tools are optimized for performance, they should not be overburdened. Data for functional coverage should be sampled as few times as possible: repeatedly sampling the same value reduces performance while providing no new information. For example, avoid sampling data using a clock. Instead, use another event that is more indicative of a potential change in value of the sampled data.

Compilation Dependencies

With so many inter-related components and extensions making up an average verification component, it is easy to attempt to create circular references and dependencies that will be difficult to resolved at compile time If one designs each components in a strict bottom-up approach, they can be created without circular references or dependencies.

The following diagram shows the dependencies between classes in a well-constructed verification environment. This class hierarchy can be implemented in separate files and compiled without difficulties as there are no circular dependencies.

Figure 8-5 Compilation Dependencies in Generic Transactors

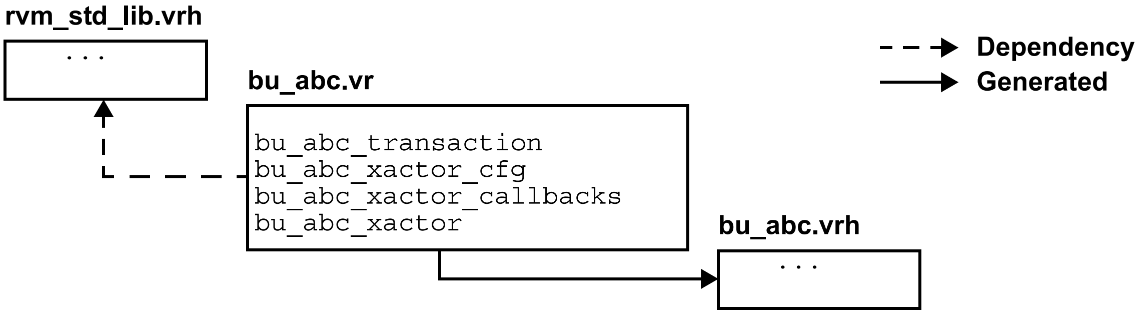


Figure 8-6 Compilation Dependencies in Testcase Configuration Descriptor

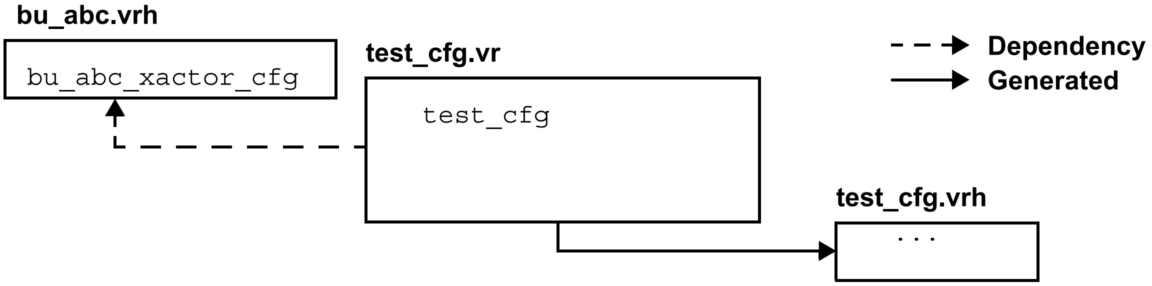


Figure 8-7 Compilation Dependencies in Scoreboard

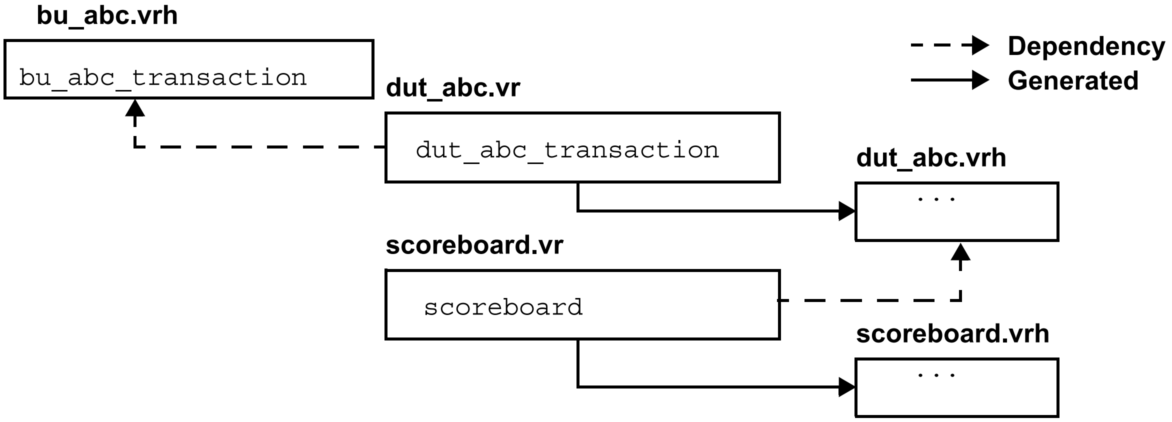
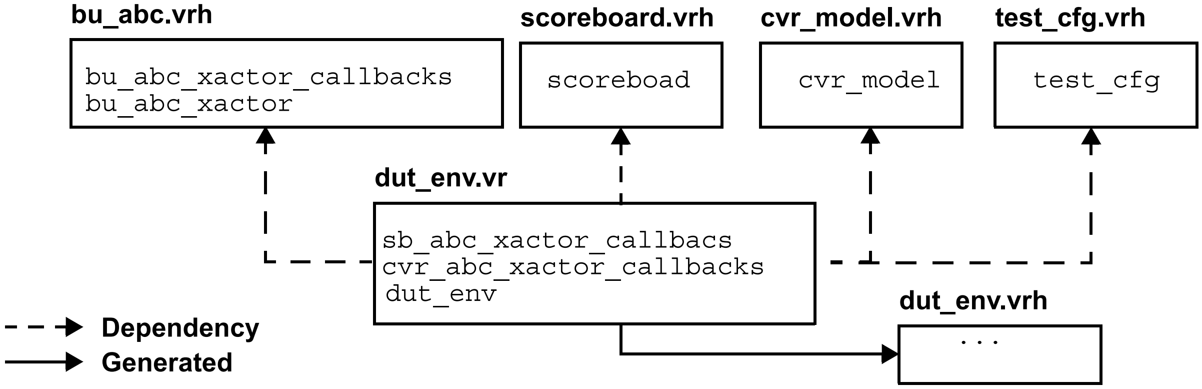


Figure 8-8 Compilation Dependencies in Verification Environment



9

Testcases

What is a Testcase?

A testcase is the top-level structure that controls the verification environment to accomplish a certain purpose:

- A testcase may be a directed simulation where the configuration, input stimulus and expected response are carefully crafted and timed to elicit the desired functionality out of the design under verification.
- A testcase may also be a set of constraints to increase the probability that the random generators in the verification environment will create certain interesting input patterns.
- A testcase may introduce synchronization between concurrent stimulus streams and interfaces to create a specific condition that must be observed by the DUT.

The default testcase runs an unconstrained simulation on the verification environment. It is the simplest testcase to write and, after initial debug of the DUT, the one that gets run with the most seeds to quickly exercise the DUT with a variety of stimulus.

```
program default_testcase {
   verify_env env = new;
   env.run_t();
}
```

Functional coverage is collected during the simulation to identify which interesting conditions were automatically generated by the default testcase. Only the uncovered functional coverage points need to be considered when creating additional testcases.

Additional testcases are variations on the default testcase. In addition to modifying the default random stimulus or providing additional stimulus or synchronization, a testcase can augment a verification environment to perform run-time operations at specific points in the simulation. A testcase can:

- Modify or specify certain aspects of the randomly generated DUT configuration.
- Inject errors in the stimulus data stream.
- Perform post-test operations and analysis.

Each individual testcase is implemented in its own layer. It is implemented as a program or a class that adds constraints or scenarios to generators in the verification environment, injects directed stimulus or performs procedural code to generate the desired stimulus patterns and synchronization, and verify the correctness of the response.

This section shows how the various elements of a testcase can be specified in the layered verification environment. Although the mechanism for implementing each of the testcase elements are individually explained, a testcase can make use of any number of elements simultaneously to achieve the desired results.

Simulating Testcases

When using random generators to create the stimulus to a design under verification, what is the best approach for meeting the coverage goals: running few, long simulations or running a large number of short simulations. And should various testcases be concatenated into a single simulation invocation, or should the simulation be restarted after every testcase?

The following guidelines will help decide on the appropriate strategy for your design.

Random testcases should be as short as possible

Short testcases reduce the time necessary to reproduce and debug a problem once it has been identified. Having to re-run an 5-hour-long simulation to debug and verify the fix of a problem does not help productivity.

A long simulation runs with a constant test configuration, whereas several short simulations will each use a different, randomly-generated test configuration. Multiple runs of a single short testcase will fill more of the functional coverage model.

Random testcases should use a biasing stimulus preamble to reach deep interesting DUT states

A short testcase may not run for long enough to allow the DUT to reach some deep internal state that must be covered. Nor it is guaranteed that running that same simulation will likely follow the long path or series of events required to reach that deep state.

Random testcases should be prefixed by a biasing stimulus preamble to put the DUT at or near an interesting state first. The random stimulus is then used to explore the solution space around that area.

Biasing stimulus shall be implemented by calling a virtual method named bias_t() in the rvm_env::start_t() method

Because the biasing stimulus preamble is not part of the testcase per se, it should be included in the `rvm_env::start_t()` method. A verification environment can be designed to add further structure to this method by creating a new virtual method. That virtual method can be extended by individual tests, or use an embedded scenario generator to randomly select a bias preamble from a set of predefined bias scenarios.

```
class dut_env extends bu_env {
   ...
   virtual protected task bias_t() {}
   virtual protected task start_t() {
      super.start_t();
      // Start the transactors
      ...
      this.bias_t();
      // Start the generators
      ...
   }
   ...
}
```

Random testcases should be run in individual simulations

Individual simulations can be parallelized and can complete an entire regression suite in a much short amount of time than by serializing them. As long as the time required to build a simulatable image of the model and verification environment, invoking the simulation and configuring the DUT is small relative to the time require to simulate the testcase itself, the overhead associated with running individual simulations will be negligible.

However, there may be cases where that overhead is significant and there is a desire to arbitrarily concatenate different tests into a single simulation. The section "Concatenating Testcases" on page 9-242 outlines additional guidelines for implementing concatenable testcases.

There shall be a single testcase program construct in a simulation

An OpenVera simulation can contain different programs but they cannot communicate or synchronize with each other. Two testcases implemented as separate programs would essentially compete with each other, without being aware of the other program's existence. Multiple programs are acceptable if they are used to model (part of) the DUT and do not interact directly with the verification environment other than through the signal layer.

In SystemVerilog, a simulation may have more than one program. A single program construct shall be used to clearly identify the top-level of the verification environment and the testcase that controls it.

Individual Testcases

It is preferable to simulate each testcase as individual simulations. The following guidelines will help implement a testcases designed to be simulated individually.

Individual testcases shall be implemented as individual programs

In OpenVera, the program is the top-level of the testbench control hierarchy. Because a testcase is implemented as the top-level of a verification environment hierarchy, it naturally maps to this construct.

```
program test_abc {
   dut_env env = new;
   env.run_t();
}
```

In SystemVerilog, the program encapsulate code that executes in the reactive region of the simulation cycle. This is the only execution region that can react to events generated by assertions and avoid race conditions with the DUT model. All verification code must eventually be located under a `program` block.

Concatenating Testcases

If the overhead to create a simulatable object of the DUT model and verification environment and starting the simulation is significant compared to the time required to simulate the actual testcase, it may be desirable to concatenate the execution of several individual testcases into a much longer simulation. However, this creates additional requirements on the testcases to help minimize the development cost of individual testcases and to debug problems identified by testcases run late in a long simulation.

The following guidelines will help implement a suitable testcase concatenation strategy for your design. The following is not currently supported in VCS-NTB)

Testcases shall not use AOP extensions or out-of-class constraint implementations (not currently supported in VCS-NTB)

These testcase implementation mechanisms are structural and apply to all instances of the affected classes and for the entire duration of the simulation. Concatenated testcases must only use procedural control and constraint mechanisms that will affect the environment for the duration of the testcase only.

```
#include "dut_env.vrh"
class test_00 {
   ...
   task run_t() {
      env.pre_test_t();
      ...
      env.run_t();
   }
}
```

Individual testcases shall be implemented as individual classes

Tests are encapsulated in individual classes and are sequenced by simply creating an instance of the desired tests and running them, one after another. The reference to the verification environment is passed to each class instance, either in the constructor or in the testcase execution method.

```
#include "dut_env.vrh"
class test_00 {
   local dut_env env;

   task new(dut_env env) {
      this.env = env;
   }
   task run_t() {
      ...
      env.run_t();
   }
}
```

Projects using a concatenated testcase simulation strategy should create a testcase base class from which all testcase implementations are derived. This will enforce a common execution model for the individual testcases and will make the creation of the sequencing program easier and more regular.

Testcases shall be sequenced in a program

The verification environment and testcases must eventually be encapsulated in a `program` construct. The program becomes the testcase sequencer for a series of testcases concatenated in a single simulation.

The rvm_env::restart() method shall be called between each concatenation

This method allows the environment to reset its simulation flow sequence. The subsequent tests can then proceed with a new simulation flow sequence, as if no other tests had been run on the environment before.

```
#include "test_00.vrh"
#include "test_01.vrh"
program test_sequence {
   dut_env env = new;

   {
      test_00 t = new(env);
      t.run_t();
   }
   env.restart(...);
   {
      test_01 t = new(env);
      t.run_t();
   }
}
```

Testcases should be designed for hard concatenation

A hard concatenation means that the test configuration, verification environment structure and DUT configuration are updated for each test in the sequence. Each concatenated test may run with a potentially different DUT configuration than the previous one.

Using hard concatenation is easier to implement as it puts less restrictions on what steps a testcase can take to execute the required operations on the verification environment. Random stability, the ability to get the same random simulation results whether the testcase is run individually or as part of a sequence, regardless of its relative position in the sequence, does not require additional steps.

Hard concatenation is implemented by specifying a TRUE value to the `reconfig` argument of the `rvm_env::restart()` method:

```
#include "test_00.vrh"
#include "test_01.vrh"
program test_sequence {
   dut_env env = new;

   {
      test_00 t = new(env);
      t.run_t();
   }
   env.restart(1);
   {
      test_01 t = new(env);
      t.run_t();
   }
}
```

Testcases may be designed for soft concatenation

A soft concatenation means that the test configuration, verification environment structure and DUT configuration remain constant for each test in the sequence. The test configuration, environment structure and DUT configuration are performed by the first test the sequence. Each subsequent test runs with the same DUT configuration as the previous one

Using soft concatenation puts additional restrictions on what steps a testcase can take to execute the required operations on the verification environment. Random stability, the ability to get the same random simulation results whether the testcase is run individually or as part of a sequence, regardless of its relative position in the sequence, also requires additional steps.

Soft concatenation is implemented by specifying a FALSE value to the `reconfig` argument of the `rvm_env::restart()` method:

```
#include "test_00.vrh"
#include "test_01.vrh"
program test_sequence {
   dut_env env = new;

   {
      test_00 t = new(env);
      t.run_t();
   }
   env.restart(0);
   {
      test_01 t = new(env);
      t.run_t();
   }
}
```

Soft Concatenation

Because the DUT is not reconfigured or reset between testcases, soft concatenation poses additional challenges to ensure that the testcases can be sequenced (or executed) in arbitrary order and are randomly stable.

The following guidelines will help implement a suitable testcase concatenation strategy for your design.

The verification environment and all transactors in the environment must implement their save_rng_state() and restore_rng_state() methods

> These methods are used to take a snapshot and then restore the random number generators in all objects and threads in the verification environment. This ensures that testcases will be randomly stable whether they are simulated individually or as part of a sequence.

Testcases shall not call the rvm_env::gen_cfg(), rvm_env::build() or rvm_env::dut_cfg_t() methods

> These methods are designed to allow testcases to modify the test configuration, environment structure or DUT configuration. Because soft-concatenated testcases use the same DU configuration, these methods are by-passed by the `rvm_env::restart()` method. To ensure that the same configuration, environment structure or DUT configuration is used when the test is simulated stand-alone or concatenated, these methods cannot be called.

Testcases should call the rvm_env::pre_test_t() method to execute test-specific code before the execution of the rvm_env::start_t() method

The `rvm_env::pre_test_t()` encapsulate the calls to `rvm_env::gen_cfg()`, `rvm_env::build()` and `rvm_env::dut_cfg_t()` (which, per the previous guideline, cannot be called directly) so they will be called if this test runs stand-alone (or is the first one in the sequence) but will not be called if this test is in a sequence.

```
#include "dut_env.vrh"
class test_00 {
   ...
   task run_t() {
      env.pre_test_t();
      ...
      env.run_t();
   }
}
```

Testcases shall reinstate default class instances in randomized properties

If a testcase replaces a randomized instance with an instance of a derived class to introduce new constraints, this replacement must be undone at the end of the test. Otherwise, subsequent tests will be executed with a different verification environment structure than if they ran individually or before the test that introduced the modification.

```
program test_X {
   verify_env env = new;

   env.pre_test_t();
   {
      my_ahb_transaction my_tr = new;
      env.ahb_src[1].randomized_tr = my_tr;
   }
```

```
      env.run_t();

      env.ahb_src[1].randomized_tr = new;
   }
```

Modifying Constraints

Instead of coding a directed testcase to hit a functional coverage point, it may be simpler to modify the constraints on the generators to increase the likelihood that they will generate the required data streams on their own. Adding, or modifying constraints that already exists, can be simple if the guidelines outlined in the section “Generator Components” on page 5-74 have been followed.

A testcase may turn the rand mode of properties ON or OFF

Because generators are always randomizing the same instance, it is possible to "remove" the rand mode on arbitrary properties which, for a particular test, must remain constant. The `rand` mode of some properties may have been turned off by default to prevent invalid data from being generated. Turning them back on and adding relevant constraints can be used to inject errors.

```
program sa_on_3 {
   verify_env env = new;

   env.build();
  env.eth_src[3].randomized_fr.sa = 48’h00083a4c453af2;
   void = env.eth_src[3].randomized_fr.rand_mode(OFF,
"sa");

   env.run_t();
}
```

This is a procedural constraint modification and can be executed at any time during the execution of a testcase.

A testcase may turn constraint blocks ON or OFF

Because generators are always randomizing the same instance, it is possible to turn constraint blocks ON or OFF using the `constraint_mode()` method. This is used to disable constraint blocks that may have been designed to prevent the injection of errors or to modify the distribution of the generated values to obtain a different distribution.

```
program bad_fcs_on_3 {
   verify_env env = new;

   env.build();
   void =
env.eth_src[3].randomized_fr.constraint_mode(OFF,
            "eth_mac_frame_valid_fcs");

   env.run_t();
}
```

This is a procedural constraint modification and can be executed at any time during the execution of a testcase.

Testcases may provide out-of-file constraint block definitions

If the definition of a randomized class contain empty constraint blocks, they can be externally defined for each testcase. This requires the pre-existence of an empty constraint block and can only be used to add constraints. The new constraint blocks are added simply by loading the VRO files that implement them.

```
constraint bu_eth_mac_frame::tc1 {
   ...
}

program test {
   verif_env env = new;
   env.start();
}
```

This is a declarative constraint modification that applies to all instances of the class. They will be taken into consideration (unless the constraint block is turned OFF) whenever an instance of the class is randomized. The constraints apply for the entire duration of the testcase execution. This mechanism cannot be used if testcase concatenation is required.

A testcase may replace randomized instances with instances of a derived class with additional constraints

It is not always possible to create the desired data stream simply by turning constraints on or off or by tweaking distribution weights. If the constraints or variable distribution weights did not exist prior, it will not be possible to create the necessary stimulus.

Because generators are always randomizing the same instance, it is possible to replace the randomized instance with an instance of a derived class. And because the `randomize()` method is virtual, the additional or overridden constraint blocks in the derived class will be used. Unlike the out-of-body constraint block implementation, this mechanism allows the addition of properties and methods and further extension of virtual methods to facilitate the expression of the required constraints.

```
class my_ahb_transaction extends ahb_transaction {
   rand integer one_hot_offset;
   constraint one_hot_address {
      one_hot_offset in {0:31};
      address == 1 << one_hot_offset;
   }
}

program test_X {
   verify_env env = new;

   env.build();
   {
      my_ahb_transaction my_tr = new;
      env.ahb_src[1].randomized_tr = my_tr;
   }

   env.run_t();
}
```

The same mechanism can be used to constrain the test configuration:

```
class duplex_test_cfg extends test_configuration {
   constraint test_Y {
      mode == DUPLEX;
   }
}

program test_Y {
   verify_env env = new;

   {
      duplex_test_cfg my_cfg = new;
      env.randomized_cfg = my_cfg;
   }

   env.run_t();
}
```

Although the class extension is declarative and global to a simulation, the substitution of the randomized instance with an instance of this new class is procedural. This constraint modification can be executed at any time during the execution of a testcase.

A testcase may add new constraints using AOP extensionsb (not currently supported in VCS-NTB)

AOP offers a mechanism that permits the addition of constraints, properties and methods and to extend predefined methods in a more succinct syntax than by using object-oriented extension.

```
extends test_X(my_ahb_transaction) {
   rand integer one_hot_offset;
   constraint one_hot_address {
      one_hot_offset in {0:31};
      address == 1 << one_hot_offset;
   }
}

program test_X {
   verify_env env = new;
   env.run_t();
}
```

This is a declarative class modification that applies to all instances of the class. The modifications will apply for the entire duration of the testcase execution. This mechanism cannot be used if test concatenation is required.

