Verification Course Exercises

Fall 2025

Contents

1 Introduction

The following document will provide to the reader an introduction to Universal Verification Methodology (UVM) verification through exercises to implement a testbench (TB) using several open-source tools.

The goal of these exercises is to:

- Understand the basic architecture of a UVM testbench using *PyUVM*;
- Understand the vertical reuse concept by integrating a Universal Verification Component (*uVC*) in a UVM testbench;
- Implement a library of tests and a library of virtual sequences, to fully verify the behavior of the Device Under Test (DUT);
- Implement UVM components, such as the Coverage and the Scoreboard, to collect testbench and design metrics.

For the exercises provided in document, it is assumed that the reader has received the source code of the UVM testbench, which is the intended template to support the exercises' development. In the document, the `<ROOT>` directory shall map to the base folder containing the source code received.

2 Design of the Saturation Filter

A *Saturation Filter* is a device that saturates the input when this exceeds certain limits. The design provided implements this device and will be used as the DUT (Device Under Test) for the following exercises. The device saturates the input data when this exceeds the threshold value defined.

2.1 Implementation

The Figure 2.1 shows the high-level diagram of the *Saturation Filter*, showing the input and output ports of the design.

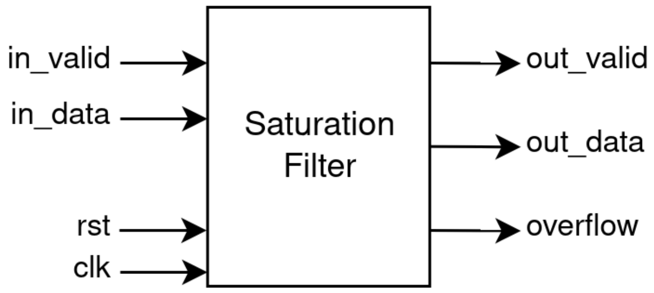

Fig. 2.1: *Saturation Filter* module.

The *Saturation Filter* I/O ports are the following:

- `clk`, the clock signal;
- `rst`, the reset signal;
- `in_valid`, the input valid signal;
- `in_data`, the input data sample;
- `out_valid`, the output valid signal;
- `out_data`, the output data sample;
- `overflow`, the output overflow signal.

The *Saturation Filter* module contains the following parameters that can be tuned to change the module operation:

- `DATA_W`, sets the data width of the data signal;
- `THRESHOLD`, the value over which to saturate the data.

The device uses the SyoSil Data Transfer Protocol (*sSDT*) Section 3, and was developed respecting the following requirements:

- **RS/01**: It shall be possible to reset the state of the saturation filter by toggling the `rst` input signal.
- **RS/02**: All signals shall react to the rising edge of the input `clk` signal.
- **RS/03**: The saturation functionality of the design shall be evaluated with the `THRESHOLD` parameter value. If the data is below the limits it is propagated to the output on the next rising edge of the `clk` signal. Otherwise, the data is `saturated` and the `out_data` signal will be the `THRESHOLD` value instead.
- **RS/04**: The `in_valid` signal is always propagated to the output on the next rising edge of the `clk`.
- **RS/05**: The *Saturation Filter* shall be compliant with the *sSDT* (simple SyoSil Data Transfer) protocol described in the following section.

3 Simple SyoSil Data Transfer Protocol

The simple SyoSil Data Transfer (*sSDT*) protocol is a simple synchronous data transfer protocol. The protocol has two signals, data and valid, and has two variants, the "producer" and the "consumer", which are modifying the direction of the signal as showed in the Table 3.1.

Signal Name	**Producer direction**	**Consumer direction**	**Comment**
valid	output	input	when asserted, data is valid
data	output	input	the data of the protocol

Table 3.1: *sSDT* protocol signals.

