



FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING  
DEGREE PROGRAMME IN ELECTRONICS AND COMMUNICATIONS ENGINEERING

## **MASTER'S THESIS**

# **UVM TESTBENCH IN PYTHON: FEATURE AND PERFORMANCE COMPARISON WITH SYSTEMVERILOG IMPLEMENTATION**

Author	Miikka Sinervä
Supervisor	Juha Häkkinen
Second Examiner	Jukka Lahti
Technical Advisor	Kimmo Varjonen

June 2023

**Sinervä M. (2023) UVM Testbench in Python: Feature and performance comparison with SystemVerilog implementation.** University of Oulu, Faculty of Information Technology and Electrical Engineering, Degree Programme in Electronics and Communications Engineering. Master's Thesis, 60 p.

## **ABSTRACT**

Python is emerging as a new language for functional verification of digital integrated circuits (ICs). With the Python verification framework *cocotb* enabling to write testbenches in Python, new libraries are being developed for various verification techniques and methodologies, such as functional coverage, constrained random verification and Universal Verification Methodology (UVM). Python testbenches have been used in some research and product development, but there is little information available on their performance, and no studies about applying UVM in Python have been published.

In this thesis, a Python UVM testbench was developed using *pyuvvm* and other Python verification libraries for an AHB-Lite slave IP, and a matching testbench in SystemVerilog was also built to examine the differences in their implementations. Testbench codebase sizes, simulation execution times, memory use and coverage accumulation were compared. The Python testbench had 30% less lines of code, suggesting that testbench development may be faster in Python than SystemVerilog. The execution times of the Python testbench on commercial simulators were 8 to 21 times longer than those of the SystemVerilog testbench in tests with AHB-Lite write operations and random stimulus.

In conclusion, given the performance gap and the UVM Register Abstraction Layer (RAL) being at an early stage of development in *pyuvvm*, the studied Python libraries are not competitive with SystemVerilog and its UVM implementation for verifying complex designs like systems-on-chip (SoCs) at this stage. Nevertheless, *pyuvvm* enables Python programmers and users of open-source simulators without support for SystemVerilog UVM to start using the methodology. A Python UVM testbench based on *pyuvvm* is currently viable for verifying simple designs, and it opens new avenues of research in digital IC verification.

**Key words:** *pyuvvm*, *cocotb*, integrated circuit (IC), register-transfer level (RTL) verification.

**Sinervä M. (2023) UVM-testipenkki Pythonilla: Ominaisuuksien ja suorituskyvyn vertailu SystemVerilog-toteutuksen kanssa.** Oulun yliopisto, tieto- ja sähkötekniikan tiedekunta, elektroniikan ja tietoliikennetekniikan tutkinto-ohjelma. Diplomityö, 60 s.

## TIIVISTELMÄ

Python on nousemassa uudeksi kieleksi digitaalisten integroitujen piirien varmennukseen. *Cocotb*-viitekehys mahdollistaa testipenkkien kirjoittamisen Pythonilla, ja uusia Python-kirjastoja kehitetään eri varmennusmenetelmille, kuten funktionaaliselle kattavuudelle, rajoitetulla satunnaisherätteellä verifiointille ja universaalille varmennusmenetelmälle (engl. Universal Verification Methodology, UVM). Python-testipenkkejä on pienissä määrin käytetty tutkimuksissa ja tuotekehityksessä, mutta niiden suorituskyvystä on hyvin vähän tietoa, ja UVM:n käytöstä Pythonilla ei ole julkaistu tutkimuksia.

