Verification Continuum™

VC Verification IP AMBA AXI/ACE Tutorial

Version S-2021.06, June 2021

SYNOPSYS®

Copyright Notice and Proprietary Information

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.

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at http://www.synopsys.com/company/legal/trademarks-brands.html.
All other product or company names may be trademarks of their respective owners.

Free and Open-Source Software Licensing Notices

If applicable, Free and Open-Source Software (FOSS) licensing notices are available in the product installation.

Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

www.synopsys.com

Contents

Chapter 1
Overview 5
1.1 Introduction 5
1.2 Prerequisites 5
1.2.1 Software Installation 5

Chapter 2
Verifying the UVM RVP Example for ACETM 7
2.1 Verifying the UVM RVP Example for ACE 7
2.1.1 Installing and Viewing the Example 7
2.1.2 Running the Test 8
2.1.3 Visualizing and Analyzing the Test Results 9
2.1.4 Injecting Errors to Trigger the System Monitor 17
2.1.5 Improving Functional Coverage by Creating a New Test 18
2.1.6 Running Predefined ACE Sequences 21

1 Overview

1.1 Introduction

This tutorial describes the usage of VC Verification IP for AMBA AXI/ACE tools to verify the UVM Reference Verification Platform (RVP) example for ARM® AMBA™ ACE™. This tutorial guides you through the following steps:

- Create a new example.
- View the simulation results using Protocol Analyzer.
- Load memories and caches.
- Improve functional coverage results.
- Back annotate the results to Verification Plans.
- Run the sequence library examples provided by Synopsys.

1.2 Prerequisites

The following prerequisites are required to run and verify the UVM RVP example for ACE:

- Familiarity with VCS, SystemVerilog, UVM, and AXI4/ACE protocols.
- Run the AMBA AXI4/ACE `amba_svt` examples including `tb_axi_svt_uvm_basic_sys` and `tb_axi_svt_uvm_intermediate_sys`.
- Install the softwares mentioned in the section "Software Installation".

1.2.1 Software Installation

Prior to running an example, make sure to install the following softwares:

- VCS F-2011.12-1 or higher, 32-bit and 64-bit versions, and GNU library.

Do not install the VCS version F-2011.12, as it is not compatible with the AXI/ACE VIP examples.

- DesignWare v1.10a or later versions from Electronic Software Transfer (EST).

Make sure to set the environment variable DESIGNWARE_HOME

To set the DESIGNWARE_HOME variable:

```
> setenv DESIGNWARE_HOME <path to DesignWare>
```

To set the path for DESIGNWARE_HOME:

```
> set path = ($DESIGNWARE_HOME/bin $path)
```

2

Verifying the UVM RVP Example for ACE™

2.1 Verifying the UVM RVP Example for ACE

This chapter explains the procedure to verify the UVM RVP Verification IP example for AXI/ACE. It leads you through installing and running the UVM RVP VIP example for AXI/ACE, viewing the simulation results and functional coverage, using the system monitor to run an error injection test, analyzing the results, and modifying the coverage using Synopsys VIP tools.

Perform the following steps to verify the UVM RVP example for AXI/ACE:

- Installing and Viewing the Example
- Running the Test
- Visualizing and Analyzing the Test Results
- Injecting Errors to Trigger the System Monitor
- Improving Functional Coverage by Creating a New Test
- Running Predefined ACE Sequences

2.1.1 Installing and Viewing the Example

To install the UVM RVP example for ACE from the DesignWare release path into your local directory and view the test files, perform the following steps:

1. Install the UVM RVP for ACE sequence example provided with VIP installation:

```
> mkdir ~/quickstart; cd ~/quickstart
> dw_vip_setup -info home | grep uvm_ace
> dw_vip_setup -svtb -path uvm_ace -e amba_svt/tb_axi_svt_uvm_ace_sys
> cd uvm_ace/examples/sverilog/amba_svt/tb_axi_svt_uvm_ace_sys
```