The protocol operation must follow these protocol requirements (PR):

1. **Reset requirement (PR1)**: when `rst` is high, `valid` can not be high
2. **Data validity requirement (PR2)**: when `vaild` is high, `data` must have a valid value
3. **Data invalidity requirement (PR3)**: when `valid` is low, `data` must be `0`

A timing diagram outlining the behavior of the protocol is shown in Figure 3.1. All requirements should be valid when checked synchronously to the clk.

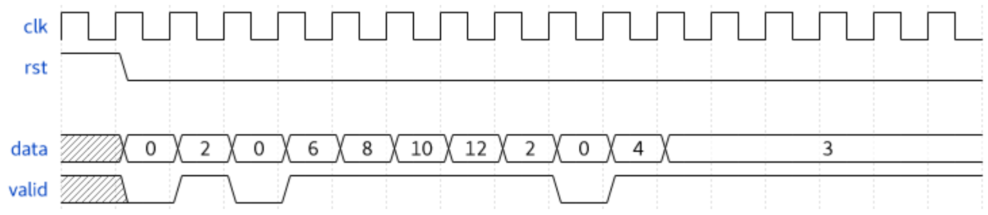

Fig. 3.1: *sSDT* protocol waveform.

4 *sSDT* Universal Verification Component

The *sSDT* Universal Verification Component (*uVC*) is implemented with the intention of generating traffic compliant with the protocol requirements. By developing the *sSDT uVC* it is ensured the reuse of the same component in multiple testbenches, without having to re-implement the code from the beginning. The Figure 4.1 shows the high-level diagram of the *sSDT uVC*.

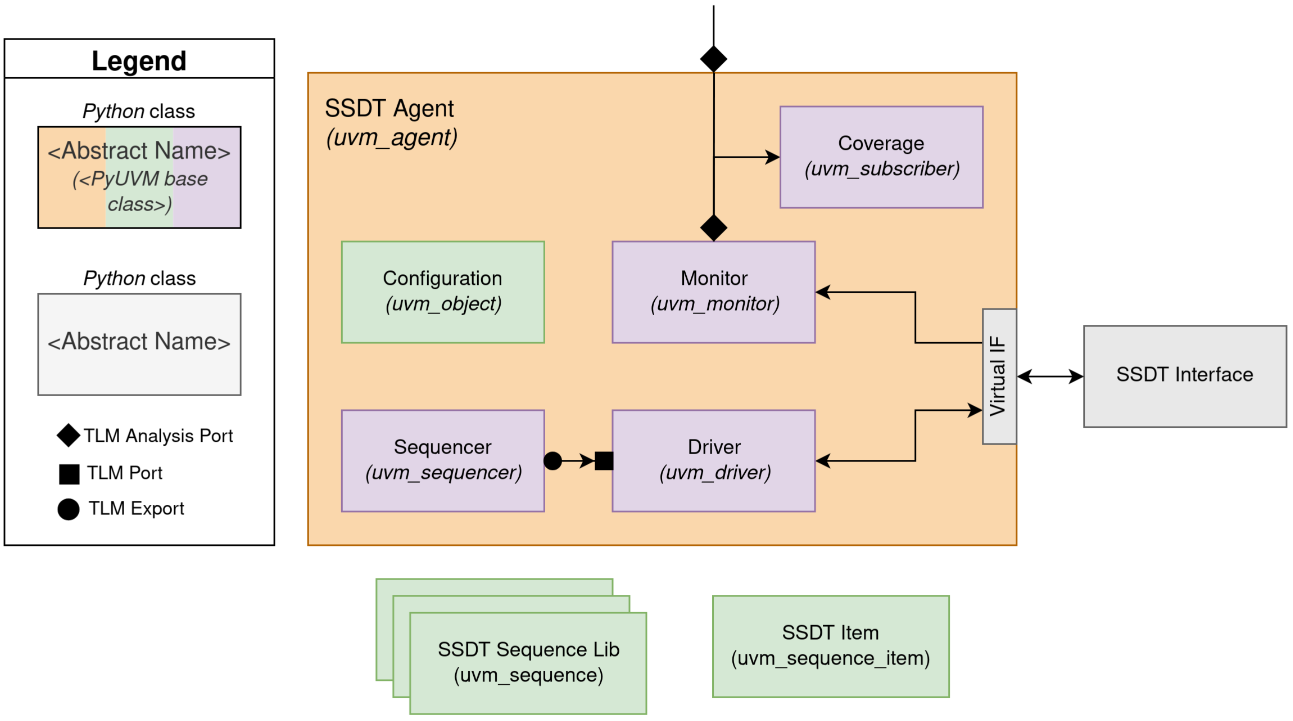

Fig. 4.1: *sSDT uVC* diagram.