Tässä työssä kehitettiin UVM-testipenkki Pythonilla AHB-Lite-orjana toimivalle IP-lohkolle käyttäen *pyuvvm*:ää ja muita Python-verifiointikirjastoja, ja vastaava testipenkki luotiin myös SystemVerilogilla toteutusten vertailua varten. Testipenkeistä verrattiin koodikannan kokoa, suoritusaikaa, muistin käyttöä ja kattavuuden kertymistä. Python-testipenkeissä oli 30 % vähemmän koodirivejä, mikä voi merkitä, että testipenkkien kehittäminen Pythonilla on nopeampaa kuin SystemVerilogilla. Suoritusajat kaupallisilla simulaattoreilla oli Python-testipenkillä 8–21 kertaa pidempiä kuin SystemVerilog-testipenkillä testeissä, joissa ajettiin AHB-Lite -kirjoitusoperaatioita ja satunnaisherätettä.

Koska suorituskykyero oli näin merkittävä, ja koska UVM:n rekisteriabstraktiotaso (engl. Register Abstraction Layer, RAL) on vasta alkutekijöissään *pyuvvm*:ssä, voidaan todeta, että tutkitut Python-kirjastot eivät ole vielä nykyisellä tasollaan kilpailukykyisiä SystemVerilogin ja sen UVM-implementaation kanssa monimutkaisten piirien kuten järjestelmäpiirien varmennukseen. Siitä huolimatta *pyuvvm* mahdollistaa UVM:n käytön Python-ohjelmoijille ja avoimen lähdekoodin simulaattoreissa, joissa ei ole vielä SystemVerilog UVM:lle tukea. *Pyuvvm*-pohjainen Python UVM-testipenkki soveltuu tällä hetkellä yksinkertaisten mallien varmennukseen ja avaa uusia tutkimussuuntia digitaalisten integroitujen piirien varmennukseen.

**Avainsanat:** *pyuvvm*, *cocotb*, integroitu piiri, RTL-varmennus.

# TABLE OF CONTENTS

ABSTRACT

TIIVISTELMÄ

TABLE OF CONTENTS

FOREWORD

LIST OF ABBREVIATIONS AND SYMBOLS

1	INTRODUCTION .....	7
2	SIMULATING A PYTHON TESTBENCH .....	9
	2.1 Communication between Python testbench and simulator .....	9
	2.2 Coroutines .....	10
	2.3 A simple testbench .....	10
3	UVM IN PYTHON.....	14
	3.1 Universal Verification Methodology .....	14
	3.2 Python and SystemVerilog.....	16
	3.3 pyuvm.....	18
	3.3.1 Class definition and object creation .....	20
	3.3.2 Phasing .....	22
	3.3.3 Configuration database .....	23
	3.3.4 Interfacing with the DUT .....	24
	3.3.5 Transaction-level modelling .....	27
	3.3.6 Reporting system.....	30
	3.3.7 UVM Register Abstraction Layer .....	31
	3.3.8 Defining test cases .....	32
4	TESTBENCH SUPPLEMENTATION .....	33
	4.1 Functional coverage .....	33
	4.2 Constrained random verification.....	36
	4.3 Passing objects between Python and SystemVerilog.....	37
5	TEST SETUP .....	39
	5.1 The device under test.....	39
	5.2 The testbenches .....	39
6	PERFORMANCE AND CODEBASE COMPARISON .....	42
	6.1 Code size comparison.....	42
	6.2 Performance tests .....	43
	6.2.1 Idle tests .....	43
	6.2.2 Write tests .....	44
	6.2.3 Random stimulus tests .....	46
	6.3 Coverage test .....	47
7	DISCUSSION .....	49
8	SUMMARY .....	52
9	REFERENCES .....	53
10	APPENDICES .....	55

## FOREWORD

This thesis was conducted at Nokia to study the viability of using Python for developing UVM testbenches. I would like to thank Nokia and Sami Jylhä for providing me with the opportunity to work on this thesis. I want to thank Kimmo Varjonen for his help with technical aspects of the work and with finalizing the thesis. I also want to thank Juha Häkkinen for supervising the thesis and for helping with structuring and polishing the work. I also wish to thank Jukka Lahti for the second examination of the thesis and for his lectures in digital electronics that sparked my interest in the functional verification of integrated circuits.

