OSVVM and UVVM Summary

Mert Ecevit

April 2025

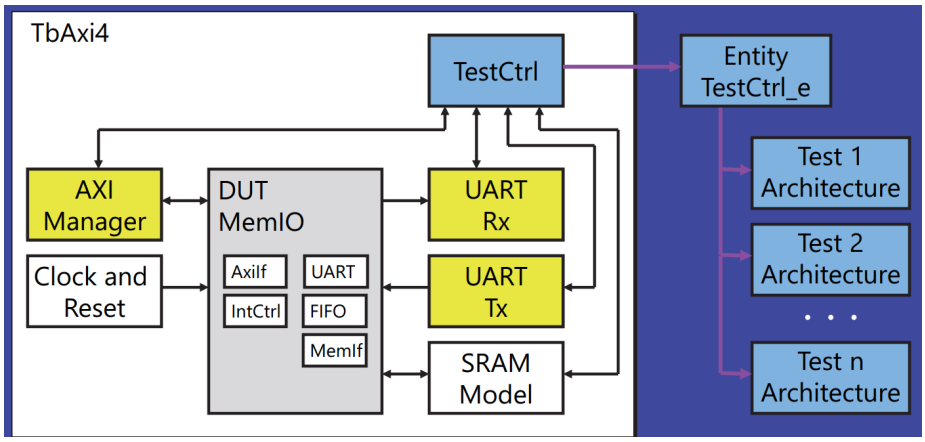

Figure 1: OSVVM VERIFICATON FRAMEWORK

OSVVM Methodology

0.1 OSVVM Verification Framework

Looks identical to a SystemVerilog framework:

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

0.1.1 Elements

- Transaction Interface + Transaction API
- Verification Components
- Test Sequencer

0.2 Transaction Interfaces and APIs

Transaction interfaces written with VHDL record types and Transaction APIs written with VHDL procedures.

- The record is "inout" port
- Long term plan of for this part is to migrate VHDL-2019 Interfaces

Model Independent Transactions (MIT) define standardized interfaces and APIs that can be applied across multiple related protocols.

For example, the **Address Bus MIT** is designed for memory-mapped interfaces such as AXI4 and Avalon, while the **Stream MIT** supports stream-based protocols like AXI Stream and UART.

By using MIT-based Verification Components, the development of test case components becomes significantly more efficient and consistent.

0.3 Test Sequencer = TestCtrl

- All test logic is kept in a single file for simplicity and maintainability.
- A control process governs the test execution flow, managing initialization and finalization.
- Each interface is assigned a separate process, enabling concurrency similar to the design.
- Each interface process is responsible for handling its own set of transactions.
- Test scenarios are constructed as sequences of transaction calls.
- The architecture supports the integration of:
  - Directed tests,
  - Constrained-random testing,
  - Scoreboards for checking results,
  - Functional coverage collection.
- Synchronization mechanisms ensure proper coordination among parallel processes.
- Error reporting and messaging are included to assist in debugging and traceability.

0.4 Constrained Random Testing

- Efficiently produces realistic stimuli suitable for driving the design under test.
- Well-suited for scenarios involving numerous similar or repeated structures.
- Capable of handling various types of data and operations such as:
  - Operational modes,
  - Instruction sequences,

  - Network packet flows,
  - Processor-level command execution.

OSVVM has a Randomized Library to test and check. It is more easy to detect the errors.

0.4.1 Scoreboards

Self-checking when data is minimally transformed.

- A built-in verification mechanism is used to ensure correct data transformations.
- This is typically implemented through customized FIFOs that include automatic checking logic.
- It is designed to be flexible by using package-level generics, allowing compatibility with various data types.
- The system supports advanced behaviors such as handling out-of-order data processing and managing dropped or lost values.

0.5 Functional Coverage

- Cross coverage monitors how independent elements relate to one another.
- For example, it helps determine whether each register set has been tested with every ALU input.
- It involves writing code that checks whether key items in the test plan are being exercised.
- It ensures coverage of requirements, features, and edge conditions.
- Item coverage, also known as point coverage, focuses on variation within a single object.
- An example is categorizing transfer sizes into bins such as: 1, 2, 3, 4–127, 128–252, 253, 254, and 255.
- The purpose is to track what the randomized testing actually stimulated.
- A test is considered complete when both functional coverage and code coverage reach 100%.
- Code coverage alone is not sufficient.
- Code coverage only reflects which lines of code were executed.
- It fails to capture conditions outside the code itself, such as bin coverage or uncorrelated combinations.

