OSVVM Model Library:
OSVVM
Test Writers User Guide

User Guide for Release 2021.11

By

Jim Lewis

SynthWorks VHDL Training

Jim@SynthWorks.com

http://www.SynthWorks.com

Table of Contents

1 Overview

Most people don't think of VHDL as a verification language. However, with the Open Source VHDL Verification Methodology (OSVVM) utility and verification component libraries it is. Using OSVVM we can create readable, powerful, and concise VHDL verification environments (testbenches) whose capabilities are similar to other verification languages, such as SystemVerilog and UVM.

This paper covers the basics of using OSVVM's transaction-based test approach to write directed tests, write constrained random tests, use OSVVM's generic scoreboard, add functional coverage, add protocol and parameter checks, add message filtering, and add test wide reporting.

2 Why VHDL? Why OSVVM?

According the 2020 Wilson Research Group Functional Verification Study,

- 64% of FPGA design worldwide uses VHDL
- 50% of FPGA verification worldwide uses VHDL
- 18% of FPGA verification projects worldwide use OSVVM (or 36% of VHDL FPGA verification projects)
- For Europe, 34% of FPGA verification projects use OSVVM while only 26% use UVM

Clearly VHDL is a dominant language in the FPGA market and OSVVM is a leading VHDL Verification Methodology.

3 Benefits of OSVVM

For the VHDL community, OSVVM is a clear win. We can write tests in the same language we already know and re-use components, tests, and testbenches from other projects. More importantly OSVVM's transaction-based approach simplifies creating readable and reviewable tests (an important metric in the safety critical community). In addition, OSVVM uses a component (entity) based approach just like RTL design. Hence, not only can RTL designers read tests and verification components, they can also write them. While having independent design and verification teams is important, it is also important to be able to deploy engineers to either a design or verification role on a project by project basis.

4 What are Transactions?

Writing tests is all about creating waveforms at an interface. In a basic test approach, each test directly drives and wiggles interface waveforms. This is tedious and error prone.

In OSVVM, signal wiggling is replaced by transactions. A transaction is an abstract representation of an interface waveform (such as Send for a UART transmit) or a directive to the verification component (such as wait for clock). In OSVVM, a transaction is initiated using a procedure call. In a verification component (VC) based approach, the procedure call collects the transaction information and passes it to the VC via a transaction interface (a record). The VC then decodes this information and creates the corresponding interface stimulus to the device under test (DUT).

Figure 1 shows two calls to a send procedure and the corresponding waveforms produced by the UartTx verification component.

```
UartTbTxProc : process
begin
  WaitForBarrier(StartTest) ;
  Send(UartTxRec, X"4A") ;
  Send(UartTxRec, X"4B") ;
```

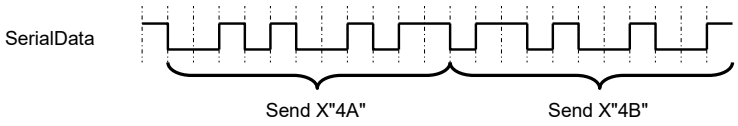

Figure 1. Two Calls to Send transaction and the resulting waveform

5 The OSVVM Testbench Framework

The objective of any verification framework is to make the Device Under Test (DUT) "feel like" it has been plugged into the board. Hence, the framework must be able to produce the same waveforms and sequence of waveforms that the DUT will see on the board.

The OSVVM testbench framework looks identical to other frameworks, including SystemVerilog. It includes verification components (AxiManager, UartRx, and UartTx) and TestCtrl (the test sequencer) as shown in Figure 2. The top level of the testbench connects the components together (using the same methods as in RTL design) and is often called a test harness. Connections between the verification components and TestCtrl use VHDL records as an interface (aka transaction interface). Connections between the verification components and the DUT are the DUT interfaces (such as UART, AxiStream, AXI4, SPI, and I2C).