Oulu, June 22, 2023

Miikka Sinervä

## LIST OF ABBREVIATIONS AND SYMBOLS

AHB	Advanced High-Performance Bus
API	application programming interface
ASIC	application-specific integrated circuit
DPI	direct programming interface
DUT	device under test
EDA	electronic design automation
FIFO	first-in, first-out
FLI	Foreign Language Interface
FPGA	field-programmable gate array
GPI	Generic Procedural Interface
HDL	hardware description language
IC	integrated circuit
IP	intellectual property
RAL	Register Abstraction Layer
RTL	register-transfer level
SMT	satisfiability modulo theories
SoC	system-on-chip
SPI	Serial Peripheral Interface
SV-UVM	SystemVerilog Universal Verification Methodology
TLM	transaction-level modelling
UCIS	Unified Coverage Interoperability Standard
UVM	Universal Verification Methodology
VIP	Verification Intellectual Property
VHPI	VHDL Procedural Interface
VPI	Verification Procedural Interface
VSZ	Virtual memory size

# 1 INTRODUCTION

An essential part of developing digital integrated circuits (ICs) is to verify their functionality before mass production is started. One step of this verification process is to create a testbench that applies stimuli to a register-transfer level (RTL) model of the design in a hardware description language (HDL) simulator to make sure that the model meets its specifications. The model needs to be synthesizable to logic gates and eventually to transistors, which sets limitations on how the designer can write the HDL code that defines the hardware. The same rules do not apply to the testbench that will not be synthesized, but its code is still typically written in a hardware description and verification language called SystemVerilog. It includes both synthesizable and non-synthesizable constructs, forming a complex language of 248 keywords [1 p. 1182–1183].

In recent years, there has been exploration on using Python for RTL verification. Python is currently the most popular programming language, with over 440 000 extension packages that verification engineers could also benefit from [2][3]. One open-source package that enables using Python for verification is *cocotb*. Initially released in 2013, *cocotb* framework contains modules to write testbenches and verification code in Python for (System)Verilog and VHDL models. *Cocotb* works on various free and proprietary simulators. [4][5][6]

Python testbenches based on *cocotb* are already in use in academia and the industry. For example, Ashmanskas et al. (2023) successfully verified three application-specific integrated circuits (ASICs) for CERN’s High Luminosity Large Hadron Collider with *cocotb*. The authors commended the Python-based verification for its ease of development [7]. The 2022 Wilson Research Group Functional Verification Study, where study participants from the IC/ASIC and field-programmable gate array (FPGA) market were surveyed, found that *cocotb* was used in 5% of the IC/ASIC and FPGA projects [8][9].

In 2020, an open-source package leveraging *cocotb* to implement Universal Verification Methodology (UVM) in Python, *pyuvvm*, was introduced. UVM is a standardized class library of SystemVerilog testbench components and utilities. The purpose of UVM is to reduce verification complexity and to improve interoperability by providing a consistent methodology which both internal and external teams in a verification project can apply [10 p. 7]. UVM makes it easier to reuse verification components in new testbenches with its modular, object-oriented design [10 p. 12]. *Pyuvvm* is an effort to implement UVM in Python while taking advantage of the characteristics of the language [11].

The goal of this thesis is to build an intellectual property (IP) level UVM testbench in Python using *pyuvvm* and other available Python libraries and to evaluate its viability for RTL verification. Another testbench is developed in SystemVerilog with the same functionality and the same tests to the extent that is possible to compare the features, codebase size and performance. Functional coverage and constrained random verification are included in both implementations. Applying the methodologies in Python is explained and compared to SystemVerilog in detail. This approach aims to build a basic understanding of developing UVM testbenches in Python for further research on the subject.