The UVM components described in the Figure 4.1 can be found in the locations defined in the Table 4.2. Consider for the `<ROOT_sSDT>` the following path, `<ROOT>/sat_filter/src/tb/uvc/ssdt/src`.

Component	**Path**
SSDT Agent	`<ROOT_sSDT>/uvc_ssdt_agent.py`
Configuration	`<ROOT_sSDT>/uvc_ssdt_config.py`
Monitor	`<ROOT_sSDT>/uvc_ssdt_monitor.py`
Driver	`<ROOT_sSDT>/uvc_ssdt_driver.py`
Sequencer	`uvm_sequencer` base class
Coverage	`<ROOT_sSDT>/uvc_ssdt_coverage.py`
SSDT Interface	`<ROOT_sSDT>/uvc_ssdt_interface.py`
SSDT Sequence Lib	`<ROOT_sSDT>/uvc_ssdt_sequence_lib.py`
SSDT Item	`<ROOT_sSDT>/uvc_ssdt_seq_item.py`

Table 4.2: *sSDT* UVC files location.

5 Back2Back Testbench

The Back2Back testbench is developed with the intention of testing the Universal Verification Component (*uVC*) which is responsible for generating the traffic in compliance with the *sSDT* protocol.

As showed in the figure 5.1, two *sSDT* agents will be instantiated in the testbench and connected to the same interface. The "producer" shall drive the data on the interface, while the "consumer" will sample the content of the interface and broadcast them to the testbench.