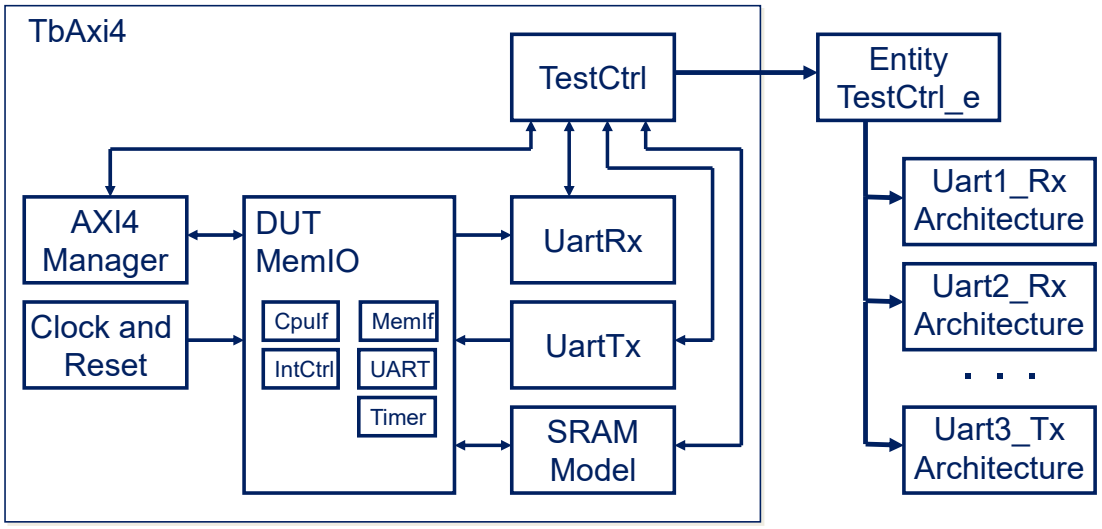

Figure 2. OSVVM Testbench Framework

For more details, including to OSVVM reference testbenches, see Axi4_VC_user_guide.pdf and AxiStream_user_guide.pdf

6 TestCtrl, The OSVVM Test Sequencer

The TestCtrl architecture consists of a control process plus one process per independent interface, see Figure 3. The control process is used for test initialization and finalization. Each test process creates interface waveform sequences by calling the transaction procedures (Write, Send, …).

Each test is a separate architecture of TestCtrl (the test sequencer). Hence, a single test is visible in a single file, improving readability.

Since the processes are independent of each other, synchronization is required to create coordinated events on the different interfaces. This is accomplished by using synchronization primitives, such as WaitForBarrier (see TbUtilPkg_user_guide.pdf and TbUtilPkg_quickref.pdf).

```
architecture UartTx1  of TestCtrl is
  . . .
begin
  ControlProc : process
  begin
    . . .
    WaitForBarrier(TestInit) ;
    . . .
    WaitForBarrier(TestDone, 5 ms) ;
    ReportAlerts ;
    std.env.stop;
  end process ;
  Axi4MProc : process
  begin
    WaitForBarrier(TestInit) ;
    Write(. . .) ;
    WaitForBarrier(DutInit);
    . . .
    WaitForBarrier(TestDone) ;
  end process Axi4MProc ;
  TxProc : process
  begin
    WaitForBarrier(DutInit);
    Send(. . .) ;
    . . .
    WaitForBarrier(TestDone) ;
  end process TxProc ;
  . . .
```

Figure 3. TestCtrl Architecture

7 Test Initialization

The ControlProc both initializes a test and finalizes a test. Test initialization is shown in Figure 4. SetAlertLogName sets the test name. Each verification component calls GetAlertLogID to allocate an ID that allows it to accumulate errors separately within the AlertLog data structure. Accessing the IDs here allows the message filtering of a verification component to be controlled by the test. WaitForBarrier stops ControlProc until the test is complete.

```
ControlProc : process
begin
  SetAlertLogName("Test_UartRx_1");
  TBID <= GetAlertLogID("TB");
  RxID <= GetAlertLogID("UartRx_1");
  SetAlertLogId(SB, "UART_SB") ;
  SetLogEnable(PASSED, TRUE) ;
  SetLogEnable(RxID, INFO, TRUE) ;
  WaitForBarrier(TestDone, 5 ms) ;
  . . .
```

Figure 4. Test Initialization

8 A Simple Directed Test

A simple test can be created by transmitting (Send) a value on one interface and receiving (Get) and checking (AffirmIfEqual) it on another interface. The receiving and checking can also be done using the Check transaction (Get plus check inside the VC). This is shown in figure 5.

```
TxProc : process

begin
  Send (TRec, X"10") ;


  Send (TRec, X"11") ;
  . . .

end process TxProc
```

```
RxProc : process
  variable RxD : ByteType;
begin
  Get(RRec, RxD) ;
  AffirmIfEqual(TBID, RxD, X"10");

  Check(RRec, X"11") ;
  . . .
end process RxProc ;
```

Figure 5. A Simple Directed Test

The AffirmIfEqual checks its two parameters. It produces a log "PASSED" message if they are equal and alert "ERROR" message otherwise. These are shown in Figure 6.

```
%% Alert ERROR   In TB, Received: 08 /= Expected: 10 at 2150 ns
%% Log   PASSED  In TB, Received: 11 at 3150 ns
```

Figure 6. Messaging from AffirmIfEqual

See the AlertLogPkg_user_guide.pdf and AlertLogPkg_quickref.pdf for more information about OSVVM's affirmation capability (AffirmIf, AffirmIfEqual, AffirmIfDiff, …).