The main content of the thesis is split into five chapters. Chapter 2 introduces *cocotb* for building Python testbenches in a non-UVM context. Chapter 3 focuses on *pyuvvm*, evaluating its feature availability and describing how its implementation of UVM is different from the SystemVerilog version with testbench code examples. Chapter 4 presents two ways to add constrained random stimulus and functional coverage to a Python testbench: with a Python and C implementation of the features with *PyVSC*, or with an interface to a SystemVerilog module with *pyquesta*. Chapter 5 is a high-level overview of the Python and SystemVerilog testbenches

built for performance tests. Chapter 6 presents a codebase comparison of the testbenches and test simulations with their results. In addition, the discussion chapter evaluates the achieved goals and results of the thesis and considers possibilities for further research, and the summary chapter provides a brief summary of the thesis.



## 2 SIMULATING A PYTHON TESTBENCH

To understand how a Python UVM testbench operates, one must first know the basics of the underlying coroutine-based cosimulation testbench environment, *cocotb*. The following sections will explain how *cocotb* communicates with a simulator, the concept of coroutines and its relation to *cocotb*, and provide an example of a simple Python testbench and its operation.

### 2.1 Communication between Python testbench and simulator

A Python testbench built with *cocotb* executes outside the HDL simulator. Whenever the testbench needs to interact with the simulator, *cocotb* does it with a call through a language interface. Simulators provide foreign language interfaces that allow external C/C++ programs to interact with RTL tools and access or affect the state of the RTL model [1 p. 953][12]. Since *cocotb* testbenches are written in Python, another type of interface is also needed between Python and C++. The connection between the testbench and the simulator is pictured below in Figure 1. [13]

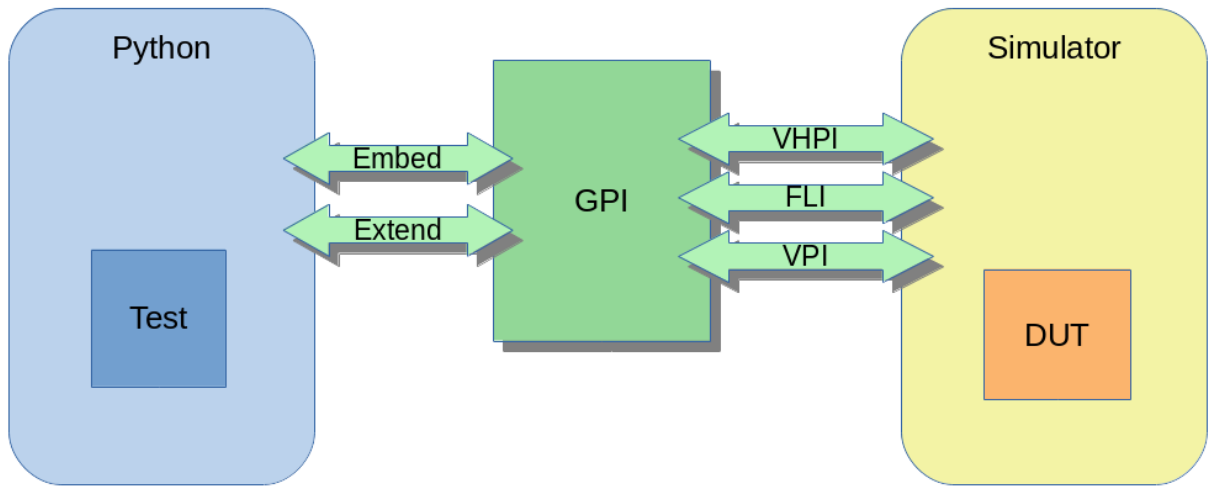


Figure 1. Communication between a Python testbench and an HDL simulator with *cocotb*.

IEEE Standard for VHDL language specifies VHDL Procedural Interface (VHPI). It provides access to a VHDL model and a simulator with a set of VHPI functions [12]. Siemens' Questa Sim uses its own proprietary and non-standard Foreign Language Interface (FLI) in addition to VHPI. The SystemVerilog Verification Procedural Interface (VPI), specified in the SystemVerilog Language Reference Manual, is used to access SystemVerilog objects [1 p. 953].