Defining or Modifying Scenarios

Instead of coding a directed testcase to hit a functional coverage point, it may be simpler to modify the constraints on an existing scenario or to defined a new scenario to increase the likelihood that the scenario generators will generate the required data streams on their own. Adding, or modifying scenarios that already exists, can be simple if the guidelines outlined in the section “Scenario Generators” on page 5-86 have been followed. Scenario definition examples can also be found in the $VERA_HOME/examples/rvm_examples/scenario directory.

The following guidelines assume that a scenario generator, defined using the rvm_scenario_generator macro (see “Scenario Generator Transactor - rvm_scenario_gen” on page A-365) is used.

Scenario-level constraints in an existing scenario descriptor may be modified using any of the previously described techniques

The instances in the `scenario_set` property are the one that will be randomized to create the scenario. The constraints in those descriptors can be modified by using any of the techniques outlined above.

Item-level constraints in an existing scenario descriptor may be modified using any of the previously described techniques

The instances in the `items` property of scenario descriptors are the one that will be randomized to create the values of the individual items in a scenario. The constraints in those instances can be modified by using any of the techniques outlined above.

Item-level constraints in an existing scenario descriptor may be modified by replacing all of the instances in its items property

The instances in the `items` property of scenario descriptors are the one that will be randomized to create the values of the individual items in a scenario. The constraints in those instances can be globally modified by replacing the content of the entire items property with copies of a factory instance. This is accomplished using the `allocate_scenario()` method in the scenario descriptor.

Example of replacing all items in a scenario descriptor:

```
class some_bad_atm_cells extends atm_cell {
   constraint hec_valid {
      hec dist {8’h00 :/ 1,  8’h01:8’hFF :/ 1};
   }
}

program try_new_scenario {
   verify_env env = new;

   env.build();
   {
      some_bad_atm_cells my_cell = new;

env.atm_src[3].scenario_set[0].allocate_scenario(my_cel
l);
   }

   env.run_t();
}
```

New scenarios must be defined by extending a scenario descriptor class

A new scenario is defined by extending a scenario descriptor class. This new scenario may replace a previously defined scenario or create a new one. The new scenario may be defined using an aspect-oriented or an object-oriented extension of an existing scenario descriptor.

Example of defining an existing scenario using an aspect-oriented extension:

```
extend atm_cell_scenario(my_new_scenario) {
   integer PROCEDURAL;

   constraint procedural_scenario {
      length == 0;
      repeated == 0;
   }

   after task new() {
      this.PROCEDURAL = define_scenario("Procedural
sequence", 0);
   }

   before virtual function integer apply_t(...) {
      if (this.kind == this.PROCEDURAL) {
         apply_t = this.procedural_scenario_t(...);
         return;
      }
   }

   function integer procedural_scenario_t(...) {
      ...
   }
}
```

Example of redefining a new scenario using an object-oriented extension:

```
class my_atm_cell_scenario extends atm_cell_scenario {
   constraint default_scenario {
      if (this.kind == this.DEFAULT) {
         length <= 3;
         repeated == 1;
      }
   }

   task new() {
    redefine_scenario(this.DEFAULT, "Stutter Scenario", 
3);
   }
}

program try_new_scenario {
   verify_env env = new;

   env.build();
   {
      my_atm_cell_scenario my_sn = new;
      env.atm_src[3].scenario_set[0] = my_sn;
   }

   env.run_t();
}
```

New scenarios may be added by adding to the available scenario set

Scenarios are selected in two stages. First, a scenario descriptor is picked from the `scenario_set` array by randomizing the `select_scenario` property. Next, the selected scenario descriptor is randomized.

This approach makes it easy sequence individual scenarios or constrain how various scenario descriptors are sequenced (e.g. never generate the same scenario twice in a row).

Example of defining a scenario as an additional scenario descriptor:

```
class my_atm_cell_scenario extends atm_cell_scenario {
   integer DEFAULT;

   constraint default_scenario {
      if (this.kind == this.DEFAULT) {
         length <= 3;
         repeated == 1;
      }
   }

   task new() {
      this.DEFAULT = define_scenario("A Scenario", 3);
   }
}

program a_test {
   dut_env env = new;

   env.build();
   {
      my_atm_cell_scenario my_sn = new;
      env.src[3].scenario_set.push_back(my_sn);
   }
   env.run_t();
}
```

Note that a scenario set may be composed of several scenario descriptors, each with different scenario kinds.

New scenarios may be added as additional kinds of an existing scenario descriptor

Scenarios are selected in two stages. First, a scenario descriptor is picked from the `scenario_set` array by randomizing the `select_scenario` property. Next, the selected scenario descriptor is randomized.

Different scenarios may be described as different kinds of scenario in the same scenario descriptor. The value of the kind property determine which scenario is actually generated. Its value is used in conditional constraint blocks to specify the scenario of that kind.

This approach makes it easy to constraint the distribution of various scenario kinds by using constraints on the scenario descriptor.

Example of defining a scenario as a kind of a scenario descriptor:

```
class my_atm_cell_scenario extends
atm_cell_atomic_scenario {
   integer A_SCENARIO;

   constraint a_scenario {
      if (this.kind == this.A_SCENARIO) {
         length <= 3;
         repeated == 1;
      }
   }

   task new() {
     this.A_SCENARIO = define_scenario("A Scenario", 3);
   }
}
```

Note that a scenario set may be composed of several scenario descriptors, each with different scenario kinds.

An integer property must be used to hold the value corresponding to the scenario kind

Scenario kinds are defined as different integer values of the `kind` property. To ensure uniqueness and consecutiveness, the actual values are managed by the scenario base class via the `defined_scenario()` method. The value assigned to the scenario kind by this method must be stored in a property so the scenario kind can be later identified.

```
class my_atm_cell_scenario extends atm_cell_scenario {
   integer A_SCENARIO;

   constraint a_scenario {
      if (this.kind == this.A_SCENARIO) {
         ...
      }
   }

   task new() {
      this.A_SCENARIO = define_scenario(...);
   }
}
```

A scenario must be specified in a separate constraint block

This will allow the scenario to be redefined in further extensions of the scenario descriptor, without affecting the other scenarios.

```
class my_atm_cell_scenario extends atm_cell_scenario {
   integer A_SCENARIO;

   constraint a_scenario {
      if (this.kind == this.A_SCENARIO) {
         ...
      }
   }
}
```

The constraint block that specifies a scenario should be named "symbol_scenario" where "symbol" is the name of the integer property identifying the scenario kind

This will make it easier to associate the constraint block specifying a scenario with the scenario itself. Furthermore, it maintains consistency with the specification of the default scenario.

```
class my_atm_cell_scenario extends atm_cell_scenario {
   integer A_SCENARIO;

   constraint a_scenario {
      if (this.kind == this.A_SCENARIO) {
         ...
      }
   }
}
```

Adding Directed Stimulus

Directed stimulus can be added for two reasons. One, the verification environment may be purely directed and not have any data generators. Second, the likelihood that a generator will produce the exact stimulus sequence to execute the desired testcase is very low.

The following guidelines will demonstrate how directed stimulus can be added to the verification environment to create a directed testcase. Not that all of these mechanisms need not be used for the entire duration of a simulation. It is possible to run a random simulation for a while, inject a directed input stream, then resume the random simulation.

The transactor immediately upstream of the directed stimulus injection point shall be stopped

Directed stimulus should not be mixed with random stimulus on the same input stream. Random data may continue to be generated on concurrent streams but the stream where the directed stimulus is injected must not inject random data in the middle of the directed sequence.

Only the transactor immediately upstream of the channel where the directed stimulus is injected needs to be stopped. Backpressure through the higher-level channels will suspend the execution of the higher level transactor and generators.

```
program directed {
   dut_env env = new;

   env.start_t()
   env.usb_device[4].stop_xactor();
   ...
}
```

Directed stimulus should be injected by using the procedural interface of a generator immediately upstream of the directed stimulus injection point

If the transactor immediately upstream of the injection point is a generator, and that generator has a directed procedural interface as described in See "Generators should provide a procedural interface to create directed transactions" on page 89, the procedural interface should be used to inject directed stimulus.

```
program directed {
   dut_env env = new;

   env.start_t()
   env.apb_master[3].stop_xactor();
   fork
   {
      ...
      env.apb_master[3].write(...);
      ...
   }
   join none
   env.run_t()
}
```

Directed stimulus may be injected in channels

Using the procedural interface requires that all properties of a transaction be entirely defined and described as argument values to the procedure call. Very often, not all elements of a transaction are purely directed and some aspects can still be randomized. In other cases, there may not be any procedural interface available. Injecting stimulus directed into a channel will bypass the callback methods of the upstream generator. In most cases, it is preferable to use the directed stimulus injection methods provided by the generator.

Directed stimulus can be created by manually instantiating data and transaction descriptor and procedurally filling their content, randomizing as necessary. The completed object is then added to the channel object at the directed stimulus injection point.

Because, according to previous guidelines, all channel references are stored in publicly accessible properties, directed stimulus can be injected at any point in the verification environment structure.

```
program directed {
   dut_env env = new;

   env.start_t()
   env.apb_master[3].stop_xactor();
   fork
   {
      apb_transaction tr;
      ...
      tr = new;
      void = tr.randomize() with {
         kind == WRITE;
      };
      env.apb_master[3].out_chan.put_t(tr);
      ...
   }
   join none
   env.run_t()
}
```

Directed scenarios shall be implemented in the scenario’s apply_t() method

One way to inject directed stimulus in the middle of random stimulus is to declare a directed scenario. The scenario becomes one of the available scenarios that will be randomly applied to the input streams where it has been made available.

A directed scenario can use procedural code to compose the data and transactions to be injected. Because the `apply_t()` method is allowed to be blocking, a directed scenario may interact with other components of the verification environment or the DUT itself. For example, a directed scenario could keep sending a series of long packets until some internal buffer is full, then stop sending any data until the buffer becomes empty.

```
class my_scenario extends apb_scenario {
   virtual protected task apply_t() {
      apb_transaction tr;
      ...
      tr = new;
      void = tr.randomize() with {
         kind == WRITE;
      };
      env.apb_master[3].out_chan.put_t(tr);
      ...
   }
}
```

Injecting Errors

Many designs have to survive and properly handle protocol and physical errors. A verification environment must be able to generate valid data and protocols but also be able to violate or corrupt them in controllable and predictable ways.

It is relatively easy to model a transactor that implements a protocol correctly. It is much harder to model a transactor that can also violate it. One way would be to build in every transactor all possible protocol failure modes. But it is unlikely that all failure modes will have been thought of, requiring constant modifications and upgrades. Secondly, it would require the creation of some language or interface to control the injection of errors according to the requirements of the testcases.

Fortunately, the callback mechanism, if rich enough, allows a transactor to be controlled in ways that may not have been originally conceived. Callback methods can be used to corrupt data, drop packets, modify a response or insert delays in a protocol.

Many of the error injection mechanisms may be implemented using OOP or AOP if an AOP extension is used, the error injection mechanism will apply to all instances of the extended class, for the entire duration of the simulation (AOP is not currently supported in VCS-NTB). An OOP extension should be used to limit the error injection to a specific portion of the test or to a single testcase in a series of concatenated testcases.

The following guidelines describe various error injection mechanisms that can be used to verify a design’s response to protocol violations. The mechanism most suitable for a testcase will depend on the error to be injected and the support offered by the transactor. Some of these mechanisms could be built in transactors to offer pre-defined error injection mechanisms.

OOP extensions of any callback method shall call the original default implementation using the super prefix

Data models, transactors, generators and verification environment may rely on functionality implemented in extensions of virtual methods. If a virtual method is extended in a testcase to inject an error, the original method implementation must be called to maintain the base original functionality.

If the callback method is a "pre" method (that is, invoked before an operation), the original implementation should be called at the end of the testcase extension. If the callback method is a "post" method (that is, invoked after an operation), the original implementation should be called first at the beginning of the testcase extension.

```
class invalid_eth_mac_frame extends bu_eth_mac_frame {
   task pre_randomize() {
      ...
      super.pre_randomize();
   }

   task post_randomize() {
      super.post_randomize();
      ...
   }
}
```

Data protection properties may be corrupted at generation time by removing or replacing the constraint that keeps them valid

Data protection properties in data objects should be kept valid by default by a constraint block (See "Data protection properties shall model their validity, not their value" on page 4-57 and "Constraint blocks shall be provided to avoid errors in randomized values" on page 4-67). Data objects with invalid CRC, FCS or checksum values can be injected by turning off the constraint block keeping the property valid, or replacing it with another one to specify a distribution or the condition for the property to be invalid.

```
class some_bad_eth_mac_frame extends bu_eth_mac_frame {
   constraint valid_fcs {
      fcs dist {9:/0, 1:/1:32'hFFFFFFFF};
   }
}

program some_bad_frames {
   dut_env env = new;

   env.build_t();
   foreach (env.eth_gen, i) {
      some_bad_eth_mac_frame my_fr = new;
      env.eth_gen[i].randomized_fr = my_fr;
   }
   env.run_t();
}
```

The integrity of properties may be corrupted at generation time by removing or replacing the constraint that keeps them coherent

Properties in data and transaction descriptors should be kept coherent by a constraint block (See "All properties corresponding to a protocol property or field shall be rand" on page 4-53 and "A constraint block shall be provided to ensure the validity of randomized property values" on page 4-65). Descriptors with invalid properties (e.g. a "length" property that does not match the actual number of bytes in a payload) can be injected by turning off the constraint block keeping the property valid, or replacing it with another one to specify a distribution or the condition for the property to be invalid.

```
class invalid_eth_mac_frame extends bu_eth_mac_frame {
   constraint valid_length {
      data.size() != length;
   }
}

program some_invalid_frames {
   dut_env env = new;

   env.build_t();
   foreach (env.eth_gen, i) {
      invalid_eth_mac_frame my_fr = new;
      env.eth_gen[i].randomized_fr = my_fr;
   }
   env.run_t();
}
```

Property values may be modified at generation time by extending the post_randomize() method

Some constraint sets used to maintain coherency of various properties may be to complex or not granular enough to violate a specific property value. Regardless of how the property values were arrived at, it is always possible to modify them via an extension of the `post_randomize()` method.

```
extends invalid_eth_mac_frame(bu_eth_mac_frame) {
   after task post_randomize() {
      length += 1;
   }
}
```

Property values may be modified in transactor "pre" callback methods

Transactors should call a callback method before processing a transaction or data (see “Transactors should call a callback method before transmitting data, allowing the user to record, modify or drop the data” on page 6-102). This callback method can be extended to modify the data or transaction before it is transmitted.

Note that it is not possible to modify the instance itself (i.e. replace the instance about to be transmitted with a new instance) if the descriptor is not supplied using a `var` argument. In that case, only the content of the data or transaction descriptor can be modified, not the instance itself.

```
extends corrupt_eth_mac_frame(bu_eth_mii) {
   before task pre_tx_frame_t(bu_eth_mac_frame fr,
                              var reg          drop) {
      randcase() {
         1: fr.fcs = 1 << random();
         9: {}
      }
   }
}
```

Transactor behavior may be modified via callback methods

Transactors should call a callback method before acting on decisions (see "Transactors should call a callback method after making a significant decision but before acting on it, allowing the user to modify the default decision" on page 6-102). This callback method can be extended to modify the default behavior of the transactor. Depending on the protocol, the callback method arguments may supply alternative responses which would not violate the protocol but would make the transaction execution sub-optimal.

```
class not_always_ack extends bu_usb_device_callbacks {
   virtual task pre_tr_ack_t(    bu_usb_device dev,
                             var bu_usb_pkt    ack,
                                bu_usb_pkt    alt[$]) {
      if (alt.size() > 0 && random % 2) {
         ack = alt.pick();
      }
   }
}
```

OOP callback extensions shall be prepended to the already-registered callbacks

A verification environment has probably already registered several callback extensions with transactors to integrate a scoreboard and collect functional coverage. When errors are injected in a transaction, that error must be observed by the scoreboard and the functional coverage metric to ensure that the correct response will be predicted and that an accurate representation of the stimulus as observed by the design will be recorded in the functional coverage model.

Because object-oriented callback extensions are called in the same order they were registered, error injections callback extension instances must be registered before any other registered callback extensions. This can be accomplished by using the `rvm_xactor::prepend_callback()` method (see “task prepend_callback(rvm_xactor_callbacks cb)” on page A-348).

```
class corrupt_eth_mac_frame extends bu_eth_mii_callbacks
{
   ...
}

program some_corrupted_frames {
   dut_env env = new;

   env.build();
   {
      corrupt_eth_mac_frame cb = new;
      foreach (env.mii, i) {
         env.mii[i].prepend_callback(cb);
      }
   }
}
```

Test Families

Certain sets of tests may require similar stimulus scenarios, synchronization mechanisms or error injection modes. Rather than duplicating similar functionality or setups and going back down the road of "one testcase per feature to be tested", it is possible to create verification environment extensions (effectively create a new verification environment based on the original one) that will be shared by a set of testcases.

This new verification environment is targeted toward a certain set of functional coverage points to be covered. Instead of writing individual tests to cover them, the necessary functionality to reach each coverage point is added to the verification environment. The selection or triggering of this additional functionality is randomized and mixed with all of the other triggering mechanisms found in the environment extension.

By randomly mixing functional aspects targeted toward related functional coverage points, it is likely that unexpected interactions will be created and potential unexpected problems identified.

The following guidelines explain how to create verification environment extensions to support a family of testcases.

Error Injection

Error injection mechanisms can be introduced by individual tests, or they may be built-in the verification environment according to the requirements of the functional coverage model or verification plan. Various kinds of errors are randomly injected at various points in the verification environment, under the control of testcases that enable or disable certain (or all) errors.

The following guidelines outline how an automated error injection mechanism can be implemented at a specific error injection point in a verification environment. An error injection point is typically located in a transactor callback method. The errors than can be injected at a particular point depend on the nature and argument of the callback method. A single transactor will likely have several potential error injection points. Each will inject a different kind of errors: for example, frame-level errors and symbol-level errors would be injected at a different level of the frame-transmission transaction. Each would be a separate implementation of the automated error injection mechanism.

An error descriptor object shall be used

An error, like a transaction, can be fully described using a descriptor class. It is a good idea (but not required) to derive the error descriptor class from the `bu_data` class. If the error descriptor is derived from the `bu_data` class, it is definitely not necessary to implement the `rvm_data::compare()`, `rvm_data::byte_size()`, `rvm_data::byte_pack()` and `rvm_data::byte_unpack()` methods as the error descriptor will never be physically transmitted nor will it be the object of a check in the scoreboard.

```
class bu_eth_mii_pre_frame_tx_error {
   ...
}
```

The error descriptor shall contain a reference to the data or transaction descriptor the error will be injected with

This will enable the expression of constraints between the error to be injected and the data or transaction that it will be injected with.

```
class bu_eth_mii_pre_frame_tx_error {
   bu_eth_mac_frame frame;
   ...
}
```

It is now possible to express constraints based on the content of the data or transaction. For example, only inject a FCS error on frames with VLAN tags.

```
class bad_fcs_on_vlan_frames
   extends bu_eth_mii_pre_frame_tx_error {
```

```
   constraint valid_fcs {

      if (frame.has_vlan) {
         fcs dist { 1:/0, 9:/1:32’hFFFFFFFF };
      } else {
         fcs == 0;
      }
   }
}
```

There shall be a "no error" error

By default, no errors must be injected. This requires that the error descriptor be able to describe a "no error" error.

```
class bu_eth_mii_pre_frame_tx_error {
   enum error_kinds (NONE, ...);
   ...
}
```

All guidelines relating the transaction descriptors shall apply

There must be a property identifying the kind of error to inject. For each kind of error, properties are used to describe the parameters of the errors. All properties must be `rand` and public.

```
class bu_eth_mii_pre_frame_tx_error {
   enum error_kinds (...);
   rand error_kinds kind;
   ...
}
```

An embedded atomic generator shall be used to generate the error to be injected

It must be possible to constrain the error injection mechanism to cause only certain types of errors to be injected only at certain times during the execution of a simulation. The atomic generator pattern (see “Atomic Generators” on page 5-79) is the standard mechanism for randomly generating descriptors in an externally controllable fashion.

```
class bu_eth_mii_pre_frame_tx_error_injection
   extends bu_eth_mii_callbacks {

   bu_eth_mii_pre_frame_tx_error randomized_error;

   virtual task pre_tx_frame_t(...) {
      this.randomized_error.frame = fr;
      void = this.randomized_error.randomize();
      ...
   }
}
```

A constraint block shall prevent the generation of errors by default

By default, no errors must be injected to maintain the base behavior of the verification environment. Because error injection is automated through an embedded random generator, a constraint must be provided to force the selection of the "no error" error by default. That constraint block is simply turned off or overridden to inject errors.

```
class bu_eth_mii_pre_frame_tx_error {
   enum error_kinds (NONE, ...);
   rand error_kinds kind;
   ...
   constraint valid_errors {
      kind == NONE;
   }
   ...
   }
```

The final value of the error descriptor shall be interpreted to perform the error injection

Eventually, the error must be injected. This is accomplished by interpreting the error descriptor and performing the corresponding actions on the arguments of the callback method.

```
   class bu_eth_mii_pre_frame_tx_error_injection
      extends bu_eth_mii_callbacks {

      bu_eth_mii_pre_frame_tx_error randomized_error;

   virtual task pre_tx_frame_t(...) {
      this.randomized_error.frame = fr;
      void = this.randomized_error.randomize();
      ...
      case (this.randomized_error.kind) {
         bu_eth_mii_pre_gframe_tx_error:NONE {}
         ...
      }
   }
}
```

A

Class Reference

This appendix details the user interface of the RVM base classes version 8.6.7.