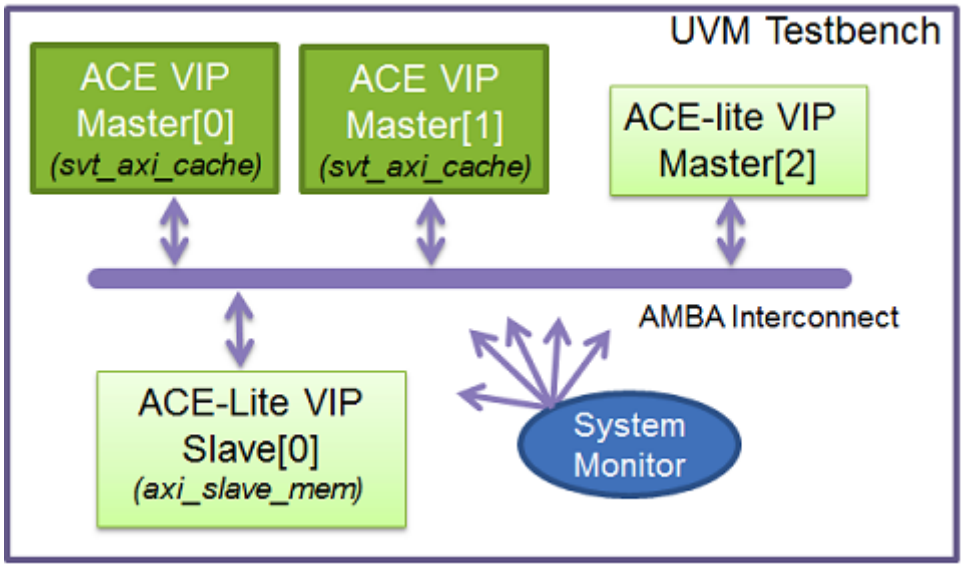
2. Open the README file to view the description of the files and the environment.
3. **View the test:** The test file `tests/ts.triple_test.sv`, sets the master virtual sequence to `axi_master_virtual_sequence` and sets the sequence length to 10.
4. **View the configuration:** Open `env/cust_svt_axi_system_configuration.sv` to view the system configuration.

   In this file, macros set the number of masters to be 3, and the number of slaves to 1. In the constructor, the AMBA `master[0]` and `master[1]` are set to AXI_ACE, while the AMBA `master[2]` is set to ACE_LITE. The slave is also set to ACE_LITE.

The AMBA interconnect is modeled with VIP. Each master connects to a slave port on the interconnect, and so, the master configuration is copied to the interconnect's slave port.

Figure 2-1 displays the reference design, memory, and cache for the ACE VIP.

Figure 2-1 ACE VIP Architecture



5. **View the sequence**: The virtual sequence in `env/axi_master_virtual_sequence.sv` runs sequences on the three masters concurrently. You can view the transactions at the end of the body() method.

   master[0] sends MAKEUNIQUE commands to address 0.

   master[1] sends READSHARED commands to address 0.

   master[2] sends READONCE commands to address 'h1000 (an ACE-Lite command).

2.1.2 Running the Test

To run the installed test, perform the following steps:

1. **Generate waves:** Compile and run `triple_test` that runs the sequences mentioned in the previous section, and present in the `tb_axi_svt_uvm_ace_sys` directory using the command:

   ```
   > run_axi_svt_uvm_ace_sys -w triple_test vcsvlog
   ```

Note If you observe odd linking errors about 64-bit files, add the `-32` switch to use 32-bit mode.

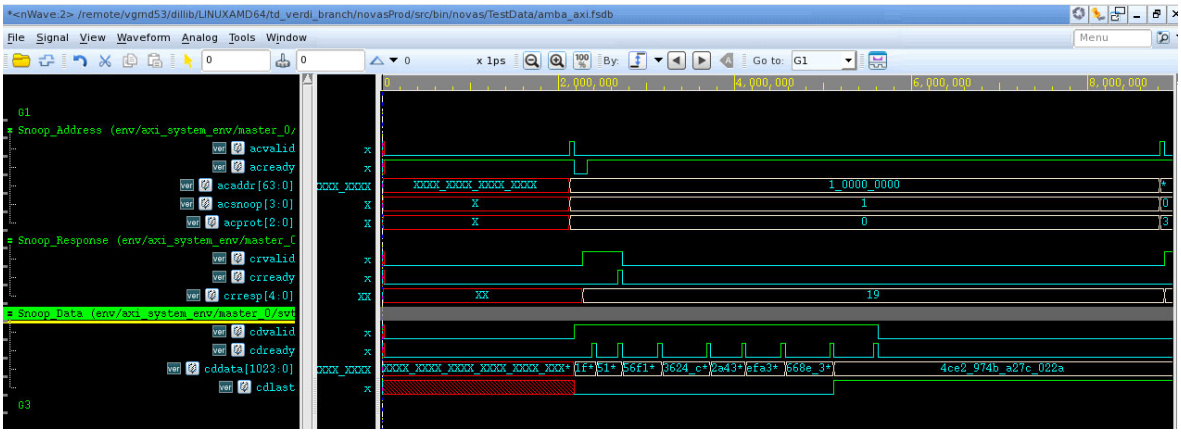
```
> run_axi_svt_uvm_ace_sys -w -32 triple_test vcsvlog
```