These three interfaces pass information between a simulator and an abstraction layer written in C++ called Generic Procedural Interface (GPI). GPI hides implementation details of VHPI, VPI and FLI from the Python side by having common routines to access all three of them. [13][14]

Python provides Python/C application programming interface (API) to write extension modules in C or C++, or to embed the Python interpreter in C/C++ modules [15]. *Cocotb* uses this API to embed the Python interpreter into the simulator process to launch the Python environment at the start of the simulation run. It also has a C++ extension module to provide the Python side with functions that use GPI to access the simulator.

## 2.2 Coroutines

A subroutine is a piece of code, stored in one place, that can be executed from an outside program with a call. The program jumps to execute that piece of code, and then jumps back to where it left. Subroutines allow reusing the same code without having to duplicate it to each place where its functionality is needed. [16]

A coroutine is a general case of a subroutine. With two linked coroutines, there is no main routine and a subroutine – they can be considered equal. The main difference of the two types is that a coroutine will continue from where its execution was last suspended, while a subroutine will always start from the beginning. [16]

Coroutines are a heavily used concept in cocotb. They serve the same purpose as a SystemVerilog task: they can contain simulation time consuming constructs. A coroutine may, for example, schedule signal value changes in the device under test (DUT), and then yield control to another coroutine while it waits for the next rising edge of a clock signal. Once all active coroutines are in a waiting state, the execution can be passed to the simulator to advance the simulation. [17]

Below is an example of a coroutine in a Python testbench with cocotb. The function is defined as a coroutine with *async* keyword. First, it schedules a value change of *aresetn* signal in the DUT to 0. The next line's *await* keyword means that the coroutine will give up control of the execution until the coroutine following the *await* keyword returns. In this case, the testbench will continue executing other coroutines – or wait if nothing else is scheduled – until five clock cycles have passed. After that, the program returns to the reset coroutine to schedule a change in *aresetn* value to 1.

```
async def reset(dut):
    dut.aresetn.setimmediatevalue(0)
    await ClockCycles(dut.clk, 5)
    dut.aresetn.value = 1
```

## 2.3 A simple testbench

Let us build a testbench consisting of a few coroutines to see how it is constructed and how multiple coroutines interact. The DUT will be a VHDL module that has a clock input *clk*, an asynchronous reset input *aresetn*, and one 32-bit register that can be written to with *data\_in* input and read from *data\_out* output.

First, the Python modules used in this testbench are imported. The Python standard library module *random* is used for generating random numbers. Cocotb also groups related functionality into individual modules. The module *clock* contains a class for clock generation, and the module *triggers* provides triggers for signal transitions and coroutine synchronization, among others.

```
import random
import cocotb
from cocotb.clock import Clock
from cocotb.triggers import ClockCycles, RisingEdge
```

The reset coroutine from the previous section will be used to reset the DUT. Let us add a coroutine for writing the register.

```

async def write(dut, data):
    dut.data_in.value = data
    await RisingEdge(dut.clk)

```

The *write* coroutine writes the given data to *data\_in* and waits one clock cycle before another write can be executed. Another coroutine will monitor that the data is written correctly, which is shown below.

```

async def monitor_write(dut):
    await RisingEdge(dut.aresetn)
    await RisingEdge(dut.clk)
    while True:
        in_val = dut.data_in.value
        await RisingEdge(dut.clk)
        out_val = dut.data_out.value
        assert (
            in_val == out_val
        ), f"data_out is {hex(out_val)}, expected {hex(in_val)}"

```

The Python keyword *assert* works such that if the given condition is false, the program will raise *AssertionError* – a Python exception. Exceptions determine the outcome of the simulation: if no exceptions are raised, the test passes successfully.