For companies or business units adopting these classes, it is recommended that they be immediately wrapped into equivalent company or business-unit-specific base classes. Initially, these wrappers will be simple pass-through implementations. However, they will provide an opportunity for later adding generic company or bu-specific functionality or to remove a user-defined environment from possible non-compatible updates to the RVM class library. See the `examples/bu_layer` directory for an example and starting template.

Message Reporting Class - rvm_log

Public Interface

```
task new(string name, string instance,rvm_log under = null
```

Creates a new instance of an error messaging object, with the specified component name and instance name. Hierarchically, this instance is located below the specified super instance.

```
virtual task is_above(rvm_log log)
```

This instance is hierarchically above the specified error message object instance.

```
virtual function rvm_log copy(rvm_log to = null)
```

Copy the configuration of this message service instance object to the specified instance and returns a reference to the copy instance. If no instances to copy to is specified, a new one with the same name and instance name is created.

All of the message configuration information is copied, but not the hierarchical relationship information. If a new instance is created, it is not located under the logical “parent” instance of the original instance.

```
function string get_name()
function string get_instance()
```

Return the name and instance name of this instance respectively.

```
virtual task list(string name = "/./",
   string instance = "/./",
   reg recurse = 0)
```

List all message service instances that match the specified name and instance name. If the recurse parameter is TRUE (that is, non-zero), then all instances logically under the specified message service instances are also listed.

```
static integer FAILURE_TYP
static integer NOTE_TYP
static integer DEBUG_TYP
static integer REPORT_TYP
static integer NOTIFY_TYP
static integer TIMING_TYP
static integer XHANDLING_TYP
static integer PROTOCOL_TYP
static integer TRANSACTION_TYP
static integer COMMAND_TYP
static integer CYCLE_TYP
static integer USER_TYP[4]
static integer INTERNAL_TYP
static integer DEFAULT_TYP
static integer ALL_TYPES
```

Symbolic values for message types. Must be used when specifying a message type in properties or method arguments.

These symbolic values are also available as macro prefixed with `rvm_log__` (for example, `rvm_log::FAILURE_TYP` is also available as `rvm_log__FAILURE_TYP`).

```
static integer FATAL_SEV
static integer ERROR_SEV
static integer WARNING_SEV
static integer NORMAL_SEV
static integer TRACE_SEV
static integer DEBUG_SEV
static integer VERBOSE_SEV
static integer DEFAULT_SEV
static integer ALL_SEVERITIES
```

Symbolic values for message severities. Must be used when specifying a message severity in properties or method arguments. By default, message types are assigned a certain severity.

FAILURE	ERROR
NOTE	NORMAL
DEBUG	DEBUG
REPORT	DEBUG
TIMING	WARNING
XHANDLING	WARNING
PROTOCOL	DEBUG
TRANSACTION	TRACE
COMMAND	TRACE
CYCLE	VERBOSE
USER	TRACE

These symbolic values are also available as macro prefixed with `rvm_log__` (for example, `rvm_log::NOTE_SEV` is also available as `rvm_log__NOTE_SEV).`

```
virtual function rvm_log_format set_format(
rvm_log_format fmt)
```

Replace the current global message formatter with the specified message formatter and returns the formatter instance it replaces. See Message Formatting Class - rvm_log_format on page 300 for more details on how to create new message formatters.

```
virtual function string set_type_image(integer type,
string image)
```

Replace the current global image for the specified message type with the specified image. The previous image is returned.

```
virtual function string set_severity_image(
integer severity, string image)
```

Replace the current global image for the specified message severity with the specified image. The previous image is returned.

```
virtual function string format(string fmt, string prefix)
```

Note:

This method is deprecated in favor of the `rvm_log_format` class and `rvm_log::set_format()` method. See Message Formatting Class - rvm_log_format on page 300 for more details.

Replace the global message formatter instance with an instance of a pattern-based message formatter. The pattern-based message formatter formats messages according to a printf-like format pattern. The pattern is displayed as-is, except for the following escape sequences:

%T	Message type.
%S	Message severity
%i	Message numerical ID
%L	Message label.
%N	Component name.
%I	Instance name.
%t	Simulation time
%u[1]	Simulation time units
%M	The message text.
%%	A '%' character.

[1] *This placeholder is not supported in NTB. The time units may be included in the output produced by the %t placeholder using the $timeformat() system task in Verilog.*

The prefix is displayed before every additional line of message text, specified as separate calls to the `rvm_log::text()` method, to align it with any offset present in the format string. A newline character is automatically added after every message text line. The default prefix is "   " (four blank spaces).

```
virtual task register_msg_info(integer msg_id,
   integer type,
   integer severity,
   string label)
```

Specify the default type and severity associated with a message ID. Should DEFAULT be specified for either the type or severity when calling `start_msg()` for the corresponding message ID, the registered type or severity will be used. The specified label will be displayed if the message format constrains a `%L` placeholder.

```
virtual function bit start_msg(integer type,
   integer severity = DEFAULT_SEV,
   integer msg_id = -1)
```

Get ready to issue a message of the specified type, severity and ID. If messages of the specified type, severity or numerical id are currently ignored, the function returns FALSE. It returns TRUE otherwise. The INTERNAL_TYP message type is reserved for messages issued by the RVM library and should not be used.

```
virtual function bit text(string msg = null)
```

Attempt to issue the specified message. If the specified message is currently ignored, the function returns FALSE. It returns TRUE otherwise. This method specifies a single line of message text and a newline character is automatically appended when the message issued. Additional lines of messages can be produced by calling this method multiple times, once per line. Each additional line is prefixed with the specified prefix in the `rvm_log::format()` method.

A message can be flushed and terminated by calling the `end_msg()` method to trigger the message display and the simulation handling. A message can also be flushed by calling text() with a null argument. A message can be flushed multiple times but the simulation handling and notification will take effect on the first termination.

If additional lines are produced using the `printf()` task or other display mechanisms, they will not be considered by the filters, nor included in explicit log files. They may also be displayed out of order if they are produced before the message is flushed.

For single-line messages, the `rvm_note()`, `rvm_report()`, `rvm_protocol()`, `rvm_cycle()`, `rvm_debug1()`, `rvm_debug2()`, `rvm_debug3()`, `rvm_warning()`, `rvm_error()`, and `rvm_fatal()` macros can be used as a short-hand notation.

Table A-1

rvm_note(rvm_log log, string msg)	NOTE	DEFAULT
rvm_report(rvm_log log, string msg)	REPORT	DEFAULT
rvm_command(rvm_log log, string msg)	COMMAND	DEFAULT
rvm_protocol(rvm_log log, string msg)	PROTOCOL	DEFAULT
rvm_transaction(rvm_log log, string msg)	TRANSACTION	DEFAULT
rvm_cycle(rvm_log log, string msg)	CYCLE	DEFAULT
rvm_trace(rvm_log log, string msg)	DEBUG	TRACE
rvm_debug(rvm_log log, string msg)	DEBUG	DEBUG
rvm_verbose(rvm_log log, string msg)	DEBUG	VERBOSE
rvm_warning(rvm_log log, string msg)	FAILURE	WARNING
rvm_error(rvm_log log, string msg)	FAILURE	DEFAULT
rvm_fatal(rvm_log log, string msg)	FAILURE	FATAL
rvm_user(integer level, rvm_log log, string msg)	USER[level]	DEFAULT

Macros providing a short-hand notation for issuing a single-line message with typical types and severities on the specified messaging service instance.

```
virtual task end_msg()
```

Flush and terminate the current message and trigger the message display and the simulation handling. A message can be flushed multiple times but the simulation handling and notification will take effect on the first termination.

```
virtual task enable_types(integer types,
   string name = null,
   string instance = null,
   reg recursive = 0)

virtual task disable_types(integer types,
   string name = null,
   string instance = null,
   reg recursive = 0)
```

Specify the message types to be displayed/disabled by the specified instances. Instances are specified by name or regular expression using the name and instance argument. If no name and no instance are explicitly specified, this instance is implicitly specified.

If name, or instance are specified between "/" characters, then the specification is a regular expression that must be matched against all known name and instance name respectively.

The types argument specifies the messages types to enable or disable. Types are specified as the sum of all relevant types.

For safety reasons, FAILURE_TYP messages cannot be disabled. By default, all message types are enabled.

```
virtual task set_verbosity(integer severity,
   string name = null,
   string instance = null,
   reg recursive = 0)
```

Specify the minimum message severity to be displayed by the specified instances. Instances are specified by name or regular expression using the name and instance argument. If no name and no instance are explicitly specified, this instance is implicitly specified.

If name, or instance are specified between "/" characters, then the specification is a regular expression that must be matched against all known name and instance name respectively.By default, only messages with a severity of NORMAL or higher are displayed.

The default minimum severity can be changed by using the `+rvm_log_default=<sev>` runtime command line option, where `<sev>` is the desired minimum severity and is one of the following: `error`, `warning`, `normal`, `trace`, `debug` or `verbose`. The default verbosity level can be later modified using this method.

Example:

```
simv ... +rvm_log_default=debug ...
```

The minimum severity level can be globally forced by using the +`rvm_force_verbosity=<sev>` runtime command line option, where `<sev>` is the desired minimum severity and is one of the following: `error`, `warning`, `normal`, `trace`, `debug`, or `verbose`. The specified verbosity overrides the verbosity level specified using this method.

Example:

```
simv ... +rvm_force_verbosity=debug ...
```

```
virtual function integer get_verbosity()
```

Return the minimum message severity to be displayed by this instance of the message service.

```
virtual function integer modify(string name = null,
   string instance = null,
   reg recursive = 0,
   integer msg_id = -1,
   integer types = ALL_TYPES,
   integer severity = ALL_SEVERITIES,
   string text = "/./",
   integer new_type = UNCHANGED,
   integer new_severity = UNCHANGED,
   integer handling = UNCHANGED)
```

Modify the specified message on any of the specified message service instances with the new specified type, severity or simulation handling. The message can be specified by type, severity, numeric ID or by text pattern. By default, messages of any type, severity, ID or text is specified. A message must match all specified criteria. The simulation handling must be one of the following symbolic values:

static integer ABORT	Abort the simulation after issuing the message. This is the default handling for message with a FATAL severity. Callback methods may be executed before aborting.
static integer COUNT_AS_ERROR	Count the message as an error message. The simulation will abort when a user-defined maximum number of error messages have been issued. This is the default handling for message with a ERROR severity.
static integer STOP	Stop the simulation in the command-line mode. Callback methods may be executed before aborting.
static integer DEBUGGER	Stop the simulation and bring the graphical debugger window. Callback methods may be invoked before breaking into the debugger.
static integer DUMP	Display context information about the message, such as a call stack dump.
static integer CONTINUE	Take no special action

The method returns a unique modifier identifier that can be used to remove it using the `unmodify()` method. Message modifiers are applied in the same order they were defined.

These symbolic values are also available as macro prefixed with `rvm_log__` (for example, `rvm_log::ABORT` is also available as `rvm_log__ABORT`).

Using run-time message modifiers has a significant run-time impact for all messages – whether modified or not – issued by the affected message service instances. Text-based modifications have the greatest run-time impact.

```
virtual task unmodify(integer modification = -1,
   string name = null,
   string instance = null,
   reg recursive = 0)
```

Remove the specified modification from the specified message service instances. By default, all modifications are removed on the specified instances.

```
virtual task log_start(integer file,
   string name = null,
   string instance = null,
   reg recurse=0)
```

All messages produced by the named reporting objects are appended to the specified file. The file argument must be a file descriptor, as returned by the `fopen()` task. By default, all instance of the message reporting object are appended to stdout. Specifying a new output file does not stop messages from being appended to previously specified files.

Instances are specified by name or regular expression using the name and instance argument. If no name and no instance are explicitly specified, this instance is implicitly specified.

```
virtual task log_stop(integer file,
   string name = null,
   string instance = null,
   reg recurse=0)
```

Stop appending the messages produced by the named reporting objects to the specified file. The file argument must be a file descriptor, as returned by the `fopen()` task. If the file argument is stdout, messages are no longer sent to the standard simulation output and transcript. If the file argument is specified as –1, appending to all files, except stdout, is stopped.

Instances are specified by name or regular expression using the name and instance argument. If no name and no instance are explicitly specified, this instance is implicitly specified.

```
virtual task stop_after_n_errs(integer n)
```

Abort the simulation after the specified number of messages when a simulation handling of `COUNT_AS_ERROR` has been issued. This value is global and all messages from any message service instances counts toward this limit. A zero or negative value specifies no maximum.

The default value is 10.

```
task stop_after_n_errors(integer n)
```

A synonym for the stop_after_n_errs() method.

```
virtual function integer get_message_count(integer
severities,
   string name = null,
   string instance = null,
   reg recurse = 0)
```

Return the total number of messages of the specified severities that have been issued from the specified instances. Message severities can be specified as a sum of message severity to specify more than one severity.

```
virtual function integer create_watchpoint(integer msg_id =
-1,
   integer types = ALL_TYPES,
   integer severity = ALL_SEVERITIES,
   string text = "/./",
   reg issued = 1'bx)
```

Create a watchpoint descriptor that will be triggered when the specified message is used. The message can be specified by type, severity, numeric ID or by test pattern. By default, messages of any type, severity, ID or text is specified. A message must match all specified criteria. The issued parameter specifies if the watchpoint is triggered when the message is physically issued (1'b1), physically not issued that is, filtered-out (1'b0) or regardless if the message is physically issued or not (1'bx).

A watchpoint will be repeatedly triggered every time a message matching the watchpoint specification is issued by a message service instance associated with the watchpoint.

Using run-time message watchpoints has a significant run-time impact for all messages – whether watched for or not – issued by the affected message service instances.

```
virtual task add_watchpoint(integer watchpoint_id,
   string name = null,
   string instance = null,
   reg recursive = 1’b0)
```

Add the specified watchpoint to the specified message service instances. If a message matching the watchpoint specification is issued by one of the message service instance, the watchpoint will be triggered.

Using run-time message watchpoints has a significant run-time impact for all messages – whether watched for or not – issued by the affected message service instances.

```
virtual task remove_watchpoint(integer watchpoint_id,
   string name = null,
   string instance = null,
   reg recursive = 1’b0)
```

Remove the specified watchpoint from the specified message service instances.

```
virtual function rvm_log_msg wait_for_watchpoint_t(integer
wp_id)
```

Wait for the specified watchpoint to be triggered by a message issued by one of the message instances attached to this watchpoint.

```
virtual function rvm_log_msg wait_for_msg_t(string name =
null,
   string instance = null,
   reg recurse = 1’b0,
   integer msg_id = -1,
   integer types = ALL_TYPES,
   integer severity = ALL_SEVERITIES,
   string text = "/./",
   reg issued = 1'bx)
```

Set-up and wait for a one-time watchpoint for the specified message on the specified message service instance. The watchpoint is triggered only once.

```
virtual task report(string name = “/./”,
   string instance = “/./”,
   reg recurse = 1’b0)
```

Report a failure if any of the specified reporting objects have issued any error or fatal messages. Report a success otherwise.

Instances are specified by name or regular expression using the name and instance argument. If no name and no instance are explicitly specified, this instance is implicitly specified.

```
task pre_abort()
```

Aspect-oriented callback method that is invoked before the message service aborts the simulation. See the modify() method.

```
task pre_stop()
```

Aspect-oriented callback method that is invoked before the message service stops the simulation. See the modify() method.

```
task pre_debug()
```

Aspect-oriented callback method that is invoked before the message service breaks into the debugger. See the modify() method.

```
virtual task append_callback(rvm_log_callbacks cb)
```

Globally register the callback façade instance with all instance of the message service after all previously-registered callbacks. Callback methods will be invoked in the order in which they were registered.

A warning is issued if the same callback façade instance is registered more than once. Callback façade instances can be unregistered and re-registered dynamically.

```
virtual task prepend_callback(rvm_log_callbacks cb)
```

Globally register the callback façade instance with all instances of the message service before all previously-registered callbacks. Callback methods will be invoked in the reverse order in which they were registered.

A warning is issued if the same callback façade instance is registered more than once. Callback façade instances can be unregistered and re-registered dynamically.

```
virtual task unregister_callback(rvm_log_callbacks cb)
```

Globally unregister the specified callback façade instance from all instances of the message service. A warning is issued if the specified façade instance is not currently registered. Callback facade instance can be re-registered.

```
virtual task append_callback(rvm_log_callbacks cb)
```

Append the callback façade instance to the back of the list of already-registered callback instances in the message reporting service. Callback methods will be invoked in the order in which they were registered.

Callbacks are registered globally with all instances of message reporting objects. A warning is issued if the same callback façade instance is registered more than once. Callback façade instances can be unregistered and re-registered dynamically.

```
virtual task prepend_callback(rvm_log_callbacks cb)
```

Prepend the callback façade instance to the front of the list of already-registered callback instances in the message reporting service. Callback methods will be invoked in the order in which they were registered.

Callbacks are registered globally with all instances of message reporting objects. A warning is issued if the same callback façade instance is registered more than once with the same transactor. Callback façade instances can be unregistered and re-registered dynamically.

```
virtual task register_callback(rvm_log_callbacks cb,
   bit prepend = 0)
```

This method has been deprecated in favor of the `append_callback()` and `prepend_callback()` methods. See “task append_callback(rvm_xactor_callbacks cb)” on page 349 and “task prepend_callback(rvm_xactor_callbacks cb)” on page 348.

```
virtual task unregister_callback(rvm_log_callbacks cb)
```

Unregister the specified callback facade instance from the message reporting service. A warning is issued if the specified facade instance is not currently registered. The callback facade instance can be re-registered later.

Example: Issuing a Simple Message

```
class ahb_master {
   rvm_log msg;

   task new(string instance, ...) {
      this.msg = new("AHB Master", instance);
   }

   local task main_t() {
      while (1) {
         ...
          if (this.msg.start_msg(this.msg.FAILURE_TYP)) {
            void = this.msg.text(psprintf(...));
            void = this.msg.text();
         }
         ...
      }
   }
}
```

Example: Issuing a Simple Message using a Macro

```
class ahb_master {
   rvm_log msg;

   task new(string instance, ...) {
      this.msg = new("AHB Master", instance);
   }

   local task main_t() {
      while (1) {
         ...
         rvm_error(this.msg, psprintf(...));
         ...
      }
   }
}
```

Example: Issuing a Complex Message

```
class ahb_master {
   rvm_log msg;

   task new(string instance, ...) {
      this.msg = new("AHB Master", instance);
   }

   local task main_t() {
      while (1) {
         ...
         if (this.msg.start_msg(this.msg.FAILURE_TYP,
                                this.msg.WARNING_SEV)) {
            void = this.msg.text(psprintf(...));
            void = this.msg.text(psprintf(...));
            void = this.msg.text(transaction.psdisplay());
            void = this.msg.text(psprintf(...));
            void = this.msg.text(psprintf(...));
            void = this.msg.text(this.status.psdisplay());
                                 this.msg.end_msg();
         }
         ...
      }
   }
}
```

Example: Creating Hierarchical References

Create a hierarchical structure of message reporting objects for the AHB bus component.

```
class ahb_bus {
   rvm_log msg;
   ahb_master masters[*];
   ahb_slaves slaves[*];
   ahb_arbiter arb;

   task new(string instance, ...) {
      ...
      this.msg = new("AHB BUS", instance);
      for (i = 0; i < this.masters.size(); i++) {
         this.msg.is_above(this.masters[i].msg);
      }
      for (i = 0; i < this.slaves.size(); i++) {
         this.msg.is_above(this.slaves[i].msg);
      }
      this.msg.is_above(this.arb.msg);
   }
}
```

Example: Hierarchical Control

Set message verbosity level to trace for the entire structure under the top-level AHB bus component.

```
program testn {
   ...
   env.ahb.msg.set_verbosity(env.ahb.ms.g.TRACE_SEV,
      *, *, 1);
   ...
}
```

Example: Pattern-Based Message Promotion

Demote all messages with an error severity containing the pattern `abort` in all instances of message reporting objects named "AHB Master" to warning severity.

```
program testn {
   verif_env env = new;
   ...
   env.msg.modify("AHB Master", "/./", *
                  *, *, env.msg.ERROR_SEV, "/abort/",
                  *, env.msg.WARNING_SEV);
   ...
}
```

Messenger Service Callbacks - rvm_log_callbacks

This class provides a set of object-oriented callback methods that are invoked at different points by the message service. See "task append_callback(rvm_xactor_callbacks cb)" on page 349.

```
task pre_abort(rvm_log log )
```

Object-oriented callback method that is invoked before the simulation is aborted. See the modify() method.

```
task pre_stop(rvm_log log )
```

Object-oriented callback method that is invoked before the simulation is stopped. See the modify() method.

```
task pre_debug(rvm_log log )
```

Object-oriented callback method that is invoked before breaking into the debugger. See the modify() method.