0.6 Scripting System

- Designed to be compatible with various simulators, ensuring platform-independent usage.
- Provide automation for tasks such as library setup, code compilation, and simulation execution.
- Support a wide range of simulators including GHDL, NVC, Aldec, Siemens, Synopsys VCS, and Cadence Xcelium.
- Scripts are structured in a modular way to enhance reusability and maintainability.

0.7 Reporting Features

- Mini-Report in Text Format
- Detailed HTML Report
  - Overview of the complete build process
  - Breakdown for each test suite
  - Individual test case summaries within suites

1. Build Summary Report (HTML Format)

- Comprehensive overview of the complete build execution
- Summarized results for all test suites and individual test cases
- Direct access points to in-depth test documentation

2. Detailed Test Case Documentation (HTML)

- System alert documentation highlighting errors and warnings
- Coverage analysis reports for functional verification
- Verification scoreboard results
- Quick-access links to simulation output transcripts

3. Enhanced Simulation Transcript (HTML)

- Critical errors prominently displayed in red
- Expandable/collapsible content sections
- Optimized for rapid analysis of extensive output logs

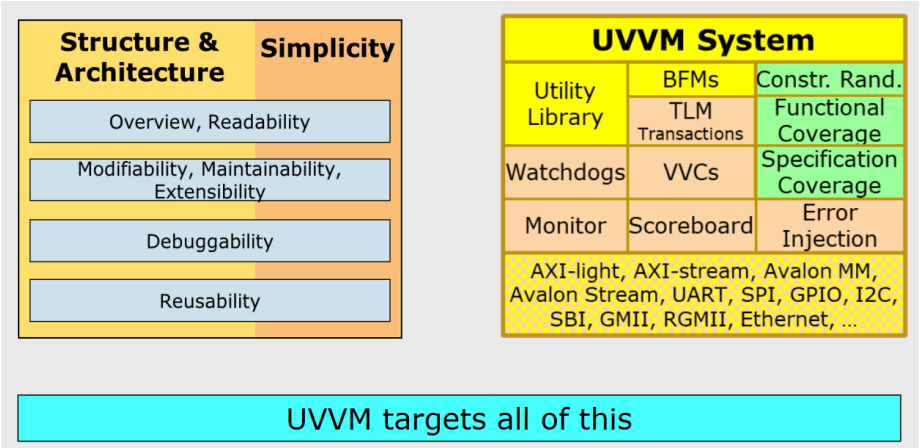

Figure 2: UVVM System

4. Requirements Verification Report

- Clear visualization of requirements coverage status

UVVM Methodology

What is UVVM?

- A comprehensive VHDL verification framework and methodology
- Freely available under open-source licensing
- Features well-organized system architecture and infrastructure
- Delivers substantial improvements in verification productivity
- Enhances final design quality and reliability
- Doulos-recommended approach for testbench development
- Currently being expanded through ESA (European Space Agency) initiatives
- IEEE Standards Association open-source development project
- Compatible with all VHDL-2008 compliant simulation tools

0.8 Utilities

- check_stable()
- await_stable()
- clock_generator()
- adjustable_clock_generator()
- random()
- randomize()
- gen_pulse()
- block_flag()
- unblock_flag()
- await_unblock_flag()
- await_barrier()
- enable_log_msg()
- disable_log_msg()
- to_string()
- fill_string()
- to_upper()
- replace()
- normalize_and_check()
- set_log_file_name()
- set_alert_file_name()
- wait_until_given_time_after_rising_edge()