2. View the report summary at the end of `logs/simulate.log` file for any problems. There should be many UVM_INFO statements but no UVM_WARNING, UVM_ERROR, or UVM_FATAL statements, as these statements signify problems in the simulation.

2.1.3 Visualizing and Analyzing the Test Results

To visualize and analyze the test results, perform the following steps:

1. **Analyze waves:** In the Verdi window, you can view the bus activity in the form of waves. For example, in the wave window, expand the groups *M0 Snoop Address Channel*, *M0 Snoop Response Channel*, and *M0 Snoop Data Channel*.



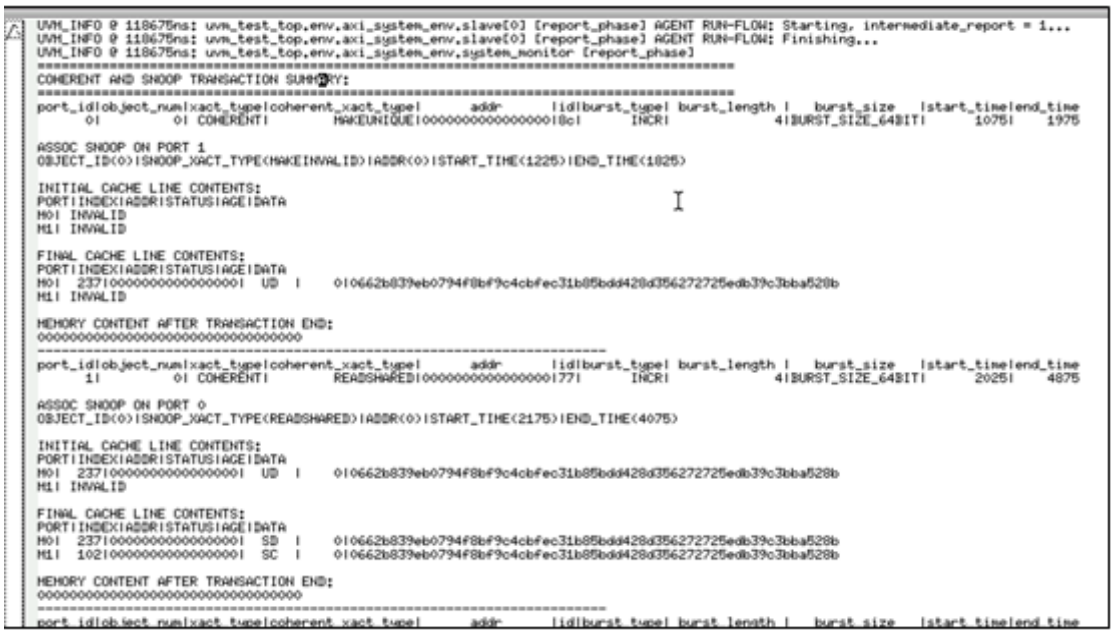
While all the signals are visible, the transactions are not immediately evident. To view and analyze the transactions in an easier and more precise way, you can view them at a higher level. Exit Verdi after this step.

2. **Analyze the test results:** Run the simulation with high verbosity:

```
> output/simvcssvlog +UVM_TESTNAME=triple_test +UVM_VERBOSITY=UVM_HIGH
```

Search the log file for "SUMMARY" to view the report on *Coherent* and *Snoop* transactions. This report shows all the details of the transaction, but it is still challenging to visualize all the transactions.

Note Simulation results will vary depending on the simulator, version, and random seed.



1. **Analyze the graphical result:** The Synopsys Protocol Analyzer allows you to graphically explore the transactions for a wide range of protocols and verification methodologies.