Message Descriptor Class - rvm_log_msg

This class describes a message issued by a message service instance. It is returned by the `rvm_log::wait_for_watchpoint_t()` and `rvm_log::wait_for_msg_t()` method.

Public Interface

```
rvm_log log
```

A reference to the message reporting object instance which has issued the message.

```
reg [63:0] timestamp
```

The simulation time when the message was issued.

```
integer id
```

Message ID

```
integer original_type
```

Original message type as specified in the code creating the message. See the `rvm_log::FAILURE_TYP` and other symbolic values for a description of the possible values.

```
integer original_severity
```

Original message severity as specified in the code creating the message. See the `rvm_log::FATAL_SEV` and other symbolic values for a description of the possible values.

`integer effective_type`

Effective message type as potentially modified by the `modify()` method. See the `rvm_log::FAILURE_TYP` and other symbolic values for a description of the possible values.

`integer effective_severity`

Effective message severity as potentially modified by the `modify()` method. See the `rvm_log::FATAL_SEV` and other symbolic values for a description of the possible values.

`string text[$]`

Formatted text of the message. Each element of the array contains one line of text as built by individual calls to the `text()` method. Only the first body of a multi-body message is included.

`reg issued`

Indicates if the message has been physically issued or not.

`integer handling`

The simulation handling after the message is physically issued. See the `handling` parameter of the `rvm_log::modify()` method for a description of the possible values.

Message Formatting Class - rvm_log_format

This class contains methods that are invoked by the message reporting object to format messages according to a user-defined format. New message formats are implemented by extending this class then registering an instance of the class extension via the "virtual function rvm_log_format set_format(rvm_log_format fmt)" method. The default implementation of the methods implement a default message format.

Public Interface

```
virtual function string format_msg( string name,
   string instance,
   string msg_typ,
   string severity,
   integer id,
   string label,
   bit [63:0] timestamp,
   string timeunit,
   string lines[$])
```

Format the specified new message and return the formatted message as a single string value. The `lines` array contains a line of message text for each call to the `rvm_log::text()` method.

In NTB, the value of the `timeunit` argument is always the empty string. Use `%t` in a `printf()` format specifier to include the time units as specified by the last `$timeformat()` system task.

```
virtual function string continue_msg( string name,
   string instance,
   string msg_typ,
   string severity,
   integer id,
   string label,
   bit [63:0] timestamp,
   string timeunit,
   string lines[$])
```

Format the continuation of the specified message and return the formatted message continuation as a single string value. Message continuation occurs when a call to the `rvm_log::text()` method is made without any message text followed by more calls to the `rvm_log::text()` method. The lines array contains a line of message text for each call to the `rvm_log::text()` method since the last formatting of the message; it does not contain the lines of message text that have already been formatted.

In NTB, the value of `timeunit` argument is always the empty string. Use `%t` in a `printf()` format specifier to include the time units as specified by the last `$timeformat()` system task.

```
virtual function string abort_on_error( integer error_count,
   integer error_limit)
```

Return an explanation message that will be displayed before the simulation is aborted when the total number of COUNT_AS_ERROR messages exceeds the user-specifiable maximum number of errors. The message is displayed before the `rvm_log_callbacks::pre_abort()` callback methods are invoked. If null is returned, no message is displayed.

The error_count argument specifies the total number of COUNT_AS_ERROR messages that have been issued. The error_limit argument specifies the user-defined (via the `rvm_log::stop_after_n_errors()` method) maximum number of error messages.

```
virtual function string pass_or_fail( bit pass,
   string name,
   string instance,
   integer fatals,
   integer errors,
   integer warnings,
   integer dem_errs,
   integer dem_warns,
   string timeunit)
```

Return a pass/fail message. This method is invoked by the `rvm_log::report()` method after the total number of `FATAL_SEV`, `ERROR_SEV` and `WARNING_SEV` messages issued by the specified message service instances have been tabulated. It is an error to return null.

The `pass` argument, when true, indicates that no `FATAL_SEV` or `ERROR_SEV` messages were issued. The `name` and `instance` arguments are the name and instance name specified to the `rvm_log::report()` method. The `fatals`, `errors` and `warnings` arguments are the total number of `FATAL_SEV`, `ERROR_SEV` and `WARNING_SEV` messages that have been issued respectively. The `dem_errs` and `dem_warns` argument specify the number of demoted originally `ERROR_SEV` and `WARNING_SEV` messages respectively. The `timeunit` argument specifies the time unit used in the Vera shell and is used to scale the time returned by the `get_time()` function.

```
virtual function string format()
```

Return a specification of the message format produced by this instance of the message formatter is compatible with the `rvm_log::format` method. This method is required for backward compatibility with environments that dynamically modified and restored the message format using the deprecated `rvm_log::format()` method.

By default, returns "%S [%T] on %N(%I) at %t%u:\n %M".

```
virtual function string prefix()
```

Return a specification of the message prefix produced by this instance of the message formatter compatible with the `rvm_log::format` method. This method is required for backward compatibility with environments that dynamically modified and restored the message format using the deprecated `rvm_log::format()` method.

By default, returns "   " (four spaces).

Message Reporting Callback Class - rvm_log_callbacks

This class contains callback methods that are invoked by the message reporting objects. New instances of callback extensions are registered using the `rvm_log::register_callback()` method.

Public Interface

```
virtual task pre_abort(rvm_log log)
```

This callback method is invoked before the specified message reporting object executes the ABORT simulation handling.

```
virtual task pre_stop(rvm_log log)
```

This callback method is invoked before the specified message reporting object executes the STOP simulation handling.

```
virtual task pre_debug(rvm_log log)
```

This callback method is invoked before the specified message reporting object executes the DEBUG simulation handling.

Transaction Interface Class - rvm_channel

The implementation uses a macro to define a class named `rvm_obj_chan` derived from the class named `rvm_channel_class` for any user-specified class named rvm_obj, using a process similar to the `MakeVeraList` macro.

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

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

Macro Interface

The following macros are available to define type-specific channel classes.

```
rvm_channel(class_name)
```

Define a channel class to transport instances of the specified class. The transported class must be derived from the `rvm_data` class.

This macro creates the class interface and implementation. It is typically invoked in the same file where the specified class is defined and implemented.

```
extern_rvm_channel(class_name)
```

Define a channel class to transport instances of the specified class as an external class.

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

Public Interface

```
task new(string name, string instance, integer full = 1,
   integer empty = 0, reg fill_as_bytes = 0)
```

Create a new instance of a channel with the specified name, instance name and full and empty levels. If the fill_as_bytes argument is TRUE (that is, non-zero) the full and empty levels and the fill level of the channel are interpreted as the number of bytes in the channel as computed by the sum of `rvm_data::byte_size()` of all transaction descriptors in the channel, not the number of objects in the channel. If the value is FALSE (that is, zero), the full and empty levels and the fill level of the channel are interpreted as the number of transaction descriptors in the channel.

Note:

The `fill_has_bytes` feature has not yet been implemented.

```
rvm_log log
```

Message reporting object instance for messages issued from within the channel instance.

```
task reconfigure(integer full = -1,
   integer empty = -1,
   reg fill_as_bytes = 1'bx)
```

If not negative, reconfiguring the full or empty levels to the specified level. Reconfiguration may cause threads currently blocked on a `put_t()` or `put_array_t()` call to unblock. It is illegal to configure a channel with a full level lower than the empty level.

```
function integer full_level()
function integer empty_level()
```

Return the currently configured full and empty levels.

```
function integer level()
```

Return the number of transaction descriptor or the total number of bytes (as computed by the sum of `rvm_data::byte_size()`) currently in the channel, including the active slot. The interpretation of the fill level depends on the configuration of the channel instance. See `new()` and `reconfigure()`.

```
function integer size()
```

Return the number of transaction descriptors currently in the channel, including the active slot.

```
function reg is_full()
```

Returns TRUE if the number of transaction descriptors in the channel, including the active slot, is greater than or equal to the currently configured full level. Returns FALSE otherwise.

```
rvm_notify notify
```

An event notification object with the following pre-configured events:

```
static integer FULL
```

This event is indicated when the channel has reached or surpassed the specified full level. This event is triggered ON/OFF. No status is returned.

```
static integer EMPTY
```

This event is indicated when the channel has reached or underflowed the specified empty level. This event is triggered ON/OFF. No status is returned.

```
static integer PUT
```

This event is indicated whenever a new transaction descriptor is added to the channel. This event is triggered ONE_SHOT. The newly added transaction descriptor is returned as status.

```
static integer GOT
```

This event is indicated whenever an transaction descriptor is removed from the channel. This event is triggered ONE_SHOT. The newly removed transaction descriptor is returned as status.

```
static integer PEEKED
```

This event is indicated whenever an transaction descriptor is peeked from the channel. This event is triggered ONE_SHOT. The newly peeked transaction descriptor is returned as status.

`static integer ACTIVATED`

This event is indicated whenever a new transaction descriptor is transferred to the active slot. This event also implies the `PEEKED` event. This event is triggered ONE_SHOT. The newly activated transaction descriptor is returned as status.

`static integer STARTED`

This event is indicated whenever the state of an transaction descriptor in the active slot is updated to `rvm_channel_class::ACT_STARTED`. This event is triggered ONE_SHOT. The currently active transaction descriptor is returned as status.

`static integer COMPLETED`

This event is indicated whenever the state of an transaction descriptor in the active slot is updated to `rvm_channel_class::ACT_COMPLETED`. This event is triggered ONE_SHOT. The currently active transaction descriptor is returned as status.

`static integer REMOVED`

This event is indicated whenever a new transaction descriptor is removed from the active slot. This event also implies the GOT event. This event is triggered ONE_SHOT. The newly removed transaction descriptor is returned as status.

`static integer LOCKED`

This event is indicated whenever a side of the channel has been locked. This event is triggered ONE_SHOT.

```
static integer UNLOCKED
```

This event is indicated whenever a side of the channel has been unlocked. This event is triggered ONE_SHOT.

These symbolic values are also available as macro prefixed with `rvm_channel__` (for example, `rvm_channel_class::REMOVED` is also available as `rvm_channel__REMOVED`).

```
task flush()
```

Flush the content of the channel. Unblock all threads currently blocked in the `put_t()` or `put_array_t()` methods. This method will cause the FULL event to be reset or the EMPTY event to be indicated. Flushing a channel unlocks all sources and consumers.

```
function void reset()
```

This task is like `flush()`, except it also resets all internal state information in the channel. Use `reset()` when the consumer or producer threads are disabled to avoid false errors on multiple consumers or producers.

```
task sink()
```

Flush the content of the channel and sink any further objects as soon as they are added to it. Consumer threads will block and remain blocked. Producer threads will never block. The content of a sunk channel is irrevocably lost. The normal flow can be restored using the `rvm_channel::flow()` method.

```
task flow()
```

Restore the normal flow of objects through the channel. Any blocked consumer threads will be unblocked as soon as a new object instance is put into the channel. The normal flow can be interrupted again using the `rvm_channel::sink()` method.

```
task lock(reg [1:0] who)
task unlock(reg [1:0] who)
```

Block any source (consumer) as if the channel was full (empty) until explicitly unlocked. The side that is locked or unlocked is specified using the sum of the following symbolic values:

```
static reg [1:0] SOURCE
```

The producer side, the thread `calling put_t()`.

```
static reg [1:0] SINK
```

The consumer side, the thread `calling get_t()`.

For example:

```
in_channel.lock(out_channel.SOURCE);

out_channel.unlock(out_channel.SOURCE +
out_channel.SINK)
```

Locking a source does not indicate the FULL event, nor locking the sink indicate the EMPTY event.

These symbolic values are also available as macro prefixed with `rvm_channel_` (for examle, `rvm_channel_class::SINK` is also available as `rvm_channel__SINK`).

```
function reg is_locked(reg [1:0] who)
```

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

```
while (out_channel.is_locked(out_channel.SOURCE +
out_channel.SINK))
   out_channel.notify.wait_for_t(out_channel.UNLOCKED);
}
```

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

Puts the specified transaction descriptor in the channel at the specified offset. If the fill level of the channel, including the active slot, is greater than or equal to the configured full level, or if the source is locked, the task will block until the fill level of the channel is less than or equal to the configure empty level and the source is unlocked.

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

This method may cause the FULL event to be indicated or the EMPTY event to be reset.

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

Puts the specified transaction descriptor in the channel at the specified offset. This task will never block, regardless of the configured full level. Use only when a guaranteed nonblocking version of `put_t()` is required. Threads using this method must have some other means of blocking their execution.

It is an error to specify an offset that does not exist or sneak a new transaction descriptor in a locked channel.

This method may cause the FULL event to be indicated or the EMPTY event to be reset.

```
task put_array_t(rvm_data array[$],
   integer offset = -1)
```

Puts the specified transaction descriptors in the channel at the specified offset. If the number of transaction descriptors in the channel, including the active slot, is greater than or equal to the configured full level, or if the source is locked, the task will block until the number of transaction descriptors in the channel is less than or equal to the configure empty level and the source is unlocked. The transaction descriptors in the array are guaranteed to be inserted in the same order at the specified offset when there are other sources competing for the channel.

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

This method may cause the FULL event to be indicated or the EMPTY event to be reset.

```
function class_name unput(integer offset = -1)
```

Remove the specified transaction descriptor from the channel. It is an error to specify an transaction descriptor that does not exist.

This method may cause the EMPTY event to be indicated or the FULL event to be reset.

```
function class_name get_t(integer offset = 0)
```

Retrieve the next transaction descriptor in the channel at the specified offset. If the channel is empty, the function will block until an transaction descriptor is available to be retrieved. This method may cause the EMPTY event to be indicated or the FULL event to be reset.

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

```
function class_name peek_t(integer offset = 0)
```

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

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

```
function class_name activate_t(integer offset = 0)
```

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

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

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

```
function class_name active_slot()
```

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

```
function class_name start()
```

Update the status of the active slot to `rvm_channel_class::ACT_STARTED`. The transaction descriptor remains in the active slot. It is an error to call this method if the active slot is empty. This method indicates the STARTED notification in the transaction descriptor in the active slot.

Note:

A warning will be issued if the `start()`, `complete()`, and remove() methods are not invoked in the proper order.

```
function class_name complete(rvm_data status = null)
```

Update the status of the active slot to `rvm_channel_class::ACT_COMPLETED`. The transaction descriptor remains in the active slot and may be re-started. It is an error to call this method if the active slot is empty.

The optional status descriptor is supplied as status information when this method indicates the ENDED notification in the transaction descriptor in the active slot.

Note:

A warning will be issued if the `start()`, `complete()` and `remove()` methods are not invoked in the proper order.

```
function class_name remove()
```

Update the status of the active slot to `rvm_channel_class::NO_ACTIVE` and remove the transaction descriptor in the active slot from the channel. This method may cause the EMPTY event to be indicated or the FULL event to be reset.

It is not an error to call this method with an empty active.

Note:

A warning will be issued if the `start()`, `complete()`, and `remove()` methods are not invoked in the proper order but not when removing a transaction from the active slot if it has not been started.

```
function reg [1:0] status()
```

Returns one of the following symbolic values:

```
static reg [1:0] NO_ACTIVE
```

No transaction descriptor is present in the active slot.

```
static reg [1:0] ACT_PENDING
```

A transaction descriptor is present in the active slot but it has not been started yet.

```
static reg [1:0] ACT_STARTED
```

An object is present in the active slot, it has been started and it is not complete yet. The object is being processed by the downstream transactor.

```
static reg [1:0] ACT_COMPLETED
```

An object is present in the active slot and has been processed by the downstream transactor, but it has not yet been removed by the downstream transactor.

These symbolic values are also available as macro prefixed with `rvm_channel__` (for example, `rvm_channel_class::NO_ACTIVE` is also available as `rvm_channel__NO_ACTIVE`).

```
function class_name tee_t()
```

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

Used to fork off a second stream of references to the transaction descriptor stream. Note that the transaction descriptors themselves are not copied. The references returned by this method are referring to the same transaction descriptor instances obtained by the `get_t()` and `activate_t()`.

```
function reg tee_mode(reg is_on)
```

Turn the tee mode ON or OFF for this channel. Returns TRUE if the tee mode was previously ON. A thread blocked on a `tee_t()` method call will not resume execution if the tee mode is turned off. If the stream of references is not drained via the `tee_t()` method, data will accumulate in the secondary channel when the tee mode is ON.

```
task connect(rvm_channel_class downstream)
```

Connect the output of this channel instance to the input of the specified channel instance. The connection is performed with a blocking model (see section Example: Simple Blocking Interface on page 321) to communicate the status of the downstream channel to the producer interface of the upstream channel. Flushing this channel will cause the connected channel to be flushed as well. However, flushing the downstream channel will not flush this channel.

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

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

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

```
function class_name for_each(reg reset = 0)
```

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

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

```
function integer for_each_offset()
```

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

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

Display an image describing the status and content of the channel on the standard output. Each line is prefixed with the specified prefix.

```
function string display_all(string prefix = "")
```

Display the status and content of all known channel instances. This is useful for locating a missing or stray transaction descriptor in a sea of channels.

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

Returns an image describing the status and content of the channel. Each line is prefixed with the specified prefix.

```
function reg record(string filename)
```

Start recording the flow of transaction descriptors added through the channel instance in the specified file. The `rvm_data::save()` method must be implemented for that transaction descriptor and defines the file format. An transaction descriptor is recorded when added to the channel by the `put_t()` or `put_array_t()` methods.

A null filename stops the recording process.

Returns TRUE if the specified file was successfully opened.

Note: This method is not yet implemented.

```
function reg playback_t(string filename,
   class_name loader,
   reg metered = 0)
```

Lock all sources of the current channel and playback the transaction descriptors found in the specified file. The `rvm_data::load()` load method in the loader object is used and defines the file format. The transaction descriptors are added one by one in the order specified in the file. If the metered argument is TRUE, the transaction descriptors are added to the channel with the same time interval as they were originally put in.

All consumers are locked out from the channel during playback. Normal operation resumes after the data has been entirely played back.

Returns TRUE if the playback was successful.

Note: This method is not yet implemented.

Example: Declaration

```
class eth_mac_frame extends rvm_data {
   rand reg [47:0] da;
   rand reg [47:0] sa;
   rand reg [15:0] len;
   rand reg [ 7:0] data;
   rand reg [31:0] fcs;
}
rvm_channel(eth_mac_frame)   // Notice: no ';'

program test {
   eth_mac_frame_channel channel = new("ETH Frame Channel",
      "Global");
}
```

Example: External Declaration

```
typedef class eth_mac_frame;
extern_rvm_channel(eth_mac_frame)   // Notice: no ';'

program test {
   eth_mac_frame_channel channel = new("ETH Frame Channel",
      "Global");
}
```

Example: Simple Nonblocking Interface

Source thread:

```
while (1) {
   eth_mac_frame fr;
   ...
   channel.put_t(fr);
}
```

Consumer thread:

```
while (1) {
   eth_mac_frame fr = channel.get_t();
}
```

Example: Simple Blocking Interface

Source thread:

```
while (1) {
   eth_mac_frame fr;
   ...
   channel.put_t(fr);
}
```

Consumer thread:

```
while (1) {
   eth_mac_frame fr = channel.peek_t();
   ...
   void = channel.get_t();
}
```

Example: Advanced Interface

Source thread:

```
while (1) {
   eth_mac_frame fr;
   ...
   channel.put_t(fr);
   ...
   while (channel.active() != fr) {
      channel.notify.wait_for_t(channel.ACTIVATED);
   }
   ...
   channel.notify.wait_for_t(channel.COMPLETED);
}
```

Consumer thread:

```
while (1) {
   eth_mac_frame fr = channel.activate_t();
   ...
   channel.start();
   ...
   channel.complete();
   ...
   channel.remove();
}
```

Example: Out-of-Order Processing

Consumer thread:

```
while (1){
   transaction tr, i,
   integer n;
   void = in_chan.peek_t();
   tr = null;
   for (i = in_chan.for_each(1); i != null;
         i = in_chan.for_each()) {
      if (tr == null || tr.priority < i.priority) {
         tr = i;
         n = in_chan.for_each_offset();
      }
   }
   void = in_chan.activate_t(n);// value returned in this
                                //function should be same as tr
   void = in_chan.start();
   ...; //operate on transaction tr
   void = in_chan.complete();
   void = in_chan.remove();
}
```

Transaction Broadcaster Class - rvm_broadcast

Channels are point-to-point data notification objects. If multiple consumers are extracting objects from a channel, the objects are distributed between the various consumers and each of the N consumers sees 1/N objects. If a point-to-multipoint notification is required, where all consumers must see all of the objects in the stream, a broadcaster component must be used to replicate the stream of objects from a source channel to an arbitrary number of output channels. If only 2 output channels are required, the `tee_t()` method of the source channel may be used.

Individual output channels can be configured to use another reference to the source object (most efficient but the same object instance is shared by the source and all like-configured output channels) or to use a new instance copied from the source object (least efficient but uses an individual instance that can be modified without affecting other channels or the original object). A broadcaster component can be configured to use references or copies by default.