Now we still need the main test coroutine that starts the clock, calls the monitor and the reset coroutines, and uses the write coroutine to drive stimulus to the DUT. This is described below.

```

@cocotb.test()
async def simple_test(dut):
    dut.data_in.setimmediatevalue(0)
    cocotb.start_soon(Clock(dut.clk, 100, units="ns").start())
    cocotb.start_soon(reset(dut))
    cocotb.start_soon(monitor_write(dut))
    await RisingEdge(dut.aresetn)
    await RisingEdge(dut.clk)
    for _ in range(5):
        await write(dut, random.randrange(0, 2**32))
    await ClockCycles(dut.clk, 3)

```

The first line is a Python decorator, which is a wrapper function that is used to modify the behaviour of or add functionality to the function or the class below it. In this case, it marks *simple\_test* as the test coroutine to be run at the start of the simulation. The decorator can be used for coroutines, classes or any callable objects. There can be multiple tests with the decorator in a module, in which case the default behaviour is to run all the tests sequentially within one simulation.

The *dut* object passed to the test gives access to the RTL model in its full hierarchy. *dut.submodule1.submodule2. ... submoduleN).signal.value* is a handle linked to the corresponding signal in the DUT and can be used for both reading and writing the signal value. The first line inside *simple\_test* coroutine also shows the special method *setimmediatevalue*, which makes a read immediately following the write to return the new value. The default behaviour is to return the old value until the change is applied to the DUT. The read value is returned in *BinaryValue* type, which models logic states like SystemVerilog's *logic* type with built-in type conversions. The *dut* object can also be accessed through the variable *cocotb.top*.

The three lines with *cocotb.start\_soon* schedule the three coroutines to be run concurrently as soon as *simple\_test* yields control at the next *await* statement. *Clock* is a class from *cocotb*'s

*clock* module for generating a clock signal with a given clock period. Using *await* instead of *cocotb.start\_soon* for *Clock.start* would result in the simulation hanging, since the coroutine never returns. Finally, the *for* loop drives random input data to the DUT, while the *monitor\_write* coroutine compares the input to the output in the background.

Cocotb uses Make to build the test run environment. The Makefile for simulating this test is shown below. It tells where to find the testbench and the HDL module files, which simulator to use, and it provides custom settings for cocotb and the HDL simulator. *MODULE* points to the Python module containing the tests, and *TOPLEVEL* defines the HDL module to be used as the DUT. The final line integrates the file to cocotb's build flow, which includes simulator specific Makefiles with simulation options that enable the Python testbench's access to the simulator. The build flow can be fully customized as long as the simulation options mandatory for cocotb are included.

```
CWD=$(shell pwd)
export PYTHONPATH := $(CWD)/../tb:$(PYTHONPATH)
SIM ?= questa
TOPLEVEL_LANG ?= vhdl
VHDL_SOURCES = $(CWD)/../src/simple.vhd
ifeq ($(SIM),questa)
    WAVES = 1
endif
export COCOTB_RESOLVE_X ?= ZEROS
export COCOTB_ANSI_OUTPUT=1
MODULE := simple_test
TOPLEVEL = simple
include $(shell cocotb-config --makefiles)/Makefile.sim
```

The results of this simulation are presented in Figure 2. No assertions failed, so the test passed.

```
#      0.00ns INFO      Found test simple_test.simple_test
#      0.00ns INFO      running simple_test (1/1)
#     1301.00ns INFO      simple_test passed
#     1301.00ns INFO      *****
#                               ** TEST                STATUS  SIM TIME (ns)  REAL TIME (s)  RATIO (ns/s) **
#                               *****
#                               ** simple_test.simple_test      PASS           1301.00         0.01      131485.74 **
#                               *****
#                               ** TESTS=1 PASS=1 FAIL=0 SKIP=0           1301.00         0.03      45972.82 **
#                               *****
#
```

Figure 2. Simulation results of the example testbench.