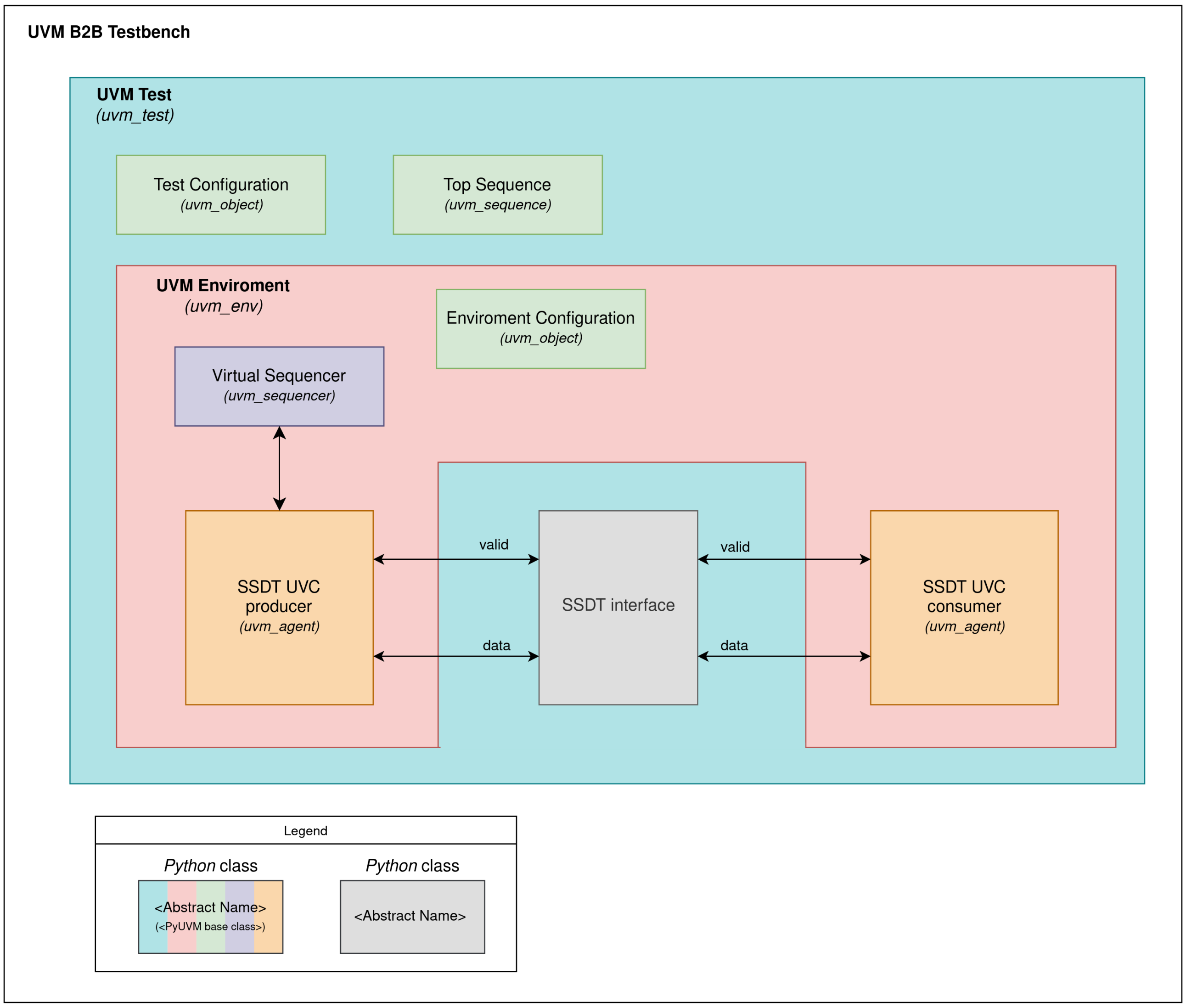

Fig. 5.1: UVM testbench for the back2back tests for the *sSDT uVC*.

6 Generic Testbench Architecture

The Figure 6.1 shows the high level structure of the *Saturation Filter* UVM testbench. Analyzing the legend of the diagram it can be seen that most of the classes implemented and used in this testbench are UVM classes. All these base classes represent the "backbone" of the *PyUVM* library that aims to implement the UVM verification methodology in the Python programming language.