The Check transaction checks received value against the supplied expected value. It produces a log "PASSED" message if they are equal and alert "ERROR" message otherwise. These are shown in Figure 7. Note since source message ("In UartRx_1,") comes from the VC.

```
%% Alert ERROR   In UartRx_1, Received: 8 /= Expected: 10 at 2150 ns
%% Log   PASSED  In UartRx_1, Received: 11 at 3150 ns
```

Figure 7. Messaging from Check

9 Reusing Test Cases between Similar VC

The transactions supported are determined by the VC being used. That said, OSVVM takes a further step to improve reuse and simplify readability by introducing our Model Independent Transactions.

Model Independent Transactions observe that many interfaces can be classified as either an address bus interface (AXI, Wishbone, Avalon, X86) or a stream interface (AxiStream, UART, …). For interfaces that fall into one of these categories, OSVVM has defined a set of transactions the interface can support. The basic set of transactions supported by the different interfaces is shown in Table 1 below.

Table 1. OSVVM Standard Transactions

Address Bus Model Independent Transactions
Write(TransactionRec, X"AAAA", X"DDDD") ; **Read (TransactionRec, X"AAAA", DataOut) ;** **ReadCheck(TransactionRec, X"AAAA", X"DDDD") ;**
Stream Model Independent Transactions
Send(TransactionRec, X"DDDD") ; **Get(Transactionrec, DataOut) ;** **Check(TransactionRec, X"DDDD") ;**
Common Directive Transactions
GetTransactionCount(TransactionRec, Count) ; **SetModelOptions(TransactionRec, Option, OptVal) ;**

The transaction, SetModelOptions, facilitates configuring the VC to support different modes.

Switching between verification components that implement the OSVVM Model Independent Transactions, is primarily a matter of switching the verification component that is instantiated in the test harness. Hence, a test for an AXI Subordinate can be easily adapted for a Wishbone Subordinate VC.

For more details, see the Address_Bus_Model_Independent_Transactions_User_Guide.pdf and Stream_Model_Independent_Transactions_User_Guide.pdf. Also see Axi4_VC_user_guide.pdf and AxiStream_user_guide.pdf.

10 Using Randomization

Constrained random randomly selects test values, modes, operations, and sequences of transactions. In general, randomization works well when there are a large variety of similar items to test.

The OSVVM package, RandomPkg, provides a library of randomization utilities. A subset of these is shown in Figure 8.

```
-- Random Range:  randomly pick a value within a range
Data_slv8 := RV.RandSlv(Min => 0, Max => 15, 8) ;

-- Random Set: randomly pick a value within a set
Data1 := RV.RandInt( (1,2,3,5,7,11) ) ;

-- Weighted distribution:  randomly pick a value between 0 and N-1
-- where N is number of values in the argument
-- the likelyhood of each value = value / (sum of all values)
Data2 := RV.DistInt( (70, 10, 10, 5, 5) ;
```

Figure 8. Subset OSVVM's Random library

An OSVVM constrained random test consists of randomization plus code patterns plus transaction calls. For example, the code in Figure 9 generates a UART test with normal transactions 70% of the time, parity errors 10% of the time, stop errors 10% of the time, parity and stop errors 5% of the time, and break errors 5% of the time.

```
TxProc : process
  variable RV : RandomPType ;
  . . .
for I in 1 to 10000 loop
  case RV.DistInt( (70, 10, 10, 5, 5) ) is
    when 0  =>    -- Nominal case   70%
      ErrorMode := UARTTB_NO_ERROR ;
      TxD:= RV.RandSlv(0, 255, Data'length) ;
    when 1  =>    -- Parity Error   10%
      ErrorMode:= UARTTB_PARITY_ERROR ;
      TxD:= RV.RandSlv(0, 255, Data'length) ;
    when  . . .  -- (2, 3, and 4)

  end case ;

  Send(UartTxRec, Data, ErrorMode) ;
end loop ;
```

Figure 9. An OSVVM Constrained Random Test

Hence, creating constrained random tests in OSVVM is simply a matter of learning the patterns. All of the pattern is written directly in the code, and hence, visible to review.

