**Brief Summary of OSVVM in a Nutshell**

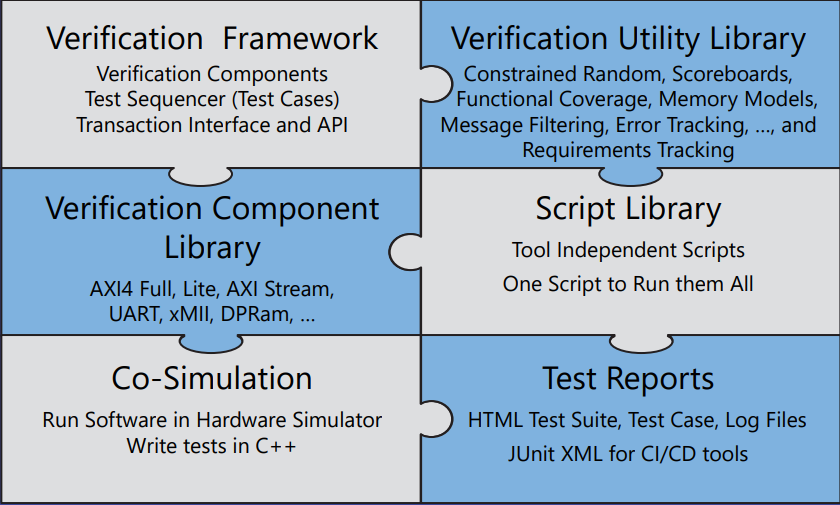
OSVVM is a free and open-source verification methodology designed to enhance VHDL testbenches. It provides a framework similar to SystemVerilog, enabling advanced verification techniques while maintaining ease of use for VHDL engineers.

Figure1: OSVVM Architecture

**Features of OSVVM**

**Verification Framework**

* Provides a structured test environment similar to SystemVerilog methodologies.
* Supports test sequencers, verification components, and transaction-based interfaces.

**Verification Components**

* Implements standard bus protocols such as AXI4 Full, AXI4 Lite, AXI Stream, UART, xMII, and Dual-Port RAM (DPRAM).
* Allows easy integration of reusable verification components.

**Test Sequencer**

* Handles test execution through a structured framework.
* Defines test cases as separate architectures for modularity

**Writing Directed Tests**

* Enables straightforward, deterministic test case development.
* Uses a procedural test case format.

**Constrained Random Tests**

* Uses a randomization library to generate test patterns dynamically.
* Supports constrained random testing for realistic test scenarios.  
  **Key Benefits**: Generates realistic stimulus in a timely fashion (to write)   
  Ideal for large variety of similar items :Modes, sequences, network packets, processor instructions, …

**Scoreboards**

* Provides a self-checking mechanism for verifying expected vs. actual results.
* Implements FIFO-based verification models

**Functional Coverage**

* Tracks test case effectiveness and identifies untested scenarios.
* Supports item coverage, cross-coverage, and coverage hole analysis

**Intelligent Coverage Randomization**

* Guides randomization based on uncovered functional coverage bins.
* Enhances test efficiency by focusing on missing test conditions

**Protocol and Parameter Checks**

* Implements protocol checks using alerts.
* Supports assertions and warnings for protocol violations.

**Test Reporting**

* Generates HTML-based test reports for test cases and test suites.
* Provides JUnit XML reports for CI/CD integration.

**Scripts**

* Tool-independent scripting for test execution across different simulators.
* Supports TCL-based procedures for automation and debugging.

**Benefits of** **OSVVM**  
Powerful and Concise – rivals other verification languages   
Unmatched reuse through the entire verification process  
Unmatched report capability with HTML for humans and JUnit XML for CI   
Tests are Readable and Reviewable by All   
Adopt incrementally as needed  
Tests and VC can be written by any VHDL Engineer

**OSVVM Verification Framework** looks identical to a SystemVerilog framework:

* Verification components (VC) implement interface signaling
* Test sequencer (TestCtrl) calls transactions = test case
* Each test case is a separate architecture of TestCtrl