Note For more details, see Synopsys Protocol Analyzer User Guide.

2.1.4 Injecting Errors to Trigger the System Monitor

The AMBA System Monitor checks the integrity of transactions flowing across the interconnect. You are using the Synopsys VIP interconnect, which is loss-less, so normally all the transactions complete successfully.

In this example, you will corrupt a transaction to mimic a bug in your RTL. The UVM uses callbacks to allow you to inject new behavior into an existing flow. The VIP calls the method `pre_output_port_put()` in a callback object, before it outputs the transaction. By default, this method is empty. But if you extend the callback object, `svt_axi_interconnect_callback`, and redefine the method, you can change the VIP behavior.

The test erases the data from a transaction, using the callback.

To view the callback class and run the error injection test that corrupts a transaction, perform the following steps:

1. The callback class is defined in `env/cust_svt_axi_interconnect_error_injection_callback.sv`:

```
class cust_svt_axi_interconnect_error_injection_callback extends
svt_axi_interconnect_callback;
   virtual function void pre_output_port_put(svt_axi_interconnect
                  axi_interconnect, svt_axi_ic_slave_transaction xact);
      `uvm_info("pre_output_port_put", "Corrupting response", UVM_LOW);
      foreach(xact.data[i])
        xact.data[i] = 32'hfeed_face;  // Erase data in the transaction
```

```
   endfunction
endclass
```

2. The test `tests/ts.error_injection_test.sv` registers the callback in the connect phase.

```
class error_injection_test extends base_test;
  …
  virtual function void connect_phase(uvm_phase phase);
    cust_svt_axi_interconnect_error_injection_callback interconnect_cb;
    super.connect_phase(phase);
    interconnect_cb = new("interconnect_cb");
    svt_axi_interconnect_callback_pool::add(
                env.axi_system_env.interconnect.driver, interconnect_cb);
  endfunction
endclass
```

3. Compile and run the test.

```
> run_axi_svt_uvm_ace_sys error_injection_test vcsvlog
```

4. Search for the message "Corrupting response" in the log file. This message shows that the callback was successful.

   View the log file, to read the messages displayed after the end of the transaction. An UVM_ERROR is displayed by the System Checker in the System Monitor.

Note The VIP only checks the data in some transactions, so the VIP may not generate a warning for every corrupted transaction.

5. Set the variable `is_error_injection` present in the base test class in `env/axi_base_test.sv`, to change how the errors are reported at the end of simulation.

   The test in `tests/ts.err_injection.sv` sets the variable to signal that an error is expected. This test deliberately causes errors.

2.1.5 Improving Functional Coverage by Creating a New Test

In a typical verification flow, an engineer writes tests, runs them, and then analyzes the functional coverage results to find the functions not tested, so that new tests can be designed.

In this section, you will run an existing test, and use the AMBA VIP to collect functional coverage to measure the coverage according to the ARM Limited specifications. Thereafter, changes are made to the configuration and stimulus to create a new UVM test.

The test in `tests/ts.directed0_test.sv` runs one MAKEUNIQUE command to address 0. This is a simple test that extends `axi_base_test` defined in `env/axi_base_test.sv`. To create a new test, perform the following steps:

1. Review the class in `env/axi_master_virtual_sequence_directed0.sv`. This extends the existing `axi_master_virtual_sequence` class, with a new `body()` method. In this task, the `master[0]` issues a single MAKEUNIQUE command to address 0. Run the simulation with verbosity turned to high to view the commands.

```
> ./output/simvcssvlog +UVM_TESTNAME=directed0_test +UVM_VERBOSITY=UVM_HIGH
```

2. **Enable functional coverage:** Collecting functional coverage data can slow the simulation, just like dumping waveform data. So, by default, it is off. Make sure, the transaction coverage is turned on for the master and slave in `env/cust_svt_axi_system_configuration.sv` using the following code:

```
…
cfg.master_cfg[i].transaction_coverage_enable = 1;
…
cfg.slave_cfg[i].transaction_coverage_enable = 1;
```

3. Compile and run the test using the specified command. (Make a note of the test name.)

   ```
   > run_axi_svt_uvm_ace_sys directed0_test vcsvlog
   ```

4. Run the test again with verbosity turned to high. Search in the log file `logs/simulate.log` for the section SUMMARY to see if the MAKEUNIQUE command ran, and to check the cache state before and after the command.