Constrained random introduces two issues to our testing. First, how do we self-check the test? Previously we recreated the transmit pattern on the receive side. Due to the complexity, this would be tedious and error prone. In the next section, we solve this problem by using OSVVM's generic scoreboards.

Second, how do we prove the test actually did something useful? We solve this problem by using OSVVM's functional coverage.

See the RandomPkg_user_guide.pdf and RandomPkg_quickref.pdf for more information about OSVVM's randomization capability.

11 OSVVM's Scoreboards

A scoreboard facilitates checking data when there is latency in the system. A scoreboard receives the expected value from the stimulus generation process and checks the value when it is received by the check process, as shown in figure 10.

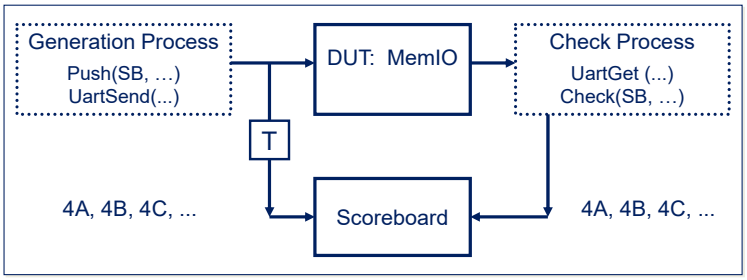

Figure 10. Scoreboard Block Diagram

The OSVVM scoreboard supports small data transformations, out of order execution, and dropped values. It uses package generics to allow the expected type and actual type to differ. The "match" function that determines if the expected and actual values match is also a package generic. The FIFO-like data structure of the scoreboard is created internal to a protected type.

The use model for OSVVM's scoreboard is shown in Figure 11. The scoreboard instance is created using a shared variable declaration. On the transmit side (TxProc), the expected value is pushed into the scoreboard (SB.Push), and then a transaction is transmitted (Send). On the receive side (RxProc), the transaction is received (Get), and then the received value is checked in the scoreboard (SB.Check). This greatly simplifies RxProc since it no longer reproduces what the transmit side did. Scoreboards can also be used to simplify checking in directed tests.

```
use work.ScoreboardPkg_uart.all ;
signal SbID : work.Scoreboardpkg_uart.ScoreboardIDType;
. . .
SB := NewID("SB") ;
```

```
TxProc : process
  variable RV   : RandomPType;
  . . .
begin
  for I in 1 to 10000 loop
    case RV.DistInt((. . .)) is
       . . .
     end case ;
    Push(SB, (TxD, ErrorMode));
    Send(TRec, TxD, ErrorMode);
  end loop ;
  . . .
```

```
RxProc : process
  variable RxD  : ByteType;
begin

  for I in 1 to 10000 loop

    Get(RRec, RxD, RxErrorMode);
    Check(SB, (RxD, RxErrorMode));
  end loop ;
  . . .
```

Figure 11. OSVVM Scoreboard Use Model

See the ScoreboardPkg_user_guide.pdf and Scoreboard_quickref.pdf for more information about OSVVM's scoreboard capability.

12 Adding Functional Coverage

Functional coverage is code that tracks items in the test plan. As such it tracks requirements, features, and boundary conditions. If a test uses constrained random, functional coverage is needed to determine if the test did something useful. Going further as design complexity increases, functional coverage is recommended to assure that a directed test actually did everything that was intended.

There are two categories of functional coverage: item (aka Point) coverage and cross coverage. Item coverage tracks relationships within a single object. For a UART, were transfers with no errors, parity errors, stop bit errors, parity and stop bit errors, and break errors seen?

Cross coverage tracks relationships between multiple objects. For a simple ALU, has each set of registers for input 1 been used with each set of registers for input 2?

Functional coverage in OSVVM is implemented as a data structure within a protected type.

Figure 12 continues with RxProc from the constrained random test and adds functional coverage. First the coverage object (RxCov) is declared. Next "AddBins (RxCov, GenBin(N))" is called to construct the functional coverage model. The value "N" corresponds to the integer representation of the UART status bits for Break, Stop, Parity, and Data Available. The calls to AddBins all complete at time 0, before any stimulus is generated or checked. Next, after the received stimulus has been retrieved (using Get), Icover(RxCov, RxErrorMode) is called to record the coverage. At the end of the test, WriteBin(RxCov) prints the coverage results.

```
architecture CR_1 of TestCtrl is
  signal RxCov : CoverageIDType ;  -- define coverage object
  . . .
begin
. . .
RxProc : process
  . . .
begin
  RxCov <= NewID("RxCov", TbID) ;
  -- Define coverage model
  AddBins(RxCov, GenBin(1) ) ;        -- Normal
  AddBins(RxCov, GenBin(3) ) ;        -- Parity Error
  AddBins(RxCov, GenBin(5) ) ;        -- Stop Error
  AddBins(RxCov, GenBin(7) ) ;        -- Parity + Stop
  AddBins(RxCov, GenBin(9, 15, 1) ) ; -- Break
  for I in 1 to 10000 loop
    Get(RRec, RxD, RxErrorMode);
    Check(SB, (RxD, RxErrorMode));
    ICover(RxCov, RxErrorMode) ;      -- Collect functional coverage
  end loop ;
  . . .
  WriteBin(RxCov) ;                   -- Print coverage results
```