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

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

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

This class is based on the `rvm_xactor` class

Public Interface

```
task new(string name,
   string instance,
   rvm_channel_class source,
   reg use_references = 1,
   integer mode = AFAP)
```

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

See the documentation for the `broadcast_mode()` method for a description of the available modes.

```
virtual task start_xactor()
```

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

```
virtual task stop_xactor()
```

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

```
virtual task reset_xactor(integer rst_type = 0)
```

Reset this instance of the broadcaster. The input channel and all output channels are flushed.

```
virtual task broadcast_mode(integer mode);
```

Changes the broadcasting mode to the specified mode. The available modes are specified by using one of the following class-level enumerated symbolic values:

rvm_broadcast::AFAP

As Fast As Possible.
Active output channels are kept non-empty as much as possible. As soon as an active output channel becomes empty, the next object from the input channel (if available) is immediately broadcasted to all active output channels, regardless of their fill level.

This mode must not be used if the data source can generate data at a higher rate than the slowest data consumer and if broadcasted data in all output channels are not consumed at the same average rate.

rvm_broadcast::ALAP

As Late As Possible.
Data is broadcasted only when all active output channels are empty. This ensures that data is not generated any faster than the slowest of all consumer can digest it.

```
virtual function integer new_output(rvm_channel_class
channel,
   reg use_references = 1'bx)
```

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

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

Returns an identifier for the output channel that must be used to modify the configuration of the output channel.

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

```
virtual task bcast_to_output(integer output_id, ON|OFF)
```

Turn broadcasting to the specified output channel ON or OFF. By default, broadcasting to an output channel is ON. When broadcasting is turned off, the output channel is flushed and the addition of new objects from the source channel is inhibited. The addition of objects from the source channel is resumed as soon as broadcasting is turned ON.

If all output channels are OFF, the data from the input channel is constantly drained to avoid data accumulation.

Any user-extension of this method should call `super.bcast_to_output()`.

```
virtual protected function reg add_to_output(integer
decision_id,
   integer output_id,
   rvm_channel_class channel,
   rvm_data obj)
```

Overloading this method allows the creation of broadcaster components with different rules. If this function returns TRUE (that is, non-zero), the object will be added to the specified output channel. If this function returns FALSE (that is, zero), the object is not added to the channel.

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

If the channel is configured to use new instances, the obj parameter is a reference to that new instance. For efficiency reason, a new instance is created only if an instance is added to an output channel.

If objects are manually added to output channels, it is important that `rvm_channel_class::sneak()` be used to add objects in the output channel to prevent the execution thread from blocking. It is also important that FALSE be returned to prevent an object from being duplicated into an output channel.

The default implementation of this method always returns TRUE.

Transaction Scheduler Class - rvm_scheduler

Channels are point-to-point data notification objects. If multiple sources are adding objects to a channel, the objects are interleaved between the various sources in a fair but uncontrollable way. If a multipoint-to-point notification is required to follow a specific scheduling algorithm, a scheduler component must be used to identify which source stream should next be forwarded to the output stream.

This class is based on the `rvm_xactor` class

Public Interface

```
rvm_log log
```

Message service instance for this scheduler. Set by the constructor and uses the name and instance name specified in the constructor.

```
protected rvm_channel_class out_chan
```

Reference to the output channel. Set by the constructor.

```
task new(string name,
   string instance,
   rvm_channel_class destination,
   integer instance_id = -1)
```

Create a new instance of a channel scheduler object with the specified name, instance name, destination channel and optional instance identifier.

```
virtual task start_xactor()
```

Start this instance of the scheduler. The scheduler can be stopped. Any extension of this method must call `super.start_xactor()`.

```
virtual task stop_xactor()
```

Suspend this instance of the scheduler. The scheduler can be restarted. Any extension of this method must call `super.stop_xactor()`.

```
virtual task reset_xactor(integer rst_type = 0)
```

Reset this instance of the scheduler. The output channel and all input channels are flushed. If a `XACTOR_HARD_RST` reset type is specified, the scheduler election factory instance in the `randomized_sched` property is replaced with a new default instance.

```
virtual function integer new_source(rvm_channel_class
channel)
```

Add the specified channel instance as a new input channel to the scheduler.

Returns an identifier for the input channel that must be used to modify the configuration of the input channel or -1 if an error occurred.

Any user-extension of this method must call `super.new_input()`.

```
virtual task sched_from_input(integer input_id, ON|OFF)
```

Turn scheduling from the specified input channel ON or OFF. By default, scheduling from an input channel is ON. When scheduling is turned off, the input channel is not flushed and the scheduling of new objects from the source channel is inhibited. The scheduling of objects from the source channel is resumed as soon as scheduling is turned ON.

Any user-extension of this method should call `super.sched_from_input()`.

```
virtual protected function rvm_data schedule(
   rvm_channel_class sources[$], integer input_ids[$])
```

Overloading this method allows the creation of scheduling components with different rules. It is invoked for each scheduling cycle. The object returned by this method is added to the output channel. If this method returns null, no object is added for this scheduling cycle. The input channels provided in the sources argument are all of the currently non-empty ON channels. Their corresponding input ID is found in the input_ids argument.

If no object is scheduled from any of the currently non-empty channels, the next scheduling cycle will be delayed until an additional ON input channel becomes non-empty. If there are no empty input channels and no OFF channels, lock-up will occur. Scheduling cycles are attempted whenever the output channel is not full.

The default implementation of this method randomizes the instance found in the randomized_sched property

```
virtual protected function rvm_data get_object(
    rvm_channel_class source, integer input_id,
    integer offset)
```

This method is invoked by the default implementation of the `rvm_scheduler::schedule()` method to extract the next scheduled object from the specified input channel at the specified offset within the channel.

Overloading this method allows access to or replacement of the object that is about to be scheduled. User-defined extensions can be used to introduce errors by modifying the object, interfere with the scheduling algorithm by substituting a different object or recording of the schedule into a functional coverage model.

Any object that is returned by this method must either have been internally created or physically removed from the input source using the `rvm_channel_class::get_t()` method. If a reference to the object remains in the input channel (for example, by using the `rvm_channel_class::peek_t()` or `rvm_channel_class::activate_t()` method), it is liable to be scheduled more than once as the mere presence of an instance in any of the input channel makes it available to the scheduler.

```
rvm_scheduler_election randomized_sched
```

Factory instance randomized by the default implementation of the schedule method. Can be replaced with user-defined extension to modify the election rules.

Transaction Scheduler Election Class - rvm_scheduler_election

This class implements the election rules for the next scheduling cycle. The election is performed by randomizing an instance of this class. The default implementation provides a round-robin election process.

Public Interface

```
integer instance_id
```

Instance identifier of the `rvm_scheduler` class instance that is randomizing this object instance. Can be used to specified instance-specific constraints.

```
integer election_id
```

Incremented by the `rvm_scheduler` instance that is randomizing this object instance before every election cycle. Can be used to specified election-specific constraints.

`integer n_sources`

Number of sources. Equal to `sources.size()`.

`rvm_channel_class sources[$]`

Input source channels with input available to be scheduled.

`integer ids[$]`

Unique input ID corresponding to the source channels at the same index in the sources array.

`integer id_history[$]`

A list of the (up to) 10 last input Ids that were elected.

`integer next_idx`

The value to assign to source_idx to implement a round-robin election.

`rvm_data obj_history[$]`

A list of the (up to) 10 last objects that were elected.

`rand integer source_idx`

Index in the sources array of the elected source. An index of –1 indicates no election. The `rvm_scheduler_election_valid` constraint block constraints this property to be in the 0 to `sources.size()`-1.

```
rand integer obj_offset
```

Offset within the source channel indicated by the `source_idx` property of the elected object within the elected source. The `rvm_scheduler_election_valid` constraint block constraints this property to be equal to 0.

```
constraint default_round_robin
```

Implements a round-robin election process. Simply turn off this constraint block to implement a random scheduling process.

```
task post_randomize()
```

The default implementation of this method performs the round-robin election.

Event Notification Class - rvm_notify

This class implements an event notification object that can be used by other classes to meet their respective requirements.

Public Interface

```
task new(rvm_log log)
```

Create a new instance of this base class, using the specified message service instance to issue error and debug messages.

```
virtual function rvm_notify copy(rvm_notify to = null)
```

Copy the current configuration of this event notification object to the specified instance. If no instance is specified, a new one is allocated using the same message service instance as the original one. A reference to the instance copied into is returned.

Only the event configuration information is copied and merged with any pre-configured event in the destination instance. Copied event configuration will replace any pre-existing configuration for the same event ID. Event descriptors are not copied.

```
virtual function integer configure(integer event_id = -1,
   integer trigger = ONE_SHOT_TRIGGER)
```

Define a new notification event associated with the specified unique identifier. If a negative identifier value is specified, a new, unique identifier greater than 1,000,000 is returned. Unlike the OpenVera `trigger()` task, the triggering mode of an event is defined when the event is configured, using one of the `ONE_SHOT_TRIGGER`, `ONE_BLAST_TRIGGER`, `HAND_SHAKE_TRIGGER` or symbolic value.

A warning is issued if an event is configured more than once.

Events numbered 1,000,000 and higher are reserved for auto-generated event Ids. Pre-defined events in RVM base classes use number 999,999 and lower. User-defined event Ids can thus use 0 and up.

```
virtual function integer is_configured(integer event_id);
```

Checks if the specified event is currently configured. If this method returns 0, the notification is not configured. Otherwise, it returns an integer value corresponding to the current `ONE_SHOT_TRIGGER`, `ONE_BLAST_TRIGGER` or `ON_OFF_TRIGGER` configuration.

```
static integer ONE_SHOT_TRIGGER
static integer ONE_BLAST_TRIGGER
static integer HAND_SHAKE_TRIGGER
static integer ON_OFF_TRIGGER
```

Specify the event triggering mechanism when the event is configured. Each symbolic value corresponds to the OpenVera triggering mechanism ONE_SHOT, ONE_BLAST, HAND_SHAKE and ON respectively.

These symbolic values are also available as macro prefixed with rvm_notify__ (for example, rvm_notify::ONE_SHOT_TRIGGER is also available as rvm_notify__ONE_SHOT_TRIGGER).

```
virtual function reg is_on(integer event_id)
```

Check if the execution thread will be suspended if the wait_for_t() method is called. If this method returns TRUE, the thread will NOT be suspended. If this method returns FALSE, the thread will be suspended.

A value of TRUE can only be returned on event configured as ON_OFF_TRIGGER events. A warning is issued if this method is called on any other types of events.

```
virtual function rvm_data wait_for_t(integer event_id)
```

Suspend the execution thread until the specified event is notified. It is an error to specify an unconfigured event. The function returns any status description object specified in the event notification.

```
virtual function rvm_data wait_for_off_t(integer event_id)
```

Suspend the execution thread until the specified ON_OFF event is reset. It is an error to specify an unconfigured or a non-ON_OFF event. The function returns any status description object that was specified in the event notification.

```
virtual function reg is_waited_for(integer event_id)
```

Check if a thread is currently waiting for the specified event. It is an error to specify an unconfigured event. The function returns TRUE if there is a thread known to be waiting for the specified event.

Note that the knowledge about the number of threads waiting for a particular event is not (currently) definitive. As threads call the `rvm_notify::wait_for_t()` or `rvm_notify::wait_for_off_t()` methods, the fact that they are waiting for the event is recorded. Once the event is indicated and each thread returns from the method call, the fact that they are no longer waiting is also recorded. But if the threads are externally terminated via the terminate statement or a timeout, the fact that they are no longer waiting is not recorded. In this case, it is up to the terminated threads to report that they are no longer waiting by calling the `rvm_notify::terminate()` method.

When an instance is reset with a hard reset, no threads are assumed to be waiting for any event.

```
virtual function terminated(integer event_id)
```

Indicate to the event notification object that a thread waiting for the specified event has been terminated and is no longer waiting.

```
virtual function reg [63:0] timestamp(integer event_id)
```

Return the simulation time when the specified event was last notified. It is an error to specify an unconfigured event.

```
virtual task indicate(integer event_id,
   rvm_data status = null,
   reg handshake = 0)
```

Notify the specified event with the optional status description. HAND_SHAKE triggers only one sync, even if multiple syncs are waiting for triggers. It triggers the most recent pending sync, or queues requests. If the order of triggering the unblocking of a sync is important, use a semaphore_get() and semaphore_put() around the sync to maintain the desired order. If a sync has already been called and is waiting for a trigger, the HAND_SHAKE trigger unblocks the sync. If no sync has been called when the trigger occurs, the HAND_SHAKE trigger is stored. When a sync is called, the sync is immediately unblocked and the trigger is removed.

```
virtual task set_event(integer event_id,
   rvm_notify_event event = null)
```

Define the specified event using the specified event descriptor instance. If the descriptor is null, the event is undefined and can only be indicated using the indicate() method. If an event is already defined, the new definition replaces the previous definition.

```
virtual function  rvm_notify_event get_event  (integer
event_id)
```

Get the event descriptor instance associated with the specified event, if any. If no event descriptor is associated with the specified event, null is returned.

```
virtual task reset(integer event_id = -1, reg hard = 0)
```

Reset the specified event. A soft reset clears the specified ON/OFF event and re-starts the `rvm_notify_event::indicate_t()` and `rvm_notify_event::reset_t()` methods on any attached event descriptor. A hard reset clears all status information and attached event descriptor on the specified event and assumes that no threads are waiting for an event. If no event is specified, all events are reset.

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

Display the value returned by the `rvm_notify::psdisplay()` method as a `NOTE_TYP` message with a `NORMAL_SEV` severity.

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

Returns a description of all notifications currently configured in this notification object.

Example: Defining Three User-Defined Events

```
class bus_mon extends rvm_xactor {

   static integer EVENT_A = 0;
   static integer EVENT_B = 1;
   static integer EVENT_C = 2;

   task new() {
      super.new();
      super.notify.configure(this.EVENT_A);
      super.notify.configure(this.EVENT_B,
         super.notify.ON_OFF_TRIGGER);
      super.notify.configure(this.EVENT_C,
         super.notify.ONE_BLAST_TRIGGER);
   }
}
```

Example: Attaching Event Definition to Event

This example uses the event definition shown in “Example: Event Defined as Indicated when Two Other Events are Indicated”.

```
class bus_mon extends rvm_xactor {

   static integer EVENT_A = 0;
   static integer EVENT_B = 1;
   static integer EVENT_C = 2;

   task new() {
      super.new();
      super.notify.configure(this.EVENT_A);
      super.notify.configure(this.EVENT_B,
         super.notify.ON_OFF_TRIGGER);
      super.notify.configure(this.EVENT_C,
         super.notify.ONE_BLAST_TRIGGER);

      {
         a_and_b_event AB = new(super.notify,
            this.EVENT_A, this.EVENT_B);
         super.notify.set_event(this.EVENT_C, AB);
      }
   }
}
```

Event Definition Base Class - rvm_notify_event

This class implements an event descriptor object that can be used to autonomously trigger notification events. Event descriptors are attached to notification events using the `rvm_notify::set_event()` method.

Public Interface

```
virtual function rvm_data indicate_t()
```

Define a method that, when it returns cause the event attached to the descriptor instance to be indicated. The return value is used as the indicated event status information. This method is automatically invoked by the `rvm_notify` instance when an event definition is attached to an event.

This method must be overloaded in user-defined class extensions to implement user-defined arbitrarily complex event indication expressions or monitors.

```
virtual task reset_t()
```

Define a method that, when it returns cause the event attached to the descriptor instance, if configured as an ON/OFF event, to be reset. This method is automatically invoked by the `rvm_notify` instance when an event definition is attached to a persistent event.

This method must be overloaded in user-defined class extensions to implement user-defined event reset expressions or monitors.

Example: Event Defined as Indicated when Two Other Events are Indicated

```
class a_and_b_event extends rvm_notify_event {

   local rvm_notify notify;
   local integer a;
   local integer b;

   task new(rvm_notify notify, integer a, integer b) {
      this.notify = notify;
      this.a = a;
      this.b = b;
   }

   virtual function rvm_data indicate_t() {
      fork
         void = this.notify.wait_for_t(a);
         void = this.notify.wait_for_t(b);
      join
   }
}
```

Data Object Base Class - rvm_data

Public Interface

`task new(rvm_log log)`

Create a new instance of this object with the specified static message service instance. The specified message service instance is used when constructing the `rvm_data::notify` property.

Because of the potentially large number of instances of data objects, a static message service instance should be used to minimize memory usage and to be able to control class-generic messages:

```
class my_data extends rvm_data {
   static rvm_log log = new("rvm_data", "class");
   task new() {
      super.new(this.log);
   }
}
```

The constructor of classes derived from this base class should be callable without any arguments to be able to use the predefined atomic (see “Atomic Generator Transactor - rvm_atomic_gen” on page 357) and scenario generators (see “Scenario Generator Transactor - rvm_scenario_gen” on page 365).

```
function rvm_log set_log(rvm_log log)
```

Replace the message service instance for this instance of the object with the specified message service instance and return the old message service instance. This method can be used to associate a descriptor with the message service interface of a transactor currently processing the transaction, or to set the service instance when it was not available during initial construction.

```
integer stream_id
integer scenario_id
integer object_id
```

Unique identifiers for a randomized instance. They specify the offset of the object with a sequence and the sequence offset within a stream. These properties are set by the generator before randomization. They can be used to specify conditional constraints to create scenarios and sequences of scenarios.

```
rvm_notify notify

static integer EXECUTE
static integer STARTED
static integer ENDED
```

An event notification object with three pre-configured events. The `EXECUTE` event is ON/OFF and triggered by default. It should be used to prevent the execution of a transaction or the transmission of a data object if reset. The `STARTED` and `ENDED` events are ON/OFF events and indicate the start and end of the transaction or data transfer. The meaning and location of the events is specific to the transactor executing the transaction described by this instance.

These symbolic values are also available as macro prefixed with `rvm_data__` (for example, `rvm_data::STARTED` is also available as `rvm_data__STARTED`).

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

Display the value of the object instance in a human-readable format on the standard output. Each line of the output will be prefixed with the specified prefix. The default implementation simply prints the value returned by the psdisplay() method.

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

Return an image of the value of the object instance in a human-readable format as a string. The string may contain newline characters to split the image across multiple lines. Each line of the output will be prefixed with the specified prefix.

```
virtual function reg is_valid(reg silent = 1,
   integer kind = -1)
```

Check if the content of the object instance is valid and error-free, according to the optionally specified kind or format. The meaning (and use) of the kind argument is specific and defined by the user-extension of this method.

Returns TRUE (that is, non-zero) if the content of the object is valid. Returns FALSE otherwise.

If silent is TRUE (that is, non-zero), no error or warning messages are issued if the content is invalid. If silent is FALSE, warning or error messages may be issued if the content is invalid.

```
virtual function rvm_data allocate()
```

Allocate a new instance of the same type as the object instance. Returns a reference to the new instance. Useful to create instances of user-defined derived class in generic code written using the base class type.

```
virtual function rvm_data copy(rvm_data to = null)
```

Copy the value of the object instance to the specified object instance. If no target object instance is specified, a new instance is allocated. Returns a reference to the target instance.

Note that the following trivial implementation will not work. The objects instantiated in the object (such as those referenced by the log and notify properties) are not copied and both copies will share references to the same instances. Furthermore, it will not properly handle the case when the to argument is not null.

```
function rvm_data atm_cell::copy(rvm_data to = null) {
   copy = new this;
}
```

The following implementation is usually preferable:

```
function rvm_data atm_cell::copy(rvm_data to = null) {
   atm_cell cpy;

   if (to != null) {
      if (!cast_assign(cpy, to, CHECK)) {
         rvm_fatal(log, "Not a atm_cell instance");
         return;
      }
   } else cpy = new;

   this.copy_data(cpy);
   cpy.vpi = this.vpi;
   ...
   copy = cpy;
}
```

The base class implementation of this method must not be called as it contains error detection code of derived class that forget to supply an implementation. The `copy_data()` method should be called instead.

```
virtual protected task copy_data(rvm_data to)
```

Copy the value of all data properties in the current data object into the specified data object instance. This method should be called by the implementation of the `copy()` method in classes immediately derived from this base class.

```
virtual function reg compare(rvm_data to,
   var string diff,
   integer kind = -1)
```

Compare the value of the object instance with the value of the specified object instance, according to the specified kind. Returns TRUE (that is, non-zero) if the value is identical. If the value is different, FALSE is returned and a descriptive text of the first difference found is put in the specified string variable. The `kind` argument may be used to implement different comparison functions (for example, full compare, comparison of `rand` properties only, comparison of all properties physically implemented in a protocol, and so on)

If this method is implemented using `object_compare()`, the value of the diff argument will have to be set to a generic error description as the `object_compare()` method does not provide any indication of the mis-compares found.

```
virtual function integer byte_pack(var reg [7:0] bytes[*],
   integer offset = 0,
   integer kind = -1)
```

Pack the content of the object into the specified dynamic array of byte, starting at the specified offset. The array is resized appropriately. Returns the number of bytes packed.

If the data can be interpreted or packed in different ways, the kind argument can be used to specify which interpretation or packing to use.

```
virtual function integer byte_unpack(reg [7:0] bytes[*],
   integer offset = 0,
   integer kind = -1)
```

Unpack the data in the specified dynamic array into this object, starting at the specified offset. Returns the number of bytes unpacked. If there is not enough data in the dynamic array, the remaining properties are set to X's.

If the data can be interpreted or unpacked in different ways, the kind argument can be used to specify which interpretation or packing to use.

```
virtual function integer byte_size(integer kind = -1)
```

Returns the number of bytes required to pack the content of this object. More efficient than packing the object since no packing is actually done.