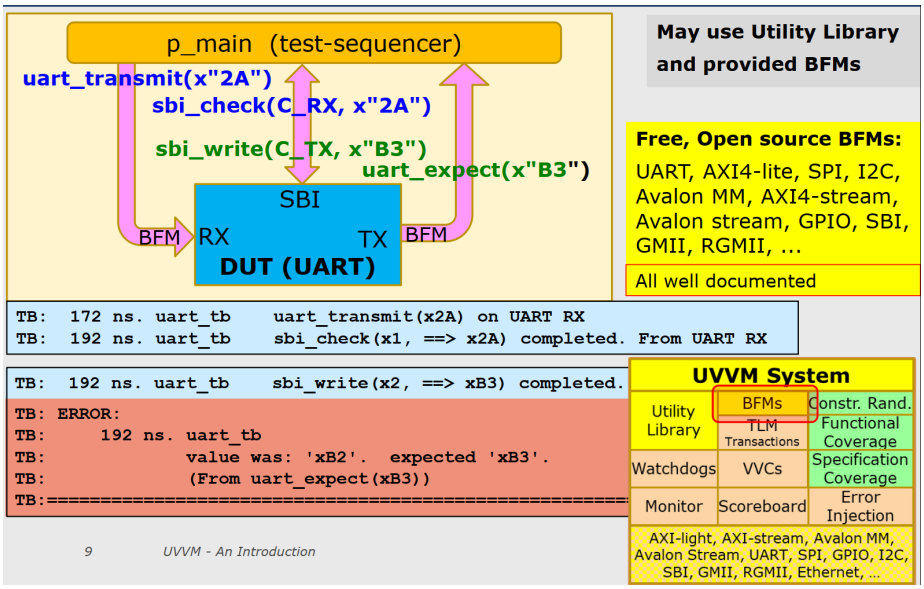

Figure 3: Simple Data Communication

0.9 Bus Functional Models(BFMs)

- **Advantages for Simple Testbenches:**
  - Implemented as specialized procedures in dedicated packages
  - Straightforward invocation from any process
- **Limitations:**
  - Processes have sequential execution constraints
  - Can only perform one BFM operation at a time
  - Cannot simultaneously execute multiple BFMs or other operations
- **Solution for Concurrent Operations:**
  - Requires implementation as separate entities/components
  - Verification Components (VCs) provide this capability

0.10 VHDL Verification Components(VVCs)

VVCs enhance traditional BFMs with advanced capabilities through a structured design:

Core Components

- **Command Interpreter:** Analyzes and decodes incoming commands
- **Command Queue:** Manages pending operations in FIFO order
- **Command Executor:** Processes commands using the underlying BFM

Advanced Features

- Configurable delays between commands for timing verification
- Queue management for handling multiple concurrent operations
- Automatic detection of command sequence completion
- Real-time interface activity monitoring
- Multicast/broadcast support for multi-agent control
- Detailed transaction logging and reporting
- Built-in sequencers for protocol automation

Extension Capabilities

- Plug-in protocol checkers (timing, framing, intervals)
- Custom sequence generator integration
- Support for complex transactions:
  - Split transactions
  - Out-of-order execution

0.11 Advanced Functions of UVVM

- **Scoreboards** - Verification data checking
- **Monitors** - Interface activity observation
- **Randomization Control** - Functional coverage management
- **Error Injection**:
  - Brute force methods
  - Protocol-aware techniques
- **Local Sequencers** - Command sequence automation
- **Property Checkers** - Protocol rule verification
- **Transaction Info** - Detailed operation logging

- **Watchdogs**:
  - Simple timeout detection
  - Activity-based monitoring
- **Hierarchical VVCs** - With integrated scoreboards
- **Specification Coverage** - Requirements-to-test tracing

0.12 Specification Coverage

Verification Steps

1. Specify all requirements
2. Report coverage from test sequencers (or other testbench components)
3. Generate summary reports

Coverage Reporting

- Coverage metrics per individual requirement
- Test cases associated with each requirement
- Requirements verified by each test case
- Cumulative coverage across multiple test cases

0.13 2nd ESA Project

- **Advanced randomization** implemented with simple interface
  - **Optimized Randomization**:
    - Randomization without replacement
    - Weighted according to:
      - Target distribution
      - Previous events
    - Achieves the lowest number of randomizations needed for a given target
  - **Functional Coverage**:
    - Verifies that all specified scenarios have been tested

  Also the Enhanced Randomisation of UVVM is structed.

- **Seamless integration** with UVVM framework
  - Unified **alert handling** system

  - Consolidated **logging** mechanism

- **Enhanced clarity and usability**
  - Intuitive **keyword system** for improved comprehension
  - Clean, organized output for better **overview**

- **Maintainability advantages**
  - Modular design for **easy extension**
  - Straightforward **code maintenance**