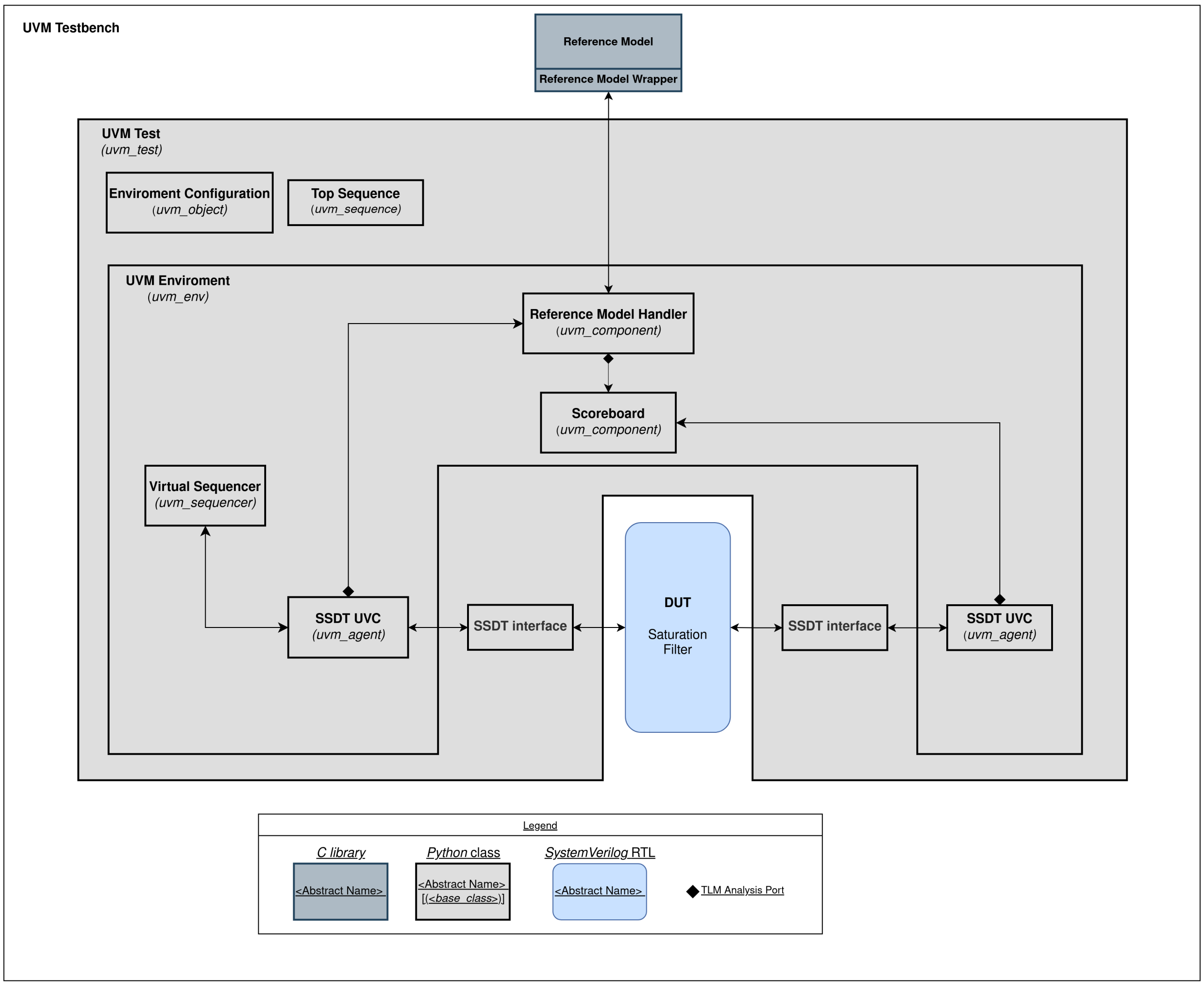

Fig. 6.1: UVM testbench for the *Saturation Filter*.

6.1 Base Test

All the testbench components are created and connected in the base test, which is an extension of the `uvm_test`. The test defines the handles for the environment class (`UVM Environment`), for the configuration object class (`Test Configuration`), the sequence class (`Top Sequence`), and for the interface classes required to communicate with the *uVC* (`ssdt interface`).

6.1.1 Build Phase

In the `build_phase` all the class' instances are created. This is achieved with the create method which invokes the class constructor. The major benefit of using the create method is to register the class in the `UVM Factory`. The registration mechanism improves the efficiency and flexibility of the UVM testbenches by allowing the user to do instance overrides and checks for registered type. The interface class is not part of the *PyUVM* library, so the creation cannot be performed with the create method.

During the `build_phase` it is also possible to set the environment and the configuration object classes inside the configuration database (`ConfigDB`) to easily pass these class instances to the entire testbench.

6.1.2 Connect Phase

After all the components have been created, the base test shall connect them as necessary during the `connect_phase`. This implies the correct connection of the DUT signals to the two *sSDT* interfaces, one used by the producer and the other by the consumer.

6.1.3 Run Phase

In contrast with the other two functions mentioned above, the `build_phase` and `connect_phase`, which can be considered as regular Python functions, the `run_phase` on the other hand is a coroutine. This implies that the `run_phase` takes simulation time to run and is the only place where other coroutines can be launched. The `run_phase` is the main simulation phase of the `uvm_test` responsible for running configuration and data traffic sequences to the DUT.