If the data can be interpreted or packed in different ways, the kind argument can be used to specify which interpretation or packing to use.

```
virtual task save(integer file)
```

Append the content of this object to the specified file. The format is user defined and may be binary.

```
virtual function reg load(integer file)
```

Set the content of this object from the data in the specified file. The format is user defined and may be binary.

Should return FALSE (that is, zero) if the loading operation was not successful.

Transactor Base Class - rvm_xactor

Public Interface

```
task new(string name,
   string instance,
   integer stream_id = -1)
```

Create an instance of the transactor base class, with the specified name, instance name and optional stream identifier. The name and instance name are used to create the message service object instance in the `rvm_xactor::log` property and the specified stream ID is used to initialize the `rvm_xactor::stream_id` property.

```
function string get_name()
function string get_instance()
```

Return the name and instance name of this instance respectively.

```
rvm_log log
```

Message reporting object instance for messages issued from within the transactor instance.

```
integer stream_id
```

Unique identifier for the stream of data objects flowing through this transactor instance. It is used to set the `stream_id` property of the objects as they are received or randomized by this transactor.

```
task append_callback(rvm_xactor_callbacks cb)
```

Register the callback facade instance for this instance of the transactor after all previously-registered callbacks. Callback methods will be invoked in the order in which they were registered.

A warning is issued if the same callback facade instance is registered more than once with the same transactor. A facade instance can be registered with more than one transactor. Callback facade instances can be unregistered and re-registered dynamically.

```
task prepend_callback(rvm_xactor_callbacks cb)
```

Register the callback facade instance for this instance of the transactor before all previously registered callbacks. Callback methods will be invoked in the reverse order in which they were registered.

A warning is issued if the same callback facade instance is registered more than once with the same transactor. A facade instance can be registered with more than one transactor. Callback facade instances can be unregistered and re-registered dynamically.

```
task append_callback(rvm_xactor_callbacks cb)
```

Append the callback facade instance to the back of the list of already-registered callback instances in this transactor. Callback methods will be invoked in the order in which they were registered.

A warning is issued if the same callback facade instance is registered more than once with the same transactor. A facade instance can be registered with more than one transactor. Callback facade instances can be unregistered and re-registered dynamically.

```
task prepend_callback(rvm_xactor_callbacks cb)
```

Prepend the callback facade instance to the front of the list of already-registered callback instances in this transactor. Callback methods will be invoked in the order in which they were registered.

A warning is issued if the same callback facade instance is registered more than once with the same transactor. A facade instance can be registered with more than one transactor. Callback facade instances can be unregistered and re-registered dynamically.

```
task register_callback(rvm_xactor_callbacks cb,
   bit prepend = 0)
```

This method has been deprecated in favor of the append_callback() and prepend_callback() methods. See "task append_callback(rvm_xactor_callbacks cb)" on page 349 and "task prepend_callback(rvm_xactor_callbacks cb)" on page 348.

```
task unregister_callback(rvm_xactor_callbacks cb)
```

Unregister the specified callback facade instance for this instance of the transactor. A warning is issued if the specified facade instance is not currently registered with the transactor. Callback facade instances can be re-registered with the same or another transactor.

```
rvm_notify notify

static integer XACTOR_IDLE
static integer XACTOR_BUSY
static integer XACTOR_STARTED
static integer XACTOR_STOPPED
static integer XACTOR_RESET
```

Event notification object and pre-configure events to indicate the state and state transitions of the transactor. The XACTOR_IDLE and XACTOR_BUSY events are ON/OFF events. All other events are ON_SHOT. These events are indicated by the rvm_xactor::start_xactor(), rvm_xactor::reset_xactor(), rvm_xactor::wait_if_stopped_t(), rvm_xactor::wait_if_stopped_or_empty_t(), and rvm_xactor::next_transaction_t() methods.

These symbolic values are also available as macro prefixed with rvm_xactor__ (for example, rvm_xactor::XACTOR_STARTED is also available as rvm_xactor__XACTOR_STARTED).

```
virtual task start_xactor()
```

Starts the execution threads in this instance of the transactor. The transactor can later be stopped. Indicates the `XACTOR_STARTED` and `XACTOR_BUSY` events and resets the `XACTOR_IDLE` event. Any extension of this method must call `super.start_xactor()`.

```
virtual task stop_xactor()
```

Stops the execution threads in this instance of the transactor after the currently executing transaction has been completed. Any extension of this method must call `super.stop_xactor()`. The execution of the transactor is actually stopped when the execution thread invokes the `rvm_xactor::wait_if_stopped_t()` or `rvm_xactor::wait_if_stopped_or_empty_t()` method.

```
virtual task reset_xactor(integer rst_type =
XACTOR_FIRM_RST)
```

Resets the state and the execution threads in this instance of the transactor according to the specified reset type (in increasing value) and indicates the `XACTOR_RESET` and `XACTOR_IDLE` events and resets the `XACTOR_BUSY` event :

```
static integer XACTOR_SOFT_RST
```

Clears the content of all channels, resets all persistent events and aborts all execution threads but maintain the current configuration, notifier and random number generation state information. The transactor must be restarted. This reset type must be implemented.

```
static integer XACTOR_FIRM_RST
```

Like soft reset, but resets all notifier and random number generation state information. This reset type must be implemented.

If random stability is required when running consecutive tests in the same simulation with soft restarts (see the `rvm_env::restart()` method), all randomized instances in generation transactors should be reset to default instances.

```
static integer XACTOR_PROTOCOL_RST
```

Equivalent to a reset signaled via the physical interface. Refer to the transactor documentation for details of the reset information. The transactor must be restarted.

Not all reset types may be implemented by all transactors. Any extension of this method must call `super.reset_xactor(rst_type)` first to terminate the `main_t()` thread, the reset notify member and reset the main thread seed according to the specified reset type. Calling `super.reset_xactor()` with a reset type of `rvm_xactor::XACTOR_PROTOCOL_RST` is functionally equivalent to `rvm_xactor::XACTOR_SOFT_RST`.

These symbolic values are also available as macro prefixed with `rvm_xactor__` (for example, `rvm_xactor::XACTOR_FIRM_RST` is also available as `rvm_xactor__XACTOR_FIRM_RST`).

```
static integer XACTOR_HARD_RST
```

Reset the instance to the same state found at the start of the simulation. The registered callbacks are unregistered.

```
virtual task kill_xactor()
```

Invoke the `reset_xactor()` method then kill all threads in the transactors so that any memory associated with those threads can be garbage collected. Any extension of this method must call `super.kill_xactor()`.

A killed transactor cannot be restarted.

```
virtual task save_rng_state()
```

This method only need to be implemented in a user extension of the `rvm_xactor` class if random stability is required between consecutive tests separated by a soft restart (see the `rvm_env::restart()` method). This method should save, in local properties, the state of the object random generator in all instantiated objects in this transactor.

```
virtual task restore_rng_state()
```

This method only need to be implemented in a user extension of the `rvm_xactor` class if random stability is required between consecutive tests separated by a soft restart (see the `rvm_env::restart()` method). This method should restore, from local properties, the state of the object random generator in all instantiated objects in this transactor.

```
virtual task xactor_status(prefix = "")
```

Display the current status of the transactor instance in a human readable format using the messaging instance. Each line of the status information is prefixed with the specified prefix.

```
protected task
wait_if_stopped_or_empty_t(rvm_channel_class chan)
```

Blocks the thread execution if the transactor has been stopped via the `stop_xactor()` method or if the specified input channel is empty. Indicates the `XACTOR_IDLE` event and resets the `XACTOR_BUSY` event. If the transactor has been stopped, indicates the `XACTOR_STOPPED` as well. This method returns once the transactor has been restarted (if it was stopped) using the `start_xactor()` method and the specified input channel is not empty. It indicates the `XACTOR_BUSY` event and resets the `XACTOR_IDLE` event. It also indicates the `XACTOR_STARTED` if the transactor was restarted.

This method superceeds the `rvm_xactor::next_transaction_t()` method. This method does not block nor indicates any event if the transactor is not stopped and the specified input channel is not empty.

```
protected virtual task main_t()
```

This task is forked off whenever the `start_xactor()` method is called. It is aborted whenever the `reset_xactor()` method is called. The functionality of a user-defined transactor must be implemented in this method. Subthreads may be started within this method. It can have a blocking or non-blocking implementation.

Any extension of this method must fork a call to `super.main_t()`.

All threads must be started in this method, not in the constructor. This will help ensure random stability when multiple tests, separated by soft restarts (see the `rvm_env::restart()` method) are sequenced in the same simulation.

```
protected function rvm_data
next_transaction_t(rvm_channel_class chan)
```

This method is deprecated and superseded by the rvm_xactor::wait_if_stopped_or_empty_t() method. See “protected task wait_if_stopped_or_empty_t(rvm_channel_class chan)” on page 353.

Blocks the thread execution while the transactor been stopped via the stop_xactor() method or the specified data notification channel is empty and indicate XACTOR_IDLE event and reset the XACTOR_BUSY event. Returns the next object removed from the data notification channel as soon as an object is available and the transactor is started, and indicates the XACTOR_BUSY event and resets the XACTOR_IDLE event. This method does not block if the transactor is not stopped and the channel is not empty.

The return value of this function must be appropriately assigned using cast_assign().

This method is suitable for single-input, variable-rate transactors with a non-blocking completion model.

Macros

rvm_OO_callback(type_name, method(args))

This macro simplifies the syntax of invoking object-oriented callback methods in a transactor. For example, instead of:

```
   this.pre_tr_t(tr, drop);
   foreach (this.callbacks, i) {
      ahb_master_callbacks cb;
      if (!cast_assign(cb, this.callbacks[i], CHECK))
   continue;
   cb.pre_tr_t(this, tr, drop);
}
```

Use:

```
this.pre_tr_t(tr, drop);
rvm_OO_callback(ahb_master_callbacks,
   pre_tr_t(this, tr, drop));
```

Example: Template for a User-Defined Implementation

```
class ahb_master_callbacks extends rvm_xactor_callbacks {
   virtual task pre_tr_t(ahb_master xactor,
                         ahb_transaction tr,
                         var reg  drop)
   {}
}

class ahb_master extends rvm_xactor{
   rvm_log msg;
   ahb_transaction_channel to_ahb;

   task new(string instance, ...) {
      this.msg = new("AHB Master", instance);
      ...
   }

   task pre_tr_t(ahb_transaction tr,
                 var reg drop) {}

   protected virtual task main_t() {

      fork
         super.main_t();
      join none

      while (1) {
         ahb_transaction tr;
         reg drop = 0;
         this.wait_if_stopped_or_empty_t(chan);
         cast_assign(tr, chan.activate_t());

         this.pre_tr_t(tr, drop);
         rvm_OO_callback(ahb_master_callbacks,
                         pre_tr_t(this, tr, drop));
```

```
            if (drop) {
               chan.remove();
               continue;
            }

            if (this.msg.start_msg(this.msg.DEBUG_TYP,
                                   this.msg.TRACE_SEV)) {
               void = this.msg.text(
                  "Executing transaction...");
               void = this.msg.text(tr.psdisplay("   "));
               this.msg.end_msg();
            }
            chan.start();
            ...
         }
      }
   }
```

Transactor Callbacks Base Class - rvm_xactor_callbacks

This class implements a base class for callback containments.See "task prepend_callback(rvm_xactor_callbacks cb)" on page 348 and "task append_callback(rvm_xactor_callbacks cb)" on page 349.

Atomic Generator Transactor - rvm_atomic_gen

The implementation uses a macro to define a class named rvm_obj_atomic_gen for any user-specified class named rvm_obj, using a process similar to the rvm_channel macro. The constructor of the generated class must be callable without arguments.

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

For more details on how to use this class, see “Atomic Generators” on page 79 and “Modifying Constraints” on page 249.

Macro Interface

The following macros are available to define type-specific atomic generator transactor classes. These macro defines classes named `class_name_atomic_gen` and `class_name_atomic_gen_callbacks`.

`rvm_atomic_gen(class_name, “Class Description”)`

> Define an atomic generator class to generate instances of the specified class. The generated class must be derived from the `rvm_data` class and the `rvm_obj_channel` class must exist.
>
> This macro creates the class interface and implementation. It is typically invoked in the same file where the specified class is defined and implemented.

`extern_rvm_atomic_gen(class_name)`

> Define an atomic generator class to generate instances of the specified class as an external class. The `rvm_obj_channel` class must be visible.
>
> This macro creates an external class declaration and no implementation. It is typically invoked when the channel class must be visible to the compiler but the actual channel class declaration is not yet available.

Public Interface

These public interface elements are in addition to the public interface elements provided by the rvm_xactor base class. Please refer to "Public Interface" on page 347 for additional interface elements.

```
task new(string instance,
   integer stream_id = 1,
   rvm_obj_channel out_chan = null)
```

Create a new instance of an atomic generator object with the specified instance name and optional stream identifier. The generator can be optionally connected to a specified output channel instance. If no output channel instance is specified, one will be created internally in the out_chan property.

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

```
class_name_channel out_chan
```

Reference to the output channel for the instances generated by this transactor. The output channel may have been specified via the constructor. If no output channel instances were specified, a new instance is automatically created. This reference in this property may be dynamically replaced but the generator should be stopped (see "virtual task stop_xactor()" on page 324) during the replacement.

```
integer stop_after_n_insts
```

The generator will stop (see “virtual task stop_xactor()” on page 324) after the specified number of object instances has been generated and consumed by the output channel. The generator must be reset (see “virtual task reset_xactor(integer rst_type = XACTOR_FIRM_RST)” on page 351) before it can be restarted (see “virtual task start_xactor()” on page 324). If the value of this property is 0, the generated will not stop on its own.

The default value of this property is 0.

```
class_name randomized_obj
```

Object instance that is repeatedly randomized to create the random content of the output object stream. The individual instances of the output stream are copied from this object, after randomization, using the `copy()` method (see “virtual function rvm_data copy(rvm_data to = null)” on page 344).

The atomic generator uses a factory pattern to generate the output stream instances. The generated stream can be constrained using various techniques on this property. See “Modifying Constraints” on page 249 for more details.

The object_id property of the generated instance is incremented after each generation. It will be reset to 0 when the generator is reset (see “virtual task reset_xactor(integer rst_type = XACTOR_FIRM_RST)” on page 351) and after the specified number of instances has been generated.

```
integer GENERATED
```

Event identifier for the `rvm_notify` event notification object in the `notify` property provided by the `rvm_xactor` base class. It is configured as a ONE_SHOT event and is indicated immediately before an instance is added to the output channel. The generated instance is specified as the status of the event.

```
integer DONE
```

Event identifier for the `rvm_notify` event notification object in the `notify` property provided by the `rvm_xactor` base class. It is configured as an ON_OFF event and is indicated when the generator stops because the specified number of instance has been generated. No status information is specified.

```
virtual function reg inject_t(class_name obj)
```

Inject the specified instance in the output stream. Unlike injecting the instance directly in the output channel, it counts toward the number of instances generated by this generator and will be subjected to the callback methods. The method returns once the instance has been consumed by the output channel or it has been dropped by the callback methods. If the instance has been dropped, this function returns FALSE (that is, zero). TRUE (that is, non-zero) is returned if the instance is forwarded to the output channel.

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

```
task post_inst_gen_t(class_name obj,
   var reg drop)
```

Aspect-oriented callback method invoked by the generator after a new object instance has been created but before it is added to the output channel and before the object-oriented callback methods (see “virtual task post_inst_gen_t(class_name_atomic_gen gen, class_name obj, var reg drop)” on page 364) are invoked.

The obj argument refers to the newly generated instance which can be modified. If the value of the drop argument is set to non-zero, the generated instance will not be forwarded to the output channel.

Example: Declaration

```
class eth_mac_frame extends rvm_data {
   rand reg [47:0] da;
   rand reg [47:0] sa;
   rand reg [15:0] len;
   rand reg [ 7:0] data;
   rand reg [31:0] fcs;
}
rvm_channel(eth_mac_frame)      // Notice: no ';'
rvm_atomic_gen(eth_mac_frame)

program test {
   eth_mac_frame_atomic_gen src = new("MAC Side");
}
```

Example: External Declaration

```
typedef class eth_mac_frame;
extern rvm_channel(eth_mac_frame)
extern_rvm_atomic_gen(eth_mac_frame)   // Notice: no ';'

program test {
   eth_mac_frame_atomic_gen src = new("MAC Side");
}
```

Atomic Generator Callbacks Base Class - rvm_atomic_gen_callbacks

This class implements a type-generic base class for object-oriented callback containments for all atomic generator transactor, reguardless of the type of the generated data.

Public Interface

```
virtual task post_inst_gen_t(rvm_xactor gen,
   rvm_data obj, var reg drop)
```

Generic object-oriented callback method invoked by the generator after a new object instance has been created but before the type-specific object-oriented callback methods are invoked (see "virtual task post_inst_gen_t(class_name_atomic_gen gen, class_name obj, var reg drop)" on page 364)and after the aspect-oriented callback method (see "task post_inst_gen_t(class_name obj, var reg drop)" on page 361) is invoked.

The gen argument refers to the generator instance that is invoking the callback method. The obj argument refers to the newly generated instance which can be modified. If the value of the drop argument is set to non-zero, the generated instance will not be forwarded to the output channel.

Atomic Generator Callbacks Base Class – rvm_obj_atomic_gen_callbacks

This class implements a type-specific base class for object-oriented callback containments for the atomic generator transactor. This class named `rvm_obj_atomic_gen_callbacks` is automatically declared or implemented for any user-specified call named `rvm_obj` by the atomic generator macros, using a process similar to the `rvm_channel` macro.

Public Interface

```
virtual task post_inst_gen_t(class_name_atomic_gen gen,
   class_name obj, var reg drop)
```

Type-specific object-oriented callback method invoked by the generator after a new object instance has been created but before it is added to the output channel and after the generic object-oriented callback methods are invoked (see “virtual task post_inst_gen_t(class_name_atomic_gen gen,   class_name obj, var reg drop)” on page 364) is invoked.

The `gen` argument refers to the generator instance that is invoking the callback method. The `obj` argument refers to the newly generated instance which can be modified. If the value of the `drop` argument is set to non-zero, the generated instance will not be forwarded to the output channel.

Scenario Generator Transactor - rvm_scenario_gen

The implementation uses a macro to define a class named `rvm_obj_scenario_gen` for any user-specified class named `rvm_obj`, using a process similar to the `rvm_channel` macro. The constructor of the generated class must be callable without arguments.

The scenario generator class is an extension of the `rvm_xactor` class and as such, inherits all of the public interface elements provided in the base class.

For more details on how to use this class, see “Scenario Generators” on page 86 and “Defining or Modifying Scenarios” on page 253.

Note:

> VCS-NTB users are referred to"Vera to Native Testbench Conversion Strategy" in the *Native Testbench Coding Guide* for coding guidelines when using scenario generators.

Macro Interface

The following macros are available to define type-specific scenario generator transactor classes. These macro defines classes named `class_name_scenario_genclass_name_scenario_gen_callbacks`, `class_name_scenario`, and `class_name_scenario_election`.

```
rvm_scenario_gen(class_name, “Class Description”)
```

Define a scenario generator class to generate sequences of instances of the specified class. The generated class must be derived from the `rvm_data` class and the `rvm_obj_channel` class must exist. It must also have a constructor with no arguments or that has default values for all of its arguments.

This macro creates the class interface and implementation. It is typically invoked in the same file where the specified class is defined and implemented.

```
extern_rvm_scenario_gen(class_name)
```

Define a scenario generator class to generate sequences of instances of the specified class as an external class. The `rvm_obj_channel` class must be visible.

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

Public Interface

These public interface elements are in addition to the public interface elements provided by the `rvm_xactor` base class. Please refer to "Public Interface" on page 347 for additional interface elements.

```
task new(string instance,
   integer stream_id = 1,
   rvm_obj_channel out_chan = null)
```

Create a new instance of a scenario generator object with the specified instance name and optional stream identifier. The generator can be optionally connected to a specified output channel instance. If no output channel instance is specified, one will be created internally in the `out_chan` property.

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

`class_name_channel out_chan`

Reference to the output channel for the instances generated by this transactor. The output channel may have been specified via the constructor. If no output channel instances were specified, a new instance is automatically created. This reference in this property may be dynamically replaced but the generator should be stopped (see “virtual task stop_xactor()” on page 324) to during the replacement.

`integer stop_after_n_insts`

The generator will stop (see “virtual task stop_xactor()” on page 324) after the specified number of object instances has been generated and consumed by the output channel. The generator must be reset (see “virtual task reset_xactor(integer rst_type = XACTOR_FIRM_RST)” on page 351) before it can be restarted (see “virtual task start_xactor()” on page 324). If the value of this property is 0, the generated will not stop on its own, based on the number of generated instances (but may still stop based on the number of generated scenarios).

The default value of this property is 0.

`integer stop_after_n_scenarios`

The generator will stop (see “virtual task stop_xactor()” on page 324) after the specified number of scenarios has been generated and entirely consumed by the output channel. The generator must be reset (see “virtual task reset_xactor(integer rst_type = XACTOR_FIRM_RST)” on page 351) before it can be restarted (see “virtual task start_xactor()” on page 324). If the value of this property is 0, the generated will not stop on its own based on the number of generated scenarios (but may still stop based on the number of generated instances).

The default value of this property is 0.

```
class_name_scenario scenario_set[$]
```

Set of scenario descriptor instances that may be repeatedly randomized to create the random content of the output object stream. The individual instances of the output stream are created by calling the `apply_t()` method of the randomized scenario descriptor. The `select_scenario` property is used to determine which scenario descriptor, out of the available set of descriptors, is randomized next.