Figure 12. UART RxProc with functional coverage added

See CoveragePkg_user_guide.pdf and CoveragePkg_quickref.pdf for more information about OSVVM's functional coverage capability.

Why not just use code coverage that is provided with a simulator? Code coverage only tracks code execution. Hence, code coverage cannot track the examples above since the information is not in the

code. On the other hand, if a design's code coverage does not reach 100% then there are untested items and testing is not done. Hence, both code coverage and functional coverage are needed to determine when testing is done.

13 Adding Protocol and Parameter Checkers

OSVVM alerts are used to check for invalid conditions on an interface or library subprogram. Alerts both report and count errors. Alerts have the levels FAILURE, ERROR (default), and WARNING. By default, FAILURE level alerts cause a simulation to stop. By default, ERROR and WARNING do not cause a simulation to stop. When a test completes, all errors reported by Alert (and AffirmIf) can be reported using ReportAlerts.

Figure 13 shows a protocol checker used in a memory model to detect if a write enable (iWE) and read enable (iOE) occur simultaneously while the memory is addressed (iCE). Parameter checkers are similar to protocol checkers and check for invalid parameters to library programs.

```
 SimultaneousAccessCheck: process
begin
  wait on iCE, iWE, iOE ;
  AlertIf(SramAlertID, (iCE and iWE and iOE) = '1',
      "nCE, nWE, and nOE are all active") ;
end process SimultaneousAccessCheck ;
```

Figure 13. Memory Model Protocol Checker

Alerts can be enabled (default) or disabled via a call to SetAlertEnable. The stopping behavior of Alert levels can be changed with SetAlertStopCount. Figure 14 shows the usage of both of these.

```
-- Turn off Warnings for a verification component
SetAlertEnable(UartRxAlertLogID, WARNING, FALSE) ;

-- If get 20 ERRORs stop the test
SetAlertStopCount(ERROR, 20) ;
```

Figure 14. Usage of SetAlertEnable and SetAlertStopCount

See the AlertLogPkg_user_guide.pdf and AlertLogPkg_quickref.pdf for more information about OSVVM's Alert capability (Alert, AlertIf, AlertIfEqual, SetAlertEnable, SetAlertStopCount, …).

14 Adding Message Filtering

OSVVM logs allow messaging to be turned on or off based on settings in the test – either globally or for a specific verification component. Logs have the levels ALWAYS, DEBUG, INFO, and PASSED. Logs print when enabled. Log ALWAYS is always enabled. The other logs are disabled by default. Figure 15 shows the usage of log and its output.

```
Log(TbID, "Test 1 Starting") ;
```

```
%% Log   ALWAYS   In Testbench, Test 1 Starting at 1770 ns
```

Figure 15. Log and its output

Logs are enabled or disabled using SetLogEnable. Figure 16 shows "PASSED" being enabled for the enitre testbench and "INFO" being enabled only for UartRx. Generally this is done in ControlProc at test initialization.

```
SetLogEnable(PASSED, TRUE) ;      -- Turn on PASSED for all VC
SetLogEnable(RxID, INFO, TRUE) ;  -- Turn on INFO for CpuID
```

Figure 16. SetLogEnable Usage

See the AlertLogPkg_user_guide.pdf and AlertLogPkg_quickref.pdf for more information about OSVVM's Alert capability (Log, SetLogEnable).