   ```
   > ./output/simvcssvlog +UVM_TESTNAME=directed0_test +UVM_VERBOSITY=UVM_HIGH
   ```

5. **Visualize the coverage:** Delete any old coverage data in `output/simvcssvlog.vdb`, and then run the model again. Generate and view the functional coverage report.

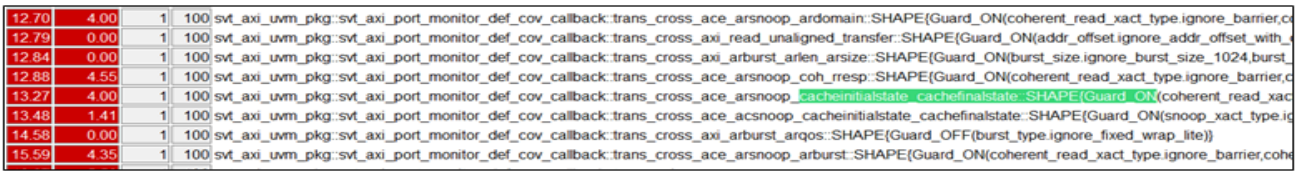
   You can view the results with Firefox on UNIX, or move the results over to a directory visible to a PC and use any browser in Windows to view the report.

   ```
   > rm -rf output/simvcssvlog.vdb
   > ./output/simvcssvlog +UVM_TESTNAME=directed0_test
   > urg -dir output/simvcssvlog -report coverage/directed0_test
   > firefox coverage/directed0_test/dashboard.html &
   ```

6. **Analyze the report:** The Synopsys VIP collects the functional coverage data, which the URG puts in the report. For the *Read Address Channel*, you can view the functional coverage data in the functional coverage report.

   a. On the *Dashboard* page, click **Groups**.

   b. Search for the cross product of *ARSNOOP* and the cache's initial and final states by searching for the group name below and clicking the one with instance coverage of 4.00%.
   `trans_cross_ace_arsnoop_cacheinitialstate_cachefinalstate::SHAPE{Guard_ON`



7. Click the link for `master[0]` and have a look at the display for "Group Instance : uvm_test_top.env.axi_system_env.master[0]_trans_cross_ace_arsnoop_cacheinitialstate_cachefinalstate". The functional coverage score of 4.0 is calculated from the cross arsnoop_cacheinitialstate_cachefinalstate which has a weight of 1. This is a cross of the variables coherent_read_xact_type, initial_cache_line_state, and final_cache_line_state, which have a weight of 0, and so do not add to the coverage score.

Notice that there is only a single covered bin, with the transition `initial_cache_invalid` and `final_cache_unique_clean`. The directed0 test does a single READCLEAN to an unused cache location (INVALID), and this bin shows that.

8. **Modify the test behavior:** You can increase the coverage by writing to the cache using the `svt_axi_cache::write()` method, before the ACE read commands that read from the cache. This has already been done in `tests/ts.directed1_test.sv` and `env/axi_master_virtual_sequence_directed1.sv`.

   You can find documentation on this method by running the Protocol Analyzer (PA).

   a. In PA, click on the *Protocols* tab in the left pane, then expand *AXI SVT VIP* > *Help Resources* and double-click *AXI SVT VIP UVM svdoc*.

   b. In the upper right search box, search for "write", select the exact match, then scroll down and select the write function in *svt_axi_cache*.

   c. The index is -1, to inform the model to allocate an index. The address is 1000. Data and the enable bits are set in the sequence. The last two arguments are the *Unique* and *Clean* bits, so this location is written as *Shared Dirty*. (If the cache line is Clean, you must also manually write the data to memory otherwise the System Monitor will print an error.)

   ```
   axi_system_env.master[1].axi_cache.write(-1, 1000, data, enable, 0, 0);
   ```

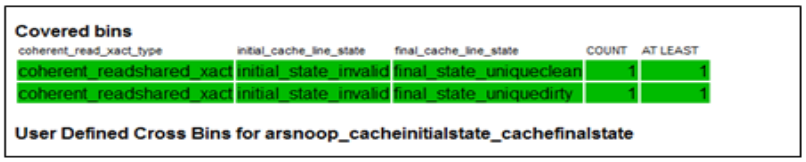
9. Run the new test. You can run either set of commands specified here.