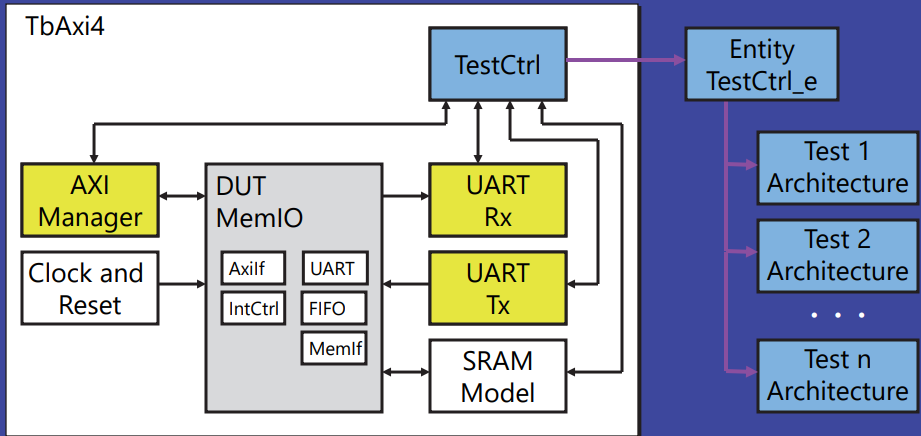


Figure2: OSVVM Example Verification Structure

Structural Code Plugs together just like RTL:

library osvvm, osvvm\_Axi4 ;

context osvvm.OsvvmContext ;

. . . entity TbAxi4 is

end entity TbAxi4 ;

architecture TestHarness of TbAxi4 is

. . .

signal ManagerRec : AddressBusRecType

( Address (AXI\_ADDR\_WIDTH-1 downto 0),

DataToModel (AXI\_DATA\_WIDTH-1 downto 0),

DataFromModel(AXI\_DATA\_WIDTH-1 downto 0) ) ;

begin

osvvm.TbUtilPkg.CreateClock(Clk, tperiod\_Clk) ;

osvvm.TbUtilPkg.CreateReset(nReset, . . .) ;

DUT\_1: DUT ( . . . ) ;

Axi4Manager\_1 : Axi4Manager (MRec, . . . ) ;

UartRx\_1 : UartRx(RxRec, . . . ) ;

UartTx\_1 : UartTx(TxRec, . . . ) ;

TestCtrl\_1 : TestCtrl (TxRec, RxRec, MRec, nReset) ;

end TestHarness ;

\*\*Verification Components, Test Sequencer, DUT

**Verification Framework – Elements:**

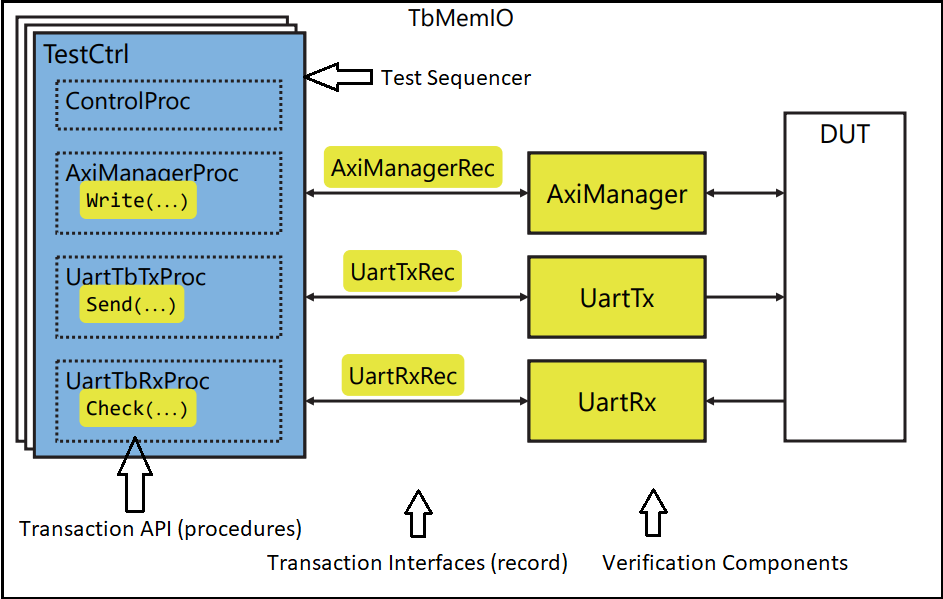


Figure3: Elements in Verification Framework

**Transaction Interfaces are Records:**

The record is an “inout” port that interfaces two components together.