15 Test Finalization

Test finalization is error checking and reporting that is done in ControlProc after test completion. This is shown in Figure 17. Finalization starts when "WaitForBarrier(TestDone, 5 ms)" resumes. This happens either when all of the test processes have called their corresponding WaitForBarrier (normal completion) or 5 ms passes. The 5 ms is a test timeout (watch dog) that activates if one of the test processes did not complete properly. The sequence of calls to AlertIf check for proper test finish conditions. ReportAlerts prints test results (see Test Wide Reporting).

```
ControlProc : process
begin
  . . .
  WaitForBarrier(TestDone, 5 ms) ;
  AlertIf(TBID, NOW >= 5 ms, "Test timed out") ;
  AlertIf(TBID, not SB.Empty, "Scoreboard not empty") ;
  AlertIf(TBID, GetAffirmCount < 1, "Checked < 1 items") ;
  EndOfTestReports ;
  std.env.stop(GetAlertCount) ;
  wait ;
end process ControlProc ;
```

Figure 17. Test Finalization

See the TbUtilPkg_user_guide.pdf and TbUtilPkg_quickref.pdf for more information about OSVVM's process synchronization capability (WaitForBarrier, WaitForClock, …).

16 Test Reporting

At the end of the test, EndOfTestReports is called to create a text, YAML, HTML, and JUnit XML test reports. EndOfTestReports calls

- ReportAlerts to generate a test completion report with PASSED/FAILED,
- WriteAlertSummaryYaml to add an alert summary to the build report,
- WriteAlertYaml to generate a detailed tests alert report, and
- WriteCovYaml to generate a detailed coverage report.

For more details on EndOfTestReports see AlertLogPkg and ReportPkg User Guide.

16.1 Test Completion Reports via ReportAlerts – Text Based

The AlertLog data structure tracks FAILURE, ERROR, and WARNING for the entire test as well as for each AlertLogID (see GetAlertLogID). ReportAlerts prints a test completion message using this information.

For tests that do not use ID based reporting, ReportAlerts prints either a simple PASSED or FAILED message shown in Figure 18.

```
%% DONE  PASSED  Test_UartRx_1  Passed: 48  Affirmations Checked: 48  at 100000 ns
```

```
%% DONE  FAILED  Test_UartRx_1  Total Error(s) = 10  Failures: 0  Errors: 1  Warnings: 1
Passed: 48  Affirmations Checked: 48  at 100000 ns
```

Figure 18. ReportAlerts when not using ID based reporting

For tests that use ID based reporting, ReportAlerts prints a simple PASSED or a detailed FAILED message shown in Figure 19.

```
%% DONE  PASSED  TbUart_SendGet1  Passed: 48  Affirmations Checked: 48  at 100000 ns
```

```
%% DONE  FAILED  TbUart_SendGet1  Total Error(s) = 7  Failures: 0  Errors: 7  Warnings: 0
Passed: 41  Affirmations Checked: 48  at 100000 ns
%%    Default                     Failures: 0  Errors: 0  Warnings: 0  Passed: 0
%%    OSVVM                       Failures: 0  Errors: 0  Warnings: 0  Passed: 0
%%    TB                          Failures: 0  Errors: 0  Warnings: 0  Passed: 0
%%    AxiM_1                      Failures: 0  Errors: 0  Warnings: 0  Passed: 0
%%    UartTx_1                    Failures: 0  Errors: 0  Warnings: 0  Passed: 0
%%    UartRx_1                    Failures: 0  Errors: 7  Warnings: 0  Passed: 0
%%      UartRx_1: Data Check      Failures: 0  Errors: 7  Warnings: 0  Passed: 41
```

Figure 19. ReportAlerts when using ID based reporting

When the ReportAll parameter is set to TRUE (first parameter of EndOfTestReports), both PASSED and FAILED will produce a detailed print of all AlertLogIDs.

16.2 Alert Reports – YAML and HTML

The alert report is the first half of the detailed test report. The call to WriteAlertYaml produces a detailed report of alerts as a YAML data structure. After a simulation completes, the simulate procedure calls Simulate2Html which in turn calls Alert2Html to convert this YAML data structure into the first part of the detailed test report show in in Figure 20. The detailed test report is in the file ./reports/"test-name".html.

TbAxi4_RandomReadWrite Alert Report

▼ TbAxi4_RandomReadWrite Alert Settings

Setting		Value	Description
FailOnWarning		true	If true, warnings are a test error
FailOnDisabledErrors		true	If true, Disabled Alert Counts are a test error
FailOnRequirementErrors		true	If true, Requirements Errors are a test error
External	Failures	0	Added to Alert Counts in determine total errors
	Errors	0	
	Warnings	0	
Expected	Failures	0	Subtracted from Alert Counts in determine total errors
	Errors	0	
	Warnings	0	