The scenario generator uses a factory pattern to generate the output stream instances. The generated stream can be constrained using various techniques on this property. See “Defining or Modifying Scenarios” on page 253 for more details. By default, this property contains one instance of the atomic scenario descriptor (see on “Scenario Descriptor Class – rvm_obj_atomic_scenario” on page 377). Out-of-the-box, the scenario generator will generate atomic object instances.

The `scenario_id` property of the generated instance is incremented after each scenario generation. It will be reset to 0 when the generator is reset (see "virtual task reset_xactor(integer rst_type = XACTOR_FIRM_RST)" on page 351) and after the specified number of instances or scenarios has been generated.

`class_name_scenario_election select_scenario`

Reference to a scenario descriptor selector that is repeatedly randomized to determine which scenario descriptor, out of the available set of scenario descriptor, will be randomized next.

By default, a round-robin selection process is used. The constraint blocks or randomized properties in this instance can be turned off or the instance can be replaced with a user-defined extension to modify the election rules. See "Modifying Constraints" on page 249 for the various techniques that can be used to modify the random scenario selection process.

`integer GENERATED`

Event identifier for the `rvm_notify` event notification object in the `notify` property provided by the `rvm_xactor` base class. It is configured as a ONE_SHOT event and is indicated immediately before a scenario is applied to the output channel. The generated scenario is specified as the status of the event.

`integer DONE`

Event identifier for the `rvm_notify` event notification object in the `notify` property provided by the `rvm_xactor` base class. It is configured as an ON_OFF event and is indicated when the generator stops because the specified number of instance or scenarios has been generated. No status information is specified.

```
virtual function reg inject_obj_t(rvm_obj obj)
```

Inject the specified instance in the output stream. Unlike injecting the instance directly in the output channel, it counts toward the number of instances generated by this generator and will be subjected to the callback methods (as an atomic scenario). The method returns once the instance has been consumed by the output channel or it has been dropped by the callback methods. If the instance has been dropped, this function returns FALSE (that is, zero). TRUE (that is, non-zero) is returned if the instance is forwarded to the output channel.

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

```
virtual function reg inject_t(rvm_obj_scenario scenario)
```

Inject the specified scenario in the output stream. Unlike injecting the instance directly in the output channel, it counts toward the number of instances and scenarios generated by this generator and will be subjected to the callback methods. The method returns once the scenario has been consumed by the output channel or it has been dropped by the callback methods. If the scenario has been dropped, this function returns FALSE (that is, zero). TRUE (that is, non-zero) is returned if the instance is forwarded to the output channel.

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

```
task pre_scenario_randomize_t(var class_name_scenario
scenario)
```

Aspect-oriented callback method invoked by the generator after a new scenario has been selected but before it is randomized and before the object-oriented callback methods (see "virtual task pre_scenario_randomize_t(class_name_scenario_gen gen, var class_name_scenario scenario)" on page 382) are invoked.

The `scenario` argument refers to the newly selected scenario descriptor which can be modified. Note that any modifications of the randomization state of the scenario descriptor – such as turning constraint blocks ON or OFF – will remain in effect the next time the scenario descriptor is selected to be randomized. If the reference to the scenario descriptor is set to null, the scenario will not be randomized and a new scenario will be selected.

To minimize memory allocation and collection, the elements of the scenarios may not be allocated. Use the `class_name_scenario::allocate_scenario()` or `class_name_scenario::fill_scenario()` to allocate the elements of the scenario if necessary.

```
task post_scenario_gen_t(class_name_scenario scenario,
   var reg drop)
```

Aspect-oriented callback method invoked by the generator after a new scenario has been created but before it is applied to the output channel and before the object-oriented callback methods (see "virtual task post_scenario_gen_t(rvm_xactor gen, rvm_data scenario, var reg drop)" on page 381) are invoked.

The `scenario` argument refers to the newly randomized scenario descriptor which can be modified. Note that any modifications of the randomization state of the scenario descriptor – such as turning constraint blocks ON or OFF – will remain in effect the next time the scenario descriptor is selected to be randomized. If the value of the drop argument is set to non-zero, the generated scenario will not be applied to the output channel.

Example: Declaration

```
class eth_mac_frame extends rvm_data {
   rand reg [47:0] da;
   rand reg [47:0] sa;
   rand reg [15:0] len;
   rand reg [ 7:0] data;
   rand reg [31:0] fcs;
}
rvm_channel(eth_mac_frame)     // Notice: no ';'
rvm_scenario_gen(eth_mac_frame)

program test {
   eth_mac_frame_scenario_gen src = new("MAC Side");
}
```

Example: External Declaration

```
typedef class eth_mac_frame;
extern rvm_channel(eth_mac_frame)
extern_rvm_scenario_gen(eth_mac_frame)   // Notice: no ';'

program test {
   eth_mac_frame_scenario_gen src = new("MAC Side");
}
```

Scenario Descriptor Class – rvm_obj_scenario

This class implements a base class for describing scenarios or sequences of objects. This class named `rvm_obj_scenario` is automatically declared or implemented for any user-specified class named `rvm_obj` by the scenario generator macros, using a process similar to the `rvm_channel` macro.

See “Defining or Modifying Scenarios” on page 253 for more details on how this class can be used to define new scenarios.

Note:

> VCS-NTB users are referred to "Vera to Native Testbench Conversion Strategy" in the *Native Testbench Coding Guide* for coding guidelines when using scenario generators.

Public Interface

```
static rvm_log log
```

Message service object instance to be used to issue generic messages when the message service object instance of the scenario generator is not available or in scope.

```
integer stream_id
```

Stream identifier. It is set by the scenario generator before the scenario descriptor is randomized. Can be used to express stream-specific constraints.

```
integer scenario_id
```

Scenario identifier within the stream. It is set by the scenario generator before the scenario descriptor is randomized and incremented after each randomization. Can be used to express scenario-specific constraints. The scenario identifier is reset to 0 when the scenario generator is reset (see “virtual task reset_xactor(integer rst_type = XACTOR_FIRM_RST)” on page 351) or when the specified number of scenarios has been generated.

```
function integer define_scenario(string name,
   integer max_len)
```

Defines a new scenario with the specified name and the specified maximum number of instances. Returns a unique scenario kind that should be assigned to an `integer` property.

```
task redefine_scenario(integer kind,
   string name,
   integer max_len)
```

Redefines the name and maximum number of instances in a previously defined scenario. Usually used to redefine the default, ATOMIC, scenario.

```
function string scenario_name(integer kind)
```

Returns the name associated with the specified scenario kind.

```
rand integer kind
```

When randomized, defines the kind of scenario that is generated. Constrained to the defined scenario identifiers (see “function integer define_scenario(string name,   integer max_len)” on page 374).

```
rand integer length
```

Randomized number of items in the scenario. Defines how many instances in the `items` array are part of the scenario.

```
rand rvm_obj items[*] dynamic_size length
```

Instances of user-specified `rvm_obj` classes that are randomized to form the values of the scenarios. Only elements from index 0 to `length`-1 are part of the scenario.

The constraint blocks and rand attributes of the objects in the randomized array may be turned ON or OFF to modify the constraints on scenarios. They can also be replaced with extensions.

The output stream is formed by copying the values of the items in this array onto the output channel.

```
rand integer repeated
```

Randomized number of times the items in the scenario are applied. The repeated instances in the scenario count toward the number of instances generated but only one scenario is considered generated regardless of the number of times it is repeated.

This property is unconstrained by default. To avoid accidentally repeating a scenario a large number of time, a warning message will be issued if the value of this property is greater than the value specified in the repeat_thresh property.

```
static integer repeat_thresh
```

To avoid accidentally repeating a scenario a large number of time, a warning message will be issued if the value of the repeated property is greater than the value specified in this property. The default value is 100.

```
rvm_obj using
```

Reference to a user-specified `rvm_obj` class (or a derivative) used by the `pre_randomize()` method when invoking the `fill_scenario()` method. The default value is null.

```
task allocate_scenario(rvm_obj using = null)
```

Allocate a new set of instances in the `items` property, up to the maximum number of items in the maximum-length scenario. Any instance previously located in the `items` array is replaced. If a reference to a user-specified `rvm_obj` class (or a derivative) is specified, then the array is filled by copies of the specified object (note: it is important that the object's `rvm_data::copy()` method be appropriately overloaded). Otherwise, the array is filled with new instance of `rvm_obj` class.

```
task fill_scenario(rvm_obj using = null)
```

Allocate new instances in the `items` property, up to the maximum number of items in the maximum-length scenario in any null element of the array. Any instance previously located in the `items` array is left untouched. If a reference to a user-specified `rvm_obj` class (or a derivative) is specified, then the array is filled by copies of the specified object. (Note: it is important that the object's `rvm_data::copy()` method be appropriately overloaded). Otherwise, the array is filled with new instance of `rvm_obj` class.

```
virtual function integer apply_t(rvm_obj_channel channel)
```

Apply the current value of the scenario descriptor to the specified output channel and returns the number of generated instances when they have all been consumed by the channel. By default, copies the values of the `items` array.

This method may be overloaded to define procedural scenarios. See “Defining or Modifying Scenarios” on page 253 for more details.

Scenario Descriptor Class – rvm_obj_atomic_scenario

This class implements a predefined atomic scenario descriptor. An atomic scenario is composed of a single unconstrained item. This class named `rvm_obj_atomic_scenario` is automatically declared or implemented for any user-specified class named `rvm_obj` by the scenario generator macros, using a process similar to the `rvm_channel` macro.

See “Defining or Modifying Scenarios” on page 253 for more details on how this class can be used to define new scenarios.

Note:

> VCS-NTB users are referred to "Vera to Native Testbench Conversion Strategy" in the *Native Testbench Coding Guide* for coding guidelines when using scenario generators.

Public Interface

```
integer ATOMIC
```

Symbolic kind for the atomic scenario described by this descriptor. The atomic scenario is a single random unconstrained object (that is, an atomic object).

See “Defining or Modifying Scenarios” on page 253 for details on how to define new scenarios.

```
constraint atomic_scenario
```

Specifies the constraints of the atomic scenario. By default, the atomic scenario is a single unrepeated unconstrained instance. This constraint block may be overridden to redefined the atomic scenario.

Scenario Selector Class – rvm_obj_scenario_election

This class implements a base class for selecting the next scenario descriptor, from a set of available descriptor, to be randomized next. This class named `rvm_obj_scenario_election` is automatically declared or implemented for any user-specified class named `rvm_obj` by the scenario generator macros, using a process similar to the `rvm_channel` macro.

See “Defining or Modifying Scenarios” on page 253 for more details on how this class can be used to modify the selection or distribution of scenarios.

Note:

> VCS-NTB users are referred to "Vera to Native Testbench Conversion Strategy" in the *Native Testbench Coding Guide* for coding guidelines when using scenario generators.

Public Interface

```
integer stream_id
```

Stream identifier. It is set by the scenario generator before the scenario selector is randomized. Can be used to express stream-specific constraints.

```
integer scenario_id
```

Scenario identifier within the stream. It is set by the scenario generator before the scenario selector is randomized and incremented after each randomization. Can be used to express scenario-specific constraints. The scenario identifier is reset to 0 when the scenario generator is reset (see “virtual task reset_xactor(integer rst_type = XACTOR_FIRM_RST)” on page 351) or when the specified number of scenarios has been generated.

```
integer n_scenarios
```

Number of available scenario descriptors. The selected value must be between 0 and the value of this property minus 1 (inclusively).

```
integer last_selected[$]
```

A history (maximum of 10) of the last scenario selections. Can be used to express constraints based on the historical distribution of the selected scenarios (for example, “Never select the same scenario twice in a row.”).

```
integer next_in_set
```

The next scenario descriptor index that would be selected in a round-robin selection process. Used by the `round_robin` constraint block.

```
rvm_obj_scenario scenario_set[$]
```

The available set of scenario descriptor. Can be used to procedurally determine which scenario to select or to express constraints based on the scenario descriptor.

```
rand integer select
```

The index, within the `scenario_set` array, of the selected scenario descriptor to be randomized next.

```
constraint round_robin
```

Constrains the random scenario selection process to a round-robin selection. This constraint block may be turned off to produce a random scenario selection process or allow a different constraint block to define a different scenario selection process.

Scenario Generator Callbacks Base Class - rvm_scenario_gen_callbacks

This class implements a generic base class for object-oriented callback containments for any scenario generator transactor.

Note:

VCS-NTB users are referred to "Vera to Native Testbench Conversion Strategy" in the *Native Testbench Coding Guide* for coding guidelines when using scenario generators.

Public Interface

```
virtual task post_scenario_gen_t(rvm_xactor gen,
   rvm_data scenario,
   var reg drop)
```

Generic object-oriented callback method invoked by the generator after a new scenario has been randomized but before the type-specific object-oriented callback methods (see "virtual task post_scenario_gen_t(class_name_scenario_gen gen,   class_name_scenario scenario,   var reg drop)" on page 382) and after the aspect-oriented callback method (see "task post_scenario_gen_t(class_name_scenario scenario,   var reg drop)" on page 371) is invoked.

The gen argument refers to the generator instance that is invoking the callback method. The scenario argument refers to the newly randomized scenario which can be modified. Note that any modifications of the randomization state of the scenario descriptor – such as turning constraint blocks ON or OFF – will remain in effect the next time the scenario descriptor is selected to be randomized. If the value of the drop argument is set to non-zero, the generated instance will not be applied to the output channel.

Scenario Generator Callbacks Base Class – rvm_obj_scenario_gen_callbacks

This class implements a base class for object-oriented callback containments for the scenario generator transactor. This class named rvm_obj_scenario_gen_callbacks is automatically declared or implemented for any user-specified class named rvm_obj by the scenario generator macros, using a process similar to the rvm_channel macro.

Note:

VCS-NTB users are referred to "Vera to Native Testbench Conversion Strategy" in the *Native Testbench Coding Guide* for coding guidelines when using scenario generators.

Public Interface

```
virtual task
pre_scenario_randomize_t(class_name_scenario_gen gen,
   var class_name_scenario scenario)
```

Object-oriented callback method invoked by the generator after a new scenario has been selected but before it is randomized and after the aspect-oriented callback method (see “task pre_scenario_randomize_t(var class_name_scenario scenario)” on page 370) has been invoked.

The `scenario` argument refers to the newly selected scenario descriptor which can be modified. Note that any modifications of the randomization state of the scenario descriptor – such as turning constraint blocks ON or OFF – will remain in effect the next time the scenario descriptor is selected to be randomized. If the reference to the scenario descriptor is set to null, the scenario will not be randomized and a new scenario will be selected.

To minimize memory allocation and collection, the elements of the scenarios may not be allocated. Use the `class_name_scenario::allocate_scenario()` or `class_name_scenario::fill_scenario()` to allocate the elements of the scenario if necessary.

```
virtual task post_scenario_gen_t(
   class_name_scenario_gen gen,
   class_name_scenario scenario,
   var reg drop)
```

Type-specific object-oriented callback method invoked by the generator after a new scenario has been randomized but before it is applied to the output channel and after the generic object-oriented callback method (see “virtual task post_scenario_gen_t(class_name_scenario_gen gen, class_name_scenario scenario, var reg drop)” on page 382) is invoked.

The gen argument refers to the generator instance that is invoking the callback method. The scenario argument refers to the newly randomized scenario which can be modified. Note that any modifications of the randomization state of the scenario descriptor – such as turning constraint blocks ON or OFF – will remain in effect the next time the scenario descriptor is selected to be randomized. If the value of the drop argument is set to non-zero, the generated instance will not be applied to the output channel.

Watchdog Base Class - rvm_watchdog

This class implements a base class for a testbench or environment watchdog to terminate the simulation in case of lack of activity. It is based on the rvm_xactor class.

Multiple instances of this class can exist to concurrently watch for simulation termination conditions.

The watchdog timer indicates the expiration of its fuse by indicating an event (see rvm_watchdog::TIMEOUT and rvm_xactor::notify) and by issuing a warning message (ID = 0) via its inherited rvm_xactor::log member. If automatic termination of the simulation is required, a separate thread could monitor the indicated event or the handling of the message can be modified using:

```
rvm_watchdog wd = new(...);
wd.log.modify(*, *, *, 0, *, *, *, *, *, wd.log.ABORT);
```

Virtual Ports

The following virtual port is used to monitor a hardware signal for activity. If more than one hardware signal is required, they can be OR'ed in the VeraShell or different instances of this class may be used.

```
port rvm_watchdog_port {pin}
```

HDL signal to monitor for activity. Whenever the value of the signal changes, the timer is reset.

Public Interface

In addition to the public interface elements provided by the `rvm_xactor` class, the following additional interface elements are available:

```
task new(string instance,
   rvm_watchdog_port pin = null,
   integer fuse = 1000)
```

Create an instance of a watchdog timer class with the specified instance name, optionally watching for activity on the specified HDL signal, with a fuse length of the specified number of time units. The name is hard-coded as “Watchdog Timer”.

Note:

> The watch_signal() method should be used to specify signals to be watched. The pin argument exists for backward compatibility reasons.

```
static integer TIMEOUT
```

Pre-defined notification event used to indicate that a timeout has occurred. The event is a ONE_SHOT event.

A warning message (message ID 0) is also issued.

After expiring, the fuse is reset to its default length and the watchdog timer continues execution. In the total absence of activity, the watchdog timer will indicate a timeout condition at every “fuse length” interval.

These symbolic values are also available as macro prefixed with `rvm_watchdog__` (for example, `rvm_watchdog::TIMEOUT` is also available as `rvm_watchdog__TIMEOUT`).

```
virtual task reset_fuse()
```

Reset the fuse on the watchdog timer to its full length. The fuse length can be specified via the constructor or the `rvm_watchdog::fuse()` method. This is equivalent to observing activity on a specified HDL signal.

```
virtual function integer fuse(integer reset_fuse = -1)
```

Return the number of time units remaining on the fuse, optionally resetting the fuse length to a new value. If the new fuse is a negative number, the fuse is not reset. Setting a fuse to 0 causes the watchdog to expire immediately but does not reset the fuse length.

```
virtual task watch_signal(rvm_watchdog_port sig, string
name)
```

Watch for activity on the specified signal and reset the fuse if activity is detected. The specified name is associated with the signal when displaying timer status.

```
virtual task unwatch_signal(rvm_watchdog_port sig)
```

Stop watching for activity on the specified signal.

```
virtual task watch_xactor(rvm_xactor xact)
```

Watch for activity on the specified transactor and reset the fuse if the transactor is not idle.

```
virtual task unwatch_xactor(rvm_xactor xact)
```

Stop watching for activity on the specified transactor.

```
virtual task watch_channel(rvm_channel chan)
```

Watch for activity on the specified channel and reset the fuse if objects are added or removed from the channel.

```
virtual task unwatch_channel(rvm_channel chan)
```

Stop watching for activity on the specified channel.

```
virtual task start_xactor()
```

Start this instance of the watchdog timer. The timer can be stopped. Any extension of this method must call `super.start_xactor()`.

```
virtual task stop_xactor()
```

Same as `reset_xactor()`. The timer can be restarted. Any extension of this method must call `super.stop_xactor()`.

```
virtual task reset_xactor(integer rst_type = 0)
```

Stop and reset this instance of the watchdog timer. The timer can be restarted. Regardless of the reset type specified, the fuse is reset to its default length.

Environment Manager Base Class - rvm_env

Public Interface

```
rvm_log log
```

Top-most message reporting object instance for the verification environment.

```
rvm_notify notify

static integer CFG_GENED
static integer BUILT
static integer DUT_RESET
static integer DUT_CFGED
static integer STARTED
static integer RESTARTED
static integer ENDED
static integer STOPPED
static integer CLEANED
static integer DONE
```

Notification object and pre-defined events used to indicate the state transitions in the verification environment. Pre-defined events are used to signal the start of the predefined virtual methods in this class. All events are ON/OFF events.

These symbolic values are also available as macro prefixed with `rvm_env__` (for example, `rvm_env::STARTED` is also available as `rvm_env__STARTED`).

`task run_t()`

Run the simulation until the `wait_for_end_t()` task returns. It then invokes the `stop()`, `cleanup_t()`, and `report()` methods. This method must be manually invoked in the test program.

`task pre_test_t()`

Execute the preparatory simulation steps up to and including the `cfg_dut_t()` method. Tests are then free to replace randomized instances before calling the `run_t()` method. This method must be used, instead of calling `cfg_dut_t()`, by tests that are to be sequenced with soft restarts in the same simulation.

`virtual task gen_cfg()`

Randomize the test configuration descriptor. If this method has not been manually invoked in the test program, it will be invoked by the `build()` method.

Any user-extension of this method must invoke `super.gen_cfg()` first.

`virtual task build()`

Build the verification environment according to the value of the test configuration descriptor. If this method has not been manually invoked in the test program, it will be invoked by the `reset_dut()` method.

Any user-extension of this method must invoke `super.gen_build()` first.

`virtual task reset_dut_t()`

Reset the DUT. If this method has not been manually invoked in the test program, it will be invoked by the `cfg_dut_t()` method.

Any user-extension of this method must invoke `super.reset_dut_t()` first.

`virtual task reset_dut_t()`

Reset the DUT to its initial, default state. If this method has not been manually invoked in the test program, it will be invoked by the `cfg_dut_t_t()` method.