Here is an Example record:  
type AddressBusRecType is record   
Rdy : RdyType ;   
Ack : AckType ;   
Operation : AddressBusOperationType ;   
Address : std\_logic\_vector\_max\_c ;   
AddrWidth : integer\_max ;   
DataToModel : std\_logic\_vector\_max\_c ;   
DataFromModel : std\_logic\_vector\_max\_c ;   
DataWidth : integer\_max ;   
. . .  
 end record AddressBusRecType ;

**Transaction API’s are VHDL Procedures**

procedure Read (  
 signal TransRec : InOut AddressBusRecType ;   
iAddr : In std\_logic\_vector ;   
variable oData : Out std\_logic\_vector ;  
 StatusMsgOn : In boolean := FALSE ) is   
begin   
-- Put Transaction into Record   
TransRec.Operation <= READ\_OP ;   
TransRec.Address <= SafeResize(iAddr, TransRec.Address'length) ;   
TransRec.AddrWidth <= iAddr'length ;   
TransRec.DataWidth <= oData'length ;  
TransRec.StatusMsgOn <= StatusMsgOn ;   
-- Handshake with Verification Component   
RequestTransaction(Rdy => TransRec.Rdy, Ack => TransRec.Ack) ;   
-- Get Results   
oData := SafeResize(TransRec.DataFromModel, oData'length) ;   
end procedure Read ;

\*\*\*Independent of VC by Putting Transaction into Record , Handshaking with VC and Getting results from record is a good abstraction strategy in OSVVM. Some interfaces do the same transactions

Address Bus Interfaces (AXI4, Avalon, …) do read and write

Streaming Interfaces (AxiStream, UART, …) do send and get

For these interfaces, Model Independent Transactions (MIT) define Transaction Interface (record) and Transaction API (procedures). This strategy combines common transactions to provide the same interface to different verification components.

|  |  |
| --- | --- |
| … Address Bus MIT (basic subset)  type AddressBusRecType is record . . . ;  Write(AddrRec, iAddr, iData) ;  Read (AddrRec, X"1111\_1110", oData) ;  . . . | … Stream MIT (basic subset) type StreamRecType is record . . . ;  Send (TxRec, iData [, iParam]) ;  Get (RxRec, oData [, oParam]) ;  . . . |

Benefits of OSVVM MIT  
**Accelerates Verification Component Development** Use the transaction interface and API are defined by MIT   
 Focus on writing VC behavior  
 A basic VC is as simple as writing a procedure  
**Accelerates Test Case Development**  
 Similar VC use the same transaction API  
 Share transaction sequences between similar VC  
**Co-Simulation is Supported**  
 OSVVM Co-Simulation supports all MIT based VC  
**Accelerates Documentation**  
 Only need to identify which transactions a VC supports   
  
\*More Information is in user guides in OsvvmLibraries/Documentation/ Address\_Bus\_Model\_Independent\_Transactions\_user\_guide.pdf Stream\_Model\_Independent\_Transactions\_user\_guide.pdf

**Verification Components**

entity Axi4Manager is

generic (   
 tperiod\_Clk : time := 10 ns ;   
 . . .   
 tpd\_Clk\_RReady : time := 2 ns   
) ;   
port (  
 -- Globals   
 Clk : in std\_logic ;   
 nReset : in std\_logic ;

-- AXI Master Functional Interface   
AxiBus : inout Axi4RecType ;

-- Testbench Transaction Interface   
TransRec : inout AddressBusRecType   
) ;

🡪DUT Interface

🡪Transaction Interface

TransactionHandler : process   
begin   
 WaitForTransaction(   
 Clk => Clk,   
 Rdy => TransRec.Rdy,   
 Ack => TransRec.Ack   
 ) ;

-- Decode and execute the transaction   
case TransRec.Operation is   
 when WRITE\_OP => AxiWrite(TransRec.Address, TransRec.Data, …);   
 when READ\_OP => AxiRead (TransRec.Address, TransRec.Data, …);   
 when . . . =>   
 -- Other Transactions   
end case ;

end process TransactionHandler ;

🡪Find Transaction in Record

🡪Do the Transaction

-----------------------------------------------------------------------------------------------------------------------------------------------------------

**TestCtrl is Test Sequencer**

entity TestCtrl is

port (   
TxRec : InOut StreamRecType ;   
RxRec : InOut StreamRecType ;   
ManagerRec : InOut AddressBusRecType ;  
   
nReset : In std\_logic   
) ;   
end TestCtrl ;

🡪These ports are Transaction Interfaces

**TestCtrl Architecture: Big Picture**   
architecture UartTx1 of TestCtrl is   
. . .   
begin   
ControlProc : process   
begin   
 . . .   
 WaitForBarrier(TestDone, 5 ms) ;   
 EndOfTestReports ;   
 std.env.stop;   
end process ;   
AxiManagerProc : process   
begin   
 wait until nReset = '1' ;   
 Write(. . .) ;   
 WaitForBarrier(TestInit);   
 . . .   
 WaitForBarrier(TestDone) ;  
 end process ;  
TxProc : process  
begin   
 WaitForBarrier(TestInit);   
 Send(. . .) ;  
 . . .   
 WaitForBarrier(TestDone) ;   
end process;   
. . .