▼ TbAxi4_RandomReadWrite Alert Results

Name	Status	Checks		Total Errors	Alert Counts			Requirements		Disabled Alert Counts		
		Passed	Total		Failures	Errors	Warnings	Passed	Checked	Failures	Errors	Warnings
TbAxi4_RandomReadWrite	PASSED	3000	3000	0	0	0	0	0	0	0	0	0
Default	PASSED	1750	1750	0	0	0	0	0	0	0	0	0
OSVVM	PASSED	0	0	0	0	0	0	0	0	0	0	0
subordinate_1	PASSED	0	0	0	0	0	0	0	0	0	0	0
subordinate_1: Protocol Error	PASSED	0	0	0	0	0	0	0	0	0	0	0
subordinate_1: Data Check	PASSED	0	0	0	0	0	0	0	0	0	0	0
subordinate_1: No response	PASSED	0	0	0	0	0	0	0	0	0	0	0
manager_1	PASSED	0	0	0	0	0	0	0	0	0	0	0
manager_1: Protocol Error	PASSED	0	0	0	0	0	0	0	0	0	0	0
manager_1: Data Check	PASSED	250	250	0	0	0	0	0	0	0	0	0
manager_1: No response	PASSED	0	0	0	0	0	0	0	0	0	0	0
manager_1: WriteResponse Scoreboard	PASSED	500	500	0	0	0	0	0	0	0	0	0
manager_1: ReadResponse Scoreboard	PASSED	500	500	0	0	0	0	0	0	0	0	0
manager_1: WriteBurstFifo	PASSED	0	0	0	0	0	0	0	0	0	0	0
manager_1: ReadBurstFifo	PASSED	0	0	0	0	0	0	0	0	0	0	0

Figure 20. Detailed Test Report – Alert portion

16.3 Coverage Reports - YAML and HTML

The alert report is the first half of the detailed test report. The call to WriteCovYaml produces a detailed report of the coverage models declared in a test as a YAML data structure. After a simulation completes, the simulate procedure calls Simulate2Html which in turn calls Cov2Html to convert this YAML data structure into the second part of the detailed test report shown in Figure 21. The detailed test report is in the file ./reports/"test-suite-name"/"test-case-name".html.

Uart7_Random_part3 Coverage Report

Total Coverage: 100.00

▼ UART_RX_STIM_COV Coverage Model Coverage: 100.0

▼ UART_RX_STIM_COV Coverage Settings

CovWeight	1
Goal	100.0
WeightMode	at_least
Seeds	824213985 792842968
CountMode	count_first
IllegalMode	illegal_on
Threashold	45.0
ThresholdEnable	0
TotalCovCount	100
TotalCovGoal	100

▼ UART_RX_STIM_COV Coverage Bins

Name	Type	Mode	Data	Idle	Count	AtLeast	Percent Coverage
NORMAL	Count	1 to 1	0 to 255	0 to 0	63	63	100.0
NORMAL	Count	1 to 1	0 to 255	1 to 15	7	7	100.0
PARITY	Count	3 to 3	0 to 255	2 to 15	11	11	100.0
STOP	Count	5 to 5	1 to 255	2 to 15	11	11	100.0
PARITY_STOP	Count	7 to 7	1 to 255	2 to 15	6	6	100.0
BREAK	Count	9 to 15	11 to 30	2 to 15	2	2	100.0
Total Percent Coverage:	100.0						

▼ UART_RX_COV Coverage Model Coverage: 100.0

▶ UART_RX_COV Coverage Settings

▼ UART_RX_COV Coverage Bins

Name	Type	Mode	Count	AtLeast	Percent Coverage
NORMAL	Count	1 to 1	70	1	7000.0
PARITY	Count	3 to 3	11	1	1100.0

Figure 21. Detailed Test Report – Coverage portion

16.4 Build Summary Report – YAML, HTML, and JUnit XML

The call to WriteAlertSummaryYaml The OSVVM scripts as well as each OSVVM test case writes to the file OsvvmRun.yml as a build is running. When build finishes, this file is renamed to "build-name".yml, and it is converted to both HTML and a JUnit XML file. A portion of the HTML based Test Suite report is shown in in Figure 22. The build summary report is in the file ./ "build-name".html.