Because the Python testbench executes outside the simulator, there is no visibility to it from the graphical user interface. The RTL model and its waveforms can be inspected as in a simulation with a SystemVerilog testbench. Figure 3 shows the graphical user interface view of the simulation in Questa Sim.

Figure 3. Graphical user interface view of the example testbench simulation.

### 3 UVM IN PYTHON

Chapter 2 showed an example of a simple Python testbench without hierarchical structure. This chapter introduces the Python library *pyuvm* for building a UVM testbench in Python. Section 3.1 includes only a brief overview of UVM for context, but the references can be followed for further reading. Section 3.2 explains the key differences between SystemVerilog and Python that are relevant to the chapter.

Section 3.3 focuses on the main topic, UVM in Python. The features of *pyuvm* are compared with the full set of features in SystemVerilog UVM to give an idea of what is possible with its current release version. The structure and functionality of a Python UVM testbench are explained with code examples that are compared with SystemVerilog UVM code. Most of the examples are sanitized excerpts from the testbenches built for the performance testing part of this thesis work, introduced in Chapter 5. While the complete codebase cannot be disclosed, the examples aim to give insight to the implementation techniques of the testbenches.

#### 3.1 Universal Verification Methodology

UVM is a standardized methodology that defines a class library of testbench components, utilities and macros for functional verification [10 p. 12][18 p. 1]. The UVM classes contain a basic set of functionalities and can be extended to make components for specific use cases. Figure 4 shows an example of a UVM testbench architecture.

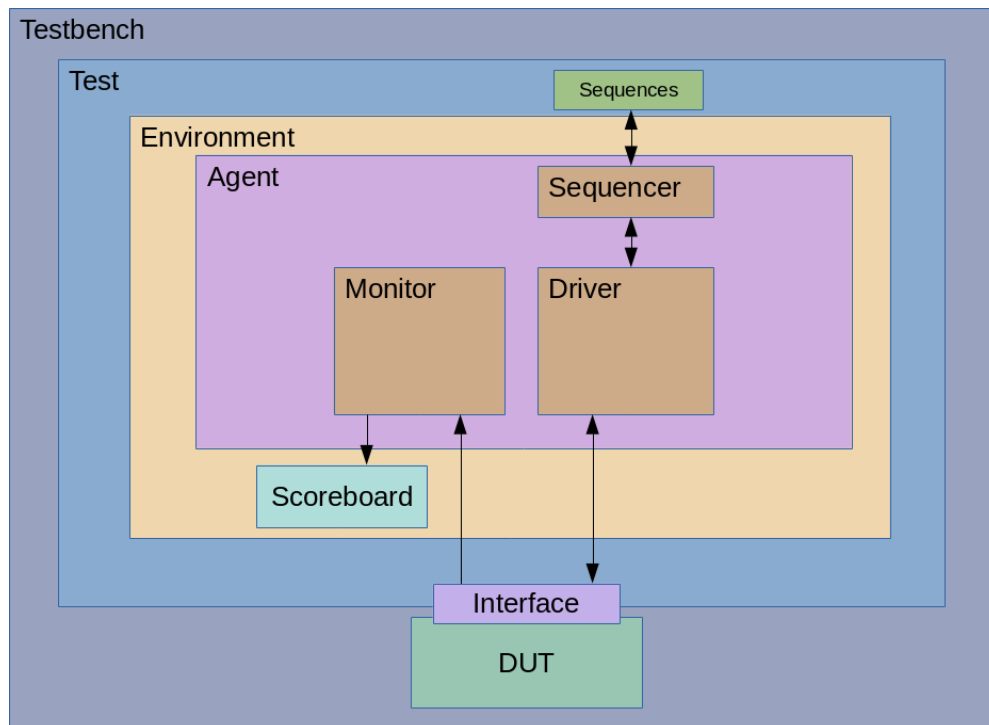


Figure 4. Example UVM testbench.