**Aspects of a Test Case**

• Whole test in one file

• Control Process

• Initialize & finalize test

• One process per interface

• Concurrent, just like design

• Tests =

• Calls to transactions

• Easy to add and mix in

• Directed Tests

• Constrained Random

• Scoreboards

• Functional Coverage

• Synchronization

• Error Reporting & Messaging

**Test Steps in OSVVM Methodology**

OSVVM follows a structured verification approach to ensure thorough and efficient testing of VHDL designs. Below are the key steps involved in an OSVVM-based test:

1. Test Initialization

* Set Up Logging and Alerts: Initialize logging mechanisms to capture test results.
* Define Test Identifiers: Assign unique IDs to different test components (e.g., transactions, scoreboards).
* Open Test Reports: Create structured test reports in HTML or XML format.
* Enable Message Filtering: Configure error reporting levels (e.g., ERROR, WARNING).

2. Test Sequencing

* Define Test Sequences: Implement test cases using a sequencer (TestCtrl).
* Synchronize Execution: Use barriers (WaitForBarrier) to synchronize test processes.
* Run Transaction-based Tests: Execute bus transactions using a well-defined API.

3. Apply Stimuli

* Directed Testing: Apply pre-determined test patterns.
* Constrained Random Testing: Use OSVVM's randomization library to generate test vectors dynamically.
* Intelligent Coverage Randomization: Generate stimuli targeting uncovered test conditions.

4. Response Checking and Assertions

* Use Scoreboards: Automatically compare actual vs. expected values using FIFO-based structures.
* Perform Protocol and Parameter Checks: Use alerts to validate timing and interface requirements.
* Implement Functional Coverage: Track coverage metrics to ensure all test conditions are exercised.

5. Test Finalization

* Check for Test Completion: Wait for all transactions to finish or timeout conditions.
* Generate Reports:
  + Test case reports (HTML, XML, text).
  + Functional coverage reports.
  + Scoreboard reports.
* Stop Simulation: Ensure the simulation halts upon completion.

6. Regression Testing and Automation

* Use OSVVM Scripts: Automate compilation, simulation, and reporting.
* Run Continuous Integration (CI): Generate JUnit reports for integration into CI/CD workflows.

**Test Case = Send, Get, and Check transactions + affirmations (checks)**

**Affirmations signal pass/fail as well as count errors and checks**  
%% 2150 ns Log PASSED In TB, Received: 10  
%% 2150 ns Alert ERROR In TB, Received: 08 /= Expected: 10  
Benefit: Improves readability. Simplifies writing self-checking tests.

• For requirements, affirmations also count errors and checks

AffirmIfEqual(ReqID1, RxD, X"10") ;   
. . .   
AffirmIf(ReqID1, TRUE, "Req1 Passed") ;   
. . .   
AffirmIf("Req2.1", A=B, "Require that A: " & to\_string(A) & " = B: " & to\_string(B) ) ;  
  
\*Note: A failed requirement is both a requirements failure and a test failure

**Alerts Simplify Protocol Checks SimultaneousAccessCheck:**   
process   
begin   
 wait on iCE, iWE, iOE ;   
 AlertIf((iCE and iWE and iOE) = '1', "iWE and iOE both active");   
end process SimultaneousAccessCheck ;   
  
%% 5500 ns Alert ERROR iWE and iOE both active

Alert Levels: FAILURE, ERROR (default), WARNING   
Alerts are used for protocol and parameter checks   
Alerts are enabled by default and rarely disabled

**Controls: StopCount, PrintCount, Enable/Disable**   
SetAlertStopCount(ERROR, 20) ; -- Stop when 20   
SetAlertPrintCount(CpuID, ERROR, 10) ; -- Limit printing   
SetAlertEnable(WARNING, FALSE) ; -- Disable Alerts

**Logs Simplify Debug**

• Logs are conditional printing (messaging)

Log(TbID, "Sequence 1 Starting", ALWAYS) ;   
. . .   
Log(TbID, "Test Last Failed Here", DEBUG) ; -- Disabled  
  
%% 2200 ns Log ALWAYS In TB, Sequence 1 Starting--log output

Log Levels: ALWAYS (default), DEBUG, INFO, FINAL, PASSED  
Logs only print when enabled, Message with level DEBUG does not print since it is disabled  
  