   ```
   > rm -rf output/simvcssvlog.vdb
   > ./output/simvcssvlog +UVM_TESTNAME=directed1_test
   > urg -dir output/simvcssvlog -dir coverage/directed_test
   > firefox coverage/directed1_test/dashboard.html &
   ```

Note

View the coverage results: On the *Dashboard*, go to *Groups* and then and then search again for the following group. arsnoop_cacheinitialstate_cachefinalstate::SHAPE{Guard_ON(… In this group, view the master[4] instance and then Click on the cross name `arsnoop_cacheinitialstate_cachefinalstate` and scroll down to the Covered bins

1. Note that, now you have covered the initial cache state of Unique Dirty



Covered bins

coherent_read_xact_type	initial_cache_line_state	final_cache_line_state	COUNT	AT LEAST
coherent_readshared_xact	initial_state_invalid	final_state_uniqueclean	1	1
coherent_readshared_xact	initial_state_invalid	final_state_uniquedirty	1	1

User Defined Cross Bins for arsnoop_cacheinitialstate_cachefinalstate

2.1.6 Running Predefined ACE Sequences

The previous section demonstrated that it requires a lot of work to increase the coverage results with directed tests.

The AMBA AXI4/ACE VIP includes a set of predefined ACE sequences to exercise all the combinations of legal transactions to verify a custom interconnect. These sequences are configuration aware, which means that these sequences view the configuration of the system (number of masters and slaves, ACE, ACE-Lite, AXI4, and so on) and generate stimulus for all the components in the system.

To run an existing ACE sequence, perform the following steps:

1. View one of the tests such as `tests/ts.single_port_makeunique_test.sv`
2. Run the test `single_port_makeunique_test` with high verbosity.

```
> ./output/simvcssvlog +UVM_TESTNAME=single_port_makeunique_test
+UVM_VERBOSITY=UVM_HIGH
```

3. Search for the section labeled COHERENT AND SNOOP TRANSACTION SUMMARY in the log file *logs/simulate.log*. A sample transaction edited for brevity is shown below:

Note

Your test result will vary depending upon the random seed.

```
port|object|xact_type|coherent_xact|  addr   |id|burst|len| size|start|end
   0|     0| COHERENT|   MAKEUNIQUE|375dbf58|2d| WRAP|  4|64BIT| 1075|1925
ASSOC SNOOP ON PORT 1
OBJECT_ID(0)|SNOOP_XACT_TYPE(MAKEINVALID)|ADDR(375dbf58)|START(1225)|END(1725)
INITIAL CACHE LINE CONTENTS:
PORT|INDEX|ADDR|STATUS|AGE|DATA
M0| INVALID
M1| INVALID
FINAL CACHE LINE CONTENTS:
PORT|INDEX|ADDR|STATUS|AGE|DATA
M0|   27|375dbf58| UD | 0 |0c2d560e3eb4b81e6598ca6d94a998c913662b6e13
M1| INVALID
MEMORY CONTENT AFTER TRANSACTION END:
00000000000000000000000000000000
```

This sequence runs a MAKEUNIQUE command for address 375dbf58 from *Master 0*. Initially, the caches in both masters are invalid. After the command, the *M0 cache* has an entry at index 27 with the data 0c2d… The M1 cache is still empty. The memory is unaffected by the transaction. View the simulation result.

4. Run Protocol Analyzer and compare the transactions in PA with the log file.
5. Generate the functional coverage report with the `single_port_makeunique_test` test.
6. Open the functional coverage report and view the *Dashboard* > *Groups* > *trans_cross_ace_**arsnoop_cacheinitialstate_cachefinalstate::SHAPE{Guard_ON**(coherent_read_xact_type.ignore_barrier,coherent_read_xact_type.ignore_dvm), Guard_OFF(coherent_read_xact_type.ignore_coh_read_xact_ace_lite)}*.

   The coverage has been boosted to 10% for the two masters (depending on the random seed). If you run multiple simulations with different seeds, and merge the coverage data, coverage for this point will be even higher.

   You can run the Synopsys ACE sequences instead of crafting new tests and analyzing coverage reports, to save your time.