Any user-extension of this method must invoke `super.reset_dut_t()` first.

`virtual task cfg_dut_t()`

Configure the DUT according to the value of the test configuration descriptor. If this method has not been manually invoked in the test program, it will be invoked by the `start_t()` method.

Any user-extension of this method must invoke `super.cfg_dut_t()` first.

`virtual task start_t()`

Start all the components of the verification environment. If this method has not been manually invoked in the test program, it will be invoked by the `run_t()` method.

Any user-extension of this method must invoke `super.start_t()` first.

```
virtual task wait_for_end_t()
```

When this task returns, indicates that the end of simulation condition has been detected. Users must extend this method as the default implementation is empty.

Any user-extension of this method must invoke `super.wait_for_end_t()` first.

```
virtual task stop_t()
```

Stop all the components of the verification environment.

```
virtual task cleanup_t()
```

Perform clean-up operations to let the simulation terminate gracefully. Clean-up operations may include letting the DUT drain of all buffered data, reading statistics registers in the DUT and sweep the scoreboard for leftover expected responses.

```
virtual task report()
```

Report final success or failure of the test and close all files.

Any user-extension of this method must invoke `super.report()` first.

```
virtual task restart(reg reconfig = 0)
```

Restart the environment to run another test in the same simulation. If the reconfig argument is TRUE (that is, non-zero), it is considered a hard restart and the environment is restarted all the way with the `gen_cfg()` and a new configuration may be used. By default, the value is FALSE (that is, zero), a soft restart is performed and the environment is restarted with the `start_t()` method.

Any user-extension of this method must invoke `super.restart()` first. If reconfig is FALSE, the `rvm_xactor::reset_xactor(XACTOR_RST_FIRM)` method must be called for all of the transactors instantiated in the environment.

Tests sequenced using soft restarts have special requirements put on them since the test and DUT configuration is not modified. They cannot call the `rvm_env::gen_cfg()`, `rvm_env::build()`, `rvm_env::reset_dut_t()` or `rvm_env::cfg_dut_t()` methods directly. They can only affect the environment immediately before the `rvm_env::start_t()` method it called. They can only call one of the subsequent simulation step method, the `rvm_env::pre_test_t()` method or the `rvm_env::run_t()` method.

```
virtual task save_rng_state()
```

This method only need to be implemented in a user extension of the `rvm_env` class if random stability is required between consecutive tests separated by a soft restart (see the `rvm_env::restart()` method). This method should save, in local properties, the state of the object random generator in all instantiated objects in the environment and call the `rvm_xactor::save_rng_state()` method of all instantiated transactors.

```
virtual task restore_rng_state()
```

This method only need to be implemented in a user extension of the `rvm_env` class if random stability is required between consecutive tests separated by a soft restart (see the `rvm_env::restart()` method). This method should restore, from local properties, the state of the object random generator in all instantiated objects in the environment and call the `rvm_xactor::restore_rng_state()` method of all instantiated transactors.

B

OVA Checker Library Quick Reference

The following list contains a brief description of each checker in the library. For more detail refer to the OpenVera Assertions Checker Library Reference Manual.

Checkers in the OVA Checker Library, except `ova_driven`, `ova_forbid_bool`, and `ova_no_contention` (marked with (*) in what follows) use logic equality (==, !=) in their underlying assertions. While the checkers may be used in four-state simulation, they do not detect equality or inequality on x and z (a check of x == y will be false if any operand has an x or z, even if it is in the same bit position).

The following checkers are available in two versions. The default one uses logic equality while another version, also available in the library under a different name, supports case equality (===). The latter checkers in both unit and template forms are located in files having the suffix `.4state` in the OVA Checker Library. Note that Magellan, like most formal tools, supports only synthesizable assertions, and so for use with Magellan, only the default checkers should be used (case equality (===) as in all `.4state` checkers is not synthesizable, whereas logic equality is).

```
ova_arith_overflow
ova_hold_value
ova_const
ova_inc
ova_data_used
ova_quiescent_state
ova_dec
ova_reg_loaded
ova_delta
ova_tri-state
ova_hold
ova_timeout
```

To use the four-state checkers, you define a macro of the same name as the checker in the OVA file, and then `` `include `` the checker file. For example, to use four-state version of the std. unit `ova_inc`, add the following two lines at the beginning of the OVA file:

```
`define inc
`include "$VCS_HOME/etc/ova/inc_u.ova.4state"
```

If you need only the inc template, the above two lines become:

```
`define inc
`include "$VCS_HOME/etc/ova/inc.ova.4state"
```

Value Integrity Checkers

ova_arith_overflow

Checks that the value of a signal does not overflow the range of a specified target signal.

ova_asserted

Once the specified start expression evaluates as true, this checker makes sure that the expression under test is asserted (1 or true) until the stop expression evaluates true (excluding the clock tick when stop is true).

ova_bits

Checks that the value of the signal being tested falls between the specified minimum and maximum number of bits (inclusive) that are asserted or deasserted as indicated by a flag.

ova_check_bool

Verifies that the specified expression is always true.

ova_code_distance

Checks that when the tested expression changes, the number of bits that are different compared to another expression fall within the specified minimum and maximum number of bits.

ova_const

Checks that the value of the signal being tested is always constant. Note: You can use a four-state version of this checker.

ova_deasserted

Once the start expression evaluates true, this checker makes sure that the signal being tested is deasserted (0 or false) until the stop expression evaluates true (excluding the clock tick when stop is true).

ova_dec

Checks that when the signal being tested changes value, the new value is always between the specified minimum and maximum less than the previous value.

ova_delta

Checks that when the signal being tested changes value, the new value is ± the specified minimum to maximum change of the previous value.

ova_even_parity

Checks that the value of the signal being tested always has an even number of bits set to 1.

ova_forbid_bool (*)

Checks that the expression is never true. By default the checker uses case equality (===), but in Magellan, it uses logic equality ==.

ova_inc

Checks that when the signal being tested changes value, the new value is always between the specified minimum and maximum more than the previous value.

ova_mutex

Checks that the two specified signals never evaluate true at the same time.

ova_no_contention(*)

Checks that bus signal being tested always has a single active driver and that there is no X or Z on the bus when driven. In Magellan the test for X or Z is ignored.

ova_odd_parity

Checks that the value of the signal being tested always has an odd number of bits set to 1.

ova_overflow

Checks that the signal being tested does not transition from >= max to <= min.

ova_range

Checks that the signal being tested is greater than or equal to the specified minimum value, and less than or equal to the specified maximum value.

ova_underflow

Checks that the signal being tested does not transition between the specified minimum and maximum values.

ova_value

Checks that the signal being tested is only one of the specified values.

State Integrity Checkers

ova_code_distance

Checks that when the tested expression changes, the number of bits that are different compared to another expression fall within the specified minimum and maximum number of bits.

ova_driven

ova_driven(*)

Checks that all bits are driven (none are floating Z or X).

ova_next_state

Checks that when the signal being tested) is in the specified current state it will transition to one of the specified legal next states.

ova_one_cold

Checks that only one bit is set to zero or, optionally, that all bits are set to 1 in the state value.

ova_one_hot

Checks that only one bit is set to one or, optionally, that all bits are set to 0 in the state value.

ova_quiescent_state

Checks that when a trigger expression evaluates true, the tested expression has the specified value.

ova_tri_state

Checks that the tri-states of the specified input and output signals are equal (==) at the start of the assertion.

Temporal Sequence Checkers

ova_hold

Checks that the value of the signal remains constant for the minimum to maximum number of cycles. A new check begins every time the signal changes

ova_hold_value

Checks that the signal holds the specified value from the specified minimum to maximum number of cycles.

ova_reg_loaded

Checks that the register being tested is loaded with source data. The value of the register is checked against a stored source value starting with a specified number of delay cycles after the trigger condition evaluates true and within the specified end cycle after the trigger evaluates true or when the stop signal evaluates true (whichever occurs first).

ova_req_ack_unique

Verifies that each req receives an ack within the specified interval of clock ticks. Note that ack's are attributed to req's in a fifo manner. (Only available as an OVA unit, no template.)

ova_sequence

Ensure that the expression takes on values in the order specified.

ova_timeout

Checks the signal changes within the specified number of cycles (period).

ova_window

Checks that individual bits in the bit vector signal are asserted or deasserted either within or outside the window delimited by a start event and a stop event or by a specified time interval in clock ticks.

Protocol Checkers

ova_arbiter

Ensures that a resource arbiter generates grants to corresponding requests within the specified minimum and maximum number of clock cycles between a request and a grant. It also verifies optional priority and round-robin or fifo arbitration rules.

ova_data_used

Checks that data from the source signal appears in the destination signal within the specified window. The window is specified as the number of cycles from the time the trigger signal is asserted.

ova_dual_clk_fifo

Implements a checker for a dual-clock, single in- and single out-port queue.

ova_fifo

Implements a checker for a single-clock, single in- and single out-port queue.

ova_flows

Checks that the follower expression evaluates true within the specified minimum and maximum latency period once the leader expression evaluates true.

ova_memory

Checks the integrity of synchronous memory contents and accesses.

ova_memory_async

Checks the integrity of asynchronous memory contents and accesses.

ova_multiport_fifo

Implements a checker for a single-clock, multi-port in- and multi-port out queue. (Cannot be used with Magellan.)

ova_no_contention

Checks that bus always has a single active driver and that there is no X or Z on the bus when driven. In Magellan the test for X or Z is ignored.

ova_req_ack_unique

Verifies that each req receives an ack within the specified interval of clock ticks. Note that ack's are attributed to req's in a fifo manner. (Only available as an OVA unit, no template.)

ova_req_requires

Checks that if the first expression in a sequence evaluates true, then the second and third expressions in the sequence evaluate true before the last expression evaluates true

ova_req_resp

Checks that the rising edge of each bit in the request signal vector is followed by a single rising edge of the corresponding bit in the response signal vector within the latency specified by the minimum and maximum number of clock cycles. It is assumed that no new request is issued until after a response is received for the current request.

ova_stack

Checks operations on a stack.

ova_valid_id

Checks that ID's are issued and returned.

Open Verification Library Compatible Checkers

This section describes OVA checkers that verify the same behavior as checkers available in Accellera's proposed "Open Verification Library", Version 02.09.24.

There are several methods you can use to convert OVL Verilog checker instances in a Verilog model to equivalent inlined OVA checkers, as described in the OVA Checker Library Reference Manual:

- Single Line Replacement
- Multiple Line Replacement
- Instantiation of checkers using Verilog wrapper modules

Note the following restrictions when using OVL-equivalent checkers:

- The OVL checker `assert_proposition` is not available in OVA because it is an asynchronous checker that does not require variable sampling.
- Unit parameters that control the extent of synchronous delays (number of clock ticks) and assertion variants in a checker must be compile-time constants — they must not be specified using design parameters. This restricts the use of Verilog wrapper modules.

Wrapper modules cannot be used with the following checkers:

```
assert_always_on_edge
assert_change
assert_cycle_sequence
assert_frame
assert_handshake
```

```
assert_next
assert_one_cold
assert_time
assert_unchange
assert_width
```

assert_always

This checker continuously monitors *test_expr* at every positive edge of clock, *clk*. It verifies that *test_expr* will always evaluate TRUE. If *test_expr* evaluates to FALSE, the assertion will fire.

assert_always_on_edge

This checker continuously monitors the *test_expr* at every specified edge of the *sampling_event* that coincides with the positive edge of clock, *clk*. The *test_expr* should always evaluate TRUE at the *sampling_event*. If *test_expr* evaluates to FALSE, the assertion will fire.

assert_change

This checker continuously monitors the *start_event* at every positive edge of the clock. When start_event is TRUE, the checker ensures that the expression, *test_expr*, changes values on a clock edge at some point within the next *num_cks* number of clocks. This assertion will fire upon a violation.

assert_cycle_sequence

This checker verifies the following conditions:

- When *necessary_condition* = 0, if all *num_cks*-1 first events of a sequence are TRUE, the last event should follow.
- When *necessary_condition* = 1, if the first event of a sequence is TRUE, then all the remaining events should follow.

assert_decrement

This checker continuously monitors the *test_expr* at every positive edge of the clock signal, *clk*. It checks that the *test_expr* will never decrease by anything other than the value specified by *value*.

assert_delta

This checker continuously monitors the *test_expr* at every positive edge of clock signal, *clk*. It verifies that *test_expr* will never change value by anything less than *min* and anything more than *max* value.

assert_even_parity

This checker continuously monitors the *test_expr* at every positive edge of the clock signal, *clk*. It verifies that *test_expr* will always have an even number of bits asserted.

assert_fifo_index

This checker ensures that the FIFO element:

- Never overflows and underflows
- Allows/disallows simultaneous push and pop.

assert_frame

This checker validates proper cycle timing relationships between two events in the design. When a *start_event* evaluates TRUE, then the *test_expr* must evaluate TRUE within a minimum and maximum number of clock cycles.

assert_handshake

This checker continuously monitors the *req* and *ack* signals at every positive edge of the clock, *clk*. Note that both *req* and *ack* must go inactive prior to starting a new cycle.

assert_implication

This checker continuously monitors *antecedent_expr*. If it evaluates to TRUE, then this checker will verify that *consequent_expr* is TRUE. When *antecedent_expr* is evaluated to FALSE, then *consequent_expr* expression will not be checked at all and the implication is satisfied.

assert_increment

This checker continuously monitors *test_expr* at every positive edge of the clock, *clk*. It verifies that *test_expr* will never increase by anything other than the value specified by value. The *test_expr* can be any valid Verilog expression. The check will not start until the first clock after the *reset_n* is asserted.

assert_never

This checker continuously monitors *test_expr* at every positive edge of clock, *clk*. It verifies that *test_expr* will never evaluate TRUE. The *test_expr* can be any valid Verilog expression. When *test_expr* evaluates TRUE, this checker will fail.

assert_next

This checker verifies the proper cycle timing relationship between two events in the design at every posedge of the clock, *clk*. When a *start_event* evaluates TRUE, then *test_expr* must evaluate TRUE exactly *num_cks* number of clock cycles later. This checker supports overlapping sequences. For example, if you assert that *test_expr* will evaluate TRUE exactly four cycles after *start_event*, it is not necessary to wait until the sequence finishes before another sequence can begin.

assert_no_overflow

This checker ensures that the *expr*, from *max* value, never goes to a value that is less than or equal to *min* and greater than *max*, at every posedge of the clock, *clk*..

assert_no_transition

This checker ensures that, when the state variable *test_expr* reaches a value specified by *start_state*, it does not transit to a state/value specified by *next_state*. All variables are sampled at posedge of the clock, *clk*.

assert_no_underflow

This checker ensures that *test_expr* never changes from *min* value to a value that is less than *min* and greater than or equal to *max*.

assert_odd_parity

This checker monitors for odd number of '1's in *test_expr* at every positive edge of the clock, *clk*.

assert_one_cold

This checker ensures that the variable, *test_expr*, has only one bit low at any positive clock edge when the checker is configured for no inactive states. The checker can also be configured to accept all bits equal to either 0 or 1 as the inactive level.

assert_one_hot

This checker ensures that the variable, *test_expr*, has only one bit high at any positive clock edge.

assert_quiescent_state

This checker verifies that the value in the variable *state_expr*, is equal to the value specified by *check_value* when a sampled positive edge is detected on *sample_event*.

assert_range

This checker ensures that the value of *test_expr* will always be within the *min* and *max* value range.

assert_time

This checker continuously monitors the *start_event* at every positive edge of the clock, *clk*. When start_event is TRUE, the checker ensures that the expression, *test_expr*, is TRUE up to *num_cks* number of clock ticks.

assert_transition

This checker ensures that, when the state variable *test_expr* reaches the value specified by *start_state*, it does transit to a state/value specified by *next_state*.

assert_unchange

This checker monitors the *start_event* at every positive edge of the clock, *clk*. When *start_event* is TRUE, the checker ensures that the expression, *test_expr* does not change its value within *num_cks* clocks.

assert_width

This checker ensures that, when *test_expr* becomes TRUE it should remain TRUE at least for *min* number of clock cycles and at most *max* number of clock cycles. It should never remain TRUE beyond that limit.

assert_win_change

This checker ensures that *test_expr* changes its value at least once between the assertions of *start_event* and *end_event*.

assert_window

This checker ensures that *test_expr* is asserted 1 as long as the window is open. Window open and close events are signaled by *start_event* and *end_event* expressions. The verification starts on the next clock tick following *start_event*.

assert_win_unchange

This checker ensures that the *test_expr* never changes its value between the assertions of *start_event* and *end_event*.

assert_zero_one_hot

This checker ensures that the variable, *test_expr*, has only one bit 1 or all bits 0 at any positive edge of the clock, *clk*.

Glossary

AOP
Aspect Oriented Programming

Assertion
An assertion is a statement of designer's and specification intent, used in simulation or in formal verification. Assertions signal failure when the expected behavior is not respected. Assertions in the form of cover statements provide information on how well the design has been functionally exercised.

Assertion-based checker
An assertion-based checker is a monitor of some specific behavior implemented using assertions. Usually, it is packaged for reuse. Checkers can range from simple ones that cover typical signal sequences or data structures, usually provided in a checker library, to very complex ones that verify complete interface protocols, such as PCI, AMBA, etc.

Assumption

An assumption describes the behavior of the environment of the "DUV" . Usually, that behavior is described using a collection of assertions. It is used in formal tools to constrain DUV inputs to valid sequences. In hybrid-formal tools it may also constrain DUV inputs during random simulation.

Component

A part of the "Verification Environment" that runs in parallel with other parts. Examples of components include

- Bus-functional models
- Generators
- Protocol checkers

Counterexample

A counterexample is a signal trace from the initial state to the point of detecting an assertion failure, generated by a formal verification tool.

Design Under Test (DUT)

The DUT is the design being functionally verified. A DUT is usually an RTL model written in Verilog or VHDL, but can be a transaction-level model written in OpenVera or C++.

Design for Verification (DFV)

Design For Verification - usually implies instrumenting the RTL design with assertions that document design intent and assumptions on neighboring blocks. Allows detecting bugs near their origin in the design; simplifies debugging.

DUV

Design Under Verification

Functional Coverage Model

A functional coverage model is a formal definition of the verification plan. It identifies all of the interesting and relevant conditions that must be applied to the input of the design under test, or observed on its output, or internally. During a simulation, every time one of those interesting or relevant conditions is identified, a datum is recorded in a database to indicate that this particular functional coverage point has been covered. A functional coverage model can be implemented implicitly as the set of all written tests, or explicitly coded and measured via a database.

Functional Coverage Point

A functional coverage point is an interesting or relevant condition in the functional coverage model. It can be a particular scalar value, a set of scalar values, a combination of scalar values or a sequence of scalar values. Functional coverage points can be implicitly implemented in a written test, or explicitly defined and recorded in a database.

Hybrid Verification Tool

A tool comprising formal verification engines such as ROBDD reachability analysis and simulation engines (single valued or symbolic), operating in a coordinated manner.

OpenVera™ Assertions (OVA)

OpenVera Assertions are part of the OpenVera hardware verification language available under an open source license. Consult the The OpenVera Language Reference Manual: Assertions (LRM) for details.

OVL

Open Verification Library

Reference Model

A reference model is a model of the design under test that is, by definition, a golden reference for its functionality. Reference models may be transaction-level models, RTL models or gate-level models. They may be written in OpenVera, Verilog, VHDL, or C++.

Scenario

Scenarios are sequences of transactions with certain random parameters. Each scenario represents an interesting sequence of individual transactions to hit a particular functional corner case.

Scoreboard

A scoreboard is a data structure used to store the expected data produced by the transfer function. It is designed to make comparison of the actual data observed from the design under verification efficient and accurate.

Self-Checking Structure

The self-checking structure is the portion of the verification environment responsible for verifying the correctness of the DUT response. A portion of the self-checking structure can be implemented using assertions written in OVA or provided by OVL. A self-checking structure usually includes a transfer function and a scoreboard.

Simulation

A simulation is a run of a particular test with a particular seed. Each simulation will collect incremental functional coverage data.

Testcase

A testcase is a set of constraints, commands or directed transaction sequences on a verification environment to verify certain functional aspects of the design under test. There can be multiple simulations of a testcase, each performed with a different seed.

Transaction

A transaction is a logical operation performed by a transactor. The type of operation that constitutes a transaction depends on the level of abstraction of the transactor. Examples of transactions include:

- Requesting access to a shared resource
- Performing a read or write cycle on a bus
- Transferring a packet
- Reliably transferring a packet, retrying the packet if a transmission error has been detected.
- Enumerating a USB network

Transaction-Level Model

A transaction-level model is a model of the design under test expressed at the data transformation level. It does not contain any specific implementation or physical interface.

Transactor

A transactor is a component that performs a transaction by translating it into or abstracting it from lower-level transactions or physical-level signals.

Transfer Function

A transfer function is a model of the data transformation performed by the design under test under the current configuration. It may be implemented using a "Reference Model" or "Transaction-Level Model" of the DUT. Examples of data transformation include:

- Moving data from one port to another
- Rewriting some portion of the data
- Executing commands specified in the data

Verification Environment

A verification environment is an encapsulation of all of the components and self-checking structure necessary to stimulate a design under test and verify at all times the correctness of its operations. Tests are run on top of the verification environment.

Verification Plan

A verification plan is a specification of the objectives of the functional verification process. A verification plan can be explicitly and formally specified using a functional coverage model.

Index