6.2 Environment

The environment component, extended from `uvm_env`, defines handles for the configuration object, the virtual sequencer and the necessary *uVC*'s agents. In the scope of the *Saturation Filter* testbench, the environment contains one *sSDT uVC* acting as a producer and another *sSDT uVC* acting as a consumer. If there is a reference model, the environment shall define handles for the reference model handler and the scoreboard.

6.2.1 Build Phase

In the `build_phase` all the handles previously defined are created similarly as for base test (e.g. using the `create` method). Furthermore, the configuration object can be taken from the `ConfigDB`, through the `get` method, if it was already set by the base test. The configuration object can then be used to update the configuration for all agents inside the environment.

6.2.2 Connect Phase

The `connect_phase` is responsible for the correct connection of the instantiated components. The connection between the sequencers inside the *uVC* agents and the virtual sequencer is done by passing the appropriate handler. Another type of connection is the one between the *uVC* agents and the scoreboard, respectively with the reference model. This is based on a TLM interface, in which analysis port from the agents are connected with analysis export from the scoreboard and reference model.

6.3 Virtual Sequencer

The virtual sequencer defines handles only for the configuration object and the necessary *uVC*'s sequencers. The connection with other testbench components (e.g. agent's sequencers) is completed by the environment.

6.4 Test Configuration

The test configuration, extended from `uvm_object`, creates all *uVC*'s configuration objects. As was previously explained for the base test, the method create is called for all these components. For the *Saturation Filter* testbench, the `data_width` and `threshold` are the configurable parameters.

6.5 Reference Model

The reference model defines handles for the FIFOs that are connected to *uVC*'s analysis ports.

6.5.1 Build Phase

The FIFOs are created in the `build_phase` of the reference model calling the `uvm_tlm_analysis_fifo` constructor. In the case of the *Saturation Filter* testbench, two FIFOs were created, one for each *uVC*.

In addition to the FIFOs, the `build_phase` of the reference model also creates an analysis port for the communication with the scoreboard component. This is required for sending the reference model items the expected output for the comparison that takes place inside the scoreboard. A coroutine has been defined to make sure that the reference model receives the same inputs as the DUT and generates outputs that are defined to be correct.

6.5.2 Run Phase

The `run_phase` of the reference model is in charge with launching this coroutine.

6.6 Scoreboard

The scoreboard receives both the output from the DUT and from the reference model. The communication with the *uVC*'s and the reference model it is done using `TLM Analysis Ports`. In the *Saturation Filter* testbench the scoreboard defines handles for two FIFOs and two corresponding queues. The first pair is responsible for monitoring and sending the consumer items, while the second one is in charge with the reference model items.

6.6.1 Run Phase

In the `run_phase`, the scoreboard checks the content of the two queues, element by element. If a mismatch is found, an error is asserted.

A Appendix

`Note` The commands listed below assume that the user is in the testbench directory (e.g. `<ROOT>/sat_filter/src/tb`).

- `make`
  Run the tests. All tests found will be run with default values for DUT parameters values.
- `make MODULE=<test-name>`
  Run a specific test. It will only simulate the test named `<test-name>` with the default values for the DUT parameters.
  e.g. `make MODULE=test_sat_filter_default_seq`.
- `make MODULE=<test-name> THRESHOLD=<threshold-value> DATA_W=<data_w-value>`
  Run a specific test with different values for the DUT parameters.
  e.g. `make MODULE=test_sat_filter_default_seq DATA_W=16 THRESHOLD=8`.
- `make clean`
  Deleting all build and log files associated with the tests.
- `make coverage`
  Merge all tests coverage results and view the output in `PyUCIS-viewer` tool.
- `pyucis-viewer sim_build/<test-name>_cov.txt`
  Visualize the coverage report for a particular test case using the `PyUCIS-viewer` tool.
  e.g. `pyucis-viewer sim_build/test_sat_filter_default_seq.xml`
- `gtkwave sim_build/<waveform-name>.vcd`
  Visualize the waveform using the `GTKWave` tool.
  e.g. `gtkwave sim_build/sat_filter_waves.vcd`

.