**Controls: Enable/Disable**  
SetLogEnable(DEBUG, FALSE) ; -- Disable Alerts

**OSVVM Randomization Library**

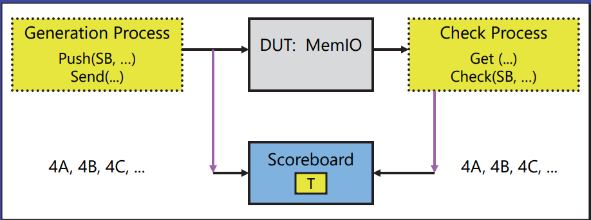
Randomize a value in an inclusive range, 0 to 15, except 5 & 11  
Data1 := RV.RandInt(Min => 0, Max => 15) ;   
Data2 := RV.RandInt(0, 15, Exclude => (5,11) ) ;

Randomize a value within the set (1, 2, 3, 5, 7, 11), except 5 & 11  
Data3 := RV.RandInt( (1,2,3,5,7,11) ) ;   
Data4 := RV.RandInt( (1,2,3,5,7,11), Exclude => (5,11) ) ;

Weighted Randomization: Weight, Value = 0 .. N-1  
Data5 := RV.DistInt ( (7, 2, 1) ) ;  
  
Weighted Randomization: Value + Weight  
. . . -- ((val1, wt1), (val2, wt2), ...)   
Data6 := RV.DistValInt( ((1,7), (3,2), (5, 1)) ) ;

\*\*By itselfi this is not constrained random

**Scoreboards**

Simplify self-checking when data is minimally transformed  
Internally it is a FIFO + Checker  
Uses package generics to support different types  
 Handles small data transformations  
Handles out of order execution & Handles dropped values

**Functional Coverage**

**What: Code that tracks that items in the test plan occur** Tracks requirements, features, and boundary conditions

**Why?**   
With Randomization, how do you know what the test did?   
 Test Done = Functional Coverage and Code Coverage @ 100 %

**Item Coverage (aka Point Coverage)**Track relationships within a single object  
 Bin transfer sizes into: 1, 2, 3, 4-127, 128-252, 253, 254, 255

**Cross Coverage**   
Track relationships between independent objects   
Has each set of registers been used with each input of an ALU?

**Why not just use code coverage?**   
Code coverage tracks code execution   
Misses anything not in code (bins, uncorrelated items)

**CoveragePkg simplifies coverage definition, collection, and reporting**Internally it has a data structure and configuration parameters   
Implemented as a singleton in CoveragePkg   
The singleton API defines the coverage capabilities

**OSVVM Creates Unmatched Test Reports**

OSVVM Test Completion Message

Build Summary Mini-Report – Text

Build Summary Report – HTML

Summary of entire build  
 Summary of each test suite in the build  
 Summary of each test case within a test suite

Requirements Summary – HTML and CSV

Test Case Reports – HTML

Reports on Alert, Functional Coverage, and Scoreboards  
 Links to HTML simulation transcript and test file output

HTML'ized simulation transcript / log file (simulator output)

Errors shown in red

JUnit Report – XML for Continuous Integration Tools

For more see: https://osvvm.github.io/Overview/Osvvm3Reports.html

**Getting OSVVM & Running Scripts**

Get the sources: (Alternately, a zip file is at: osvvm.org/downloads)  
**git clone --recursive** [**https://github.com/osvvm/OsvvmLibraries**](https://github.com/osvvm/OsvvmLibraries)

Initialize the simulator – see Documentation/Scripts\_user\_guide.pdf  
**source <path-to-osvvmlibs>/OsvvmLibraries/Scripts/StartUp.tcl**

Build all OSVVM and Run All VC Tests

build $OsvvmLibraries/OsvvmLibraries.pro   
build $OsvvmLibraries/RunAllTests.pro  
\*Each VC has a RunAllTests and RunDemoTests

**OSVVM Resources**

Documentation  
HTML: https://osvvm.github.io/Overview/Osvvm1About.html   
 PDF: OsvvmLibraries/Documentation - in OSVVM release

• Forum: <https://osvvm.org>

Recorded Webinars  
• OSVVM: Leading Edge Verification for the VHDL Community • <https://www.youtube.com/watch?v=KVmGDy_PHNI>

Faster than Lite Verification Component Development with OSVVM  
• <https://www.aldec.com/en/support/resources/multimedia/webinars/2187>  
  
OSVVM’s Test Reports and Simulator Independent Scripting  
• <https://www.aldec.com/en/support/resources/multimedia/webinars/2188>

Advances in OSVVM’s Verification Data Structures  
• <https://www.aldec.com/en/support/resources/multimedia/webinars/2190>

Jump start your VHDL verification effort with training  
Advanced VHDL Testbenches and Verification – OSVVM Boot Camp  
• <https://synthworks.com/vhdl_testbench_verification.htm>