OsvvmLibraries_RunAllTests Build Summary Report

Build	OsvvmLibraries_RunAllTests
Status	PASSED
PASSED	116
FAILED	0
SKIPPED	0
Elapsed Time (h:m:s)	0:04:48
Elapsed Time (seconds)	287.646
Date	2021-12-02T12:56-0800
Simulator	RivieraPRO
Version	RivieraPRO-2021.10.114.8313
OSVVM YAML Version	1.0

▼ **OsvvmLibraries_RunAllTests Test Suite Summary**

TestSuites	Status	PASSED	FAILED	SKIPPED	Requirements passed / goal	Disabled Alerts	Elapsed Time
Axi4Lite	PASSED	9	0	0	0 / 0	0	21.755
Axi4Full	PASSED	50	0	0	0 / 0	0	127.459
AxiStream	PASSED	49	0	0	0 / 0	0	116.419
Uart	PASSED	8	0	0	0 / 0	0	21.978

▼ **Axi4Lite Test Case Summary**

Test Case	Status	Checks passed / checked	Errors	Requirements passed / goal	Functional Coverage	Disabled Alerts	Elapsed Time
TbAxi4_BasicReadWrite	PASSED	60 / 60	0	0 / 0	-	0	1.054
TbAxi4_ReadWriteAsync1	PASSED	60 / 60	0	0 / 0	-	0	0.797
TbAxi4_ReadWriteAsync2	PASSED	60 / 60	0	0 / 0	-	0	0.923
TbAxi4_ReadWriteAsync3	PASSED	60 / 60	0	0 / 0	-	0	0.799
TbAxi4_RandomReadWrite	PASSED	3000 / 3000	0	0 / 0	-	0	0.996
TbAxi4_RandomReadWriteByte	PASSED	3000 / 3000	0	0 / 0	-	0	1.122
TbAxi4_TimeOut	PASSED	71 / 75	0	0 / 0	-	0	0.778
TbAxi4_WriteOptions	PASSED	72 / 72	0	0 / 0	-	0	0.772
TbAxi4_MemoryReadWrite1	PASSED	40 / 40	0	0 / 0	-	0	0.822

▼ **Axi4Full Test Case Summary**

Test Case	Status	Checks passed / checked	Errors	Requirements passed / goal	Functional Coverage	Disabled Alerts	Elapsed Time
TbAxi4_BasicReadWrite	PASSED	60 / 60	0	0 / 0	-	0	0.818
TbAxi4_RandomReadWrite	PASSED	3000 / 3000	0	0 / 0	-	0	1.031
TbAxi4_RandomReadWriteByte1	PASSED	3000 / 3000	0	0 / 0	-	0	1.337

Figure 22. Test Suite Summary Report

17 Building the OSVVM Libraries and Running the Testbenches

OSVVM is available on GitHub at https://github.com/OSVVM. The OSVVM repositories are all submodules of the OsvvmLibraries repository. Retrieve it using git as shown in Figure 23.

```
git clone --recursive https://github.com/OSVVM/OsvvmLibraries.git
```

Figure 23. Retrieving OSVVM from GitHub

The AXI4, Axi4Lite, AxiStream, and UART verification components come with OSVVM testbenches. Prior to starting the OSVVM scripting environment, create a directory named sim to run your simulations from. Start your simulator in the sim directory. Figure 24 shows the steps to run Mentor QuestaSim/ModelSim or Aldec RivieraPRO to build the OSVVM utility and verification component libraries and then run the verification component regression suite.

```
cd sim
source ../OsvvmLibraries/Scripts/StartUp.tcl
build  ../OsvvmLibraries/OsvvmLibraries.pro
build  ../OsvvmLibraries/RunAllTests.pro
```

Figure 24. Compiling and Running OSVVM

For more details on the OSVVM scripting environment see Script_user_guide.pdf. In particular, details to run Aldec's ActiveHDL, GHDL, Synopsys VCS and Cadence Xcelium are in the Script user guide.

18 About the OSVVM

The OSVVM utility and verification component libraries were developed and are maintained by Jim Lewis of SynthWorks VHDL Training. These libraries evolved from methodology and packages developed for SynthWorks' VHDL Testbenches and verification class.

Please support OSVVM by purchasing your VHDL training from SynthWorks.

19 About the Author - Jim Lewis

Jim Lewis, the founder of SynthWorks, has thirty plus years of design, teaching, and problem solving experience. In addition to working as a Principal Trainer for SynthWorks, Mr Lewis has done ASIC and FPGA design, custom model development, and consulting.

Mr. Lewis is chair of the IEEE 1076 VHDL Working Group (VASG) and is the primary developer of the Open Source VHDL Verification Methodology (OSVVM.org) packages. Neither of these activities generate revenue. Please support our volunteer efforts by buying your VHDL training from SynthWorks.

If you find bugs these packages or would like to request enhancements, you can reach me at jim@synthworks.com.

20 References

[1] https://blogs.sw.siemens.com/verificationhorizons/2020/12/16/part-6-the-2020-wilson-research-group-functional-verification-study/

[2] Jim Lewis, Advanced VHDL Testbenches and Verification, student manual for SynthWorks' class.