UVM components connect to each other with communication interfaces that operate at the transaction level. Transactions are class instances that encapsulate information of signal-level activity at higher levels of abstraction. For example, a transaction for reading data from a memory block in the design could contain a memory address, operation type indicating it is a read transaction, data field for the returned read data, and error field for a returned error

response in case the bus operation fails. Using transactions, testbench components can communicate at a level that hides implementation details of the communication protocol. This is called transaction-level modelling (TLM). [18 p. 7–8]

Sequences generate sequence-enabled transactions, which are also called sequence items in UVM. Sequences typically randomize data fields of the items according to a set of constraints that place rules for the randomization. Sequences may form a hierarchy where high-level sequences use lower-level sequences to perform more sophisticated operations, for example combining bus write sequences to create a sequence that configures the DUT to a certain state. [10 p. 18, 177]

A sequencer controls the flow of sequence items between sequences and a driver. The sequencer sends transactions contained by sequences to a driver and receives response transactions back. The sequencer then passes the response to the sequence that created the request transaction. [10 p. 194]

A driver receives new transactions from the sequencer. The driver breaks the transactions into individual data members and initiates a data transfer to the DUT via an interface. The driver then waits for a response from the interface, packages the response into a transaction and passes that transaction to the sequencer. [10 p. 175][18 p. 34]

A SystemVerilog interface is the only component in the example UVM architecture that directly connects to and communicates with the DUT. It is a non-UVM construct that contains nets or variables that connect to ports in the DUT. Typically, the interface implements some bus protocol with tasks to read, write and sample signal values, which UVM components driving and sampling signals call. [1 p. 748]

A monitor is a passive component that samples the interface signals. It packages the samples into transactions for analysis components such as a scoreboard. [10 p. 15][18 p. 38]

A scoreboard verifies correct functional operation of the design [18 p. 2, 70]. It predicts the design's response to stimulus and checks that the observed response of the DUT matches the prediction [10 p. 16]. The input samples and observed responses are received from the monitor through an agent. Based on the input samples, the scoreboard can either calculate the expected response or pass the samples to a predictor, or a reference model, that generates the predicted response. The predictions may also be pre-generated if the applied stimulus is non-random, in which case the reference data is read from a file.

An agent is a component that encapsulates a sequencer, a driver, and a monitor [10 p. 15]. Typically, there is an interface for each bus protocol in the design, and an agent for each interface. For example, a design that uses Serial Peripheral Interface (SPI) and Advanced High-Performance Bus (AHB) would have an SPI and AHB interface, and an SPI and AHB agent.

An environment is a container class of verification components that comprise the testbench [10 p. 15]. It provides default configurations to the components, and typically it can enable and disable components to satisfy different use cases. The environment can also be a component in another environment of a larger, subsystem or top level, testbench [18 p. 2].

A test is the topmost class in the UVM hierarchy. The test instantiates and configures the environment and creates and starts sequences to test some functionality of the DUT. [18 p. 2]

One way to pass configuration parameters in the testbench hierarchy is through a configuration database, which is a part of the UVM class library. A component creates configuration objects that contain the necessary information to configure other, typically hierarchically lower-level, components. An example of such information is a flag for enabling or disabling a driver in an agent. The component stores the objects in the database, and the components that the objects are assigned to obtain the configuration objects when the components are being built. [18 p. 58–59]

There are two ways to create UVM objects and components. The first is to call the class constructor, as is typical in object-oriented programming languages. The other way is to create the component using a UVM factory. Calling one of the factory's creation methods creates an instance of a class and returns it. The main benefit of using this method is that the type of class instances created through the factory can be overridden with a derived type of the class. The test, for example, can set an override in the factory such that whenever it creates a monitor instance of type A, it creates a monitor instance of type B derived from A instead. [18 p. 40–41]

A UVM test goes through certain pre-defined phases. Each component contains functions and tasks that define what they will execute in each phase. The three essential phases are the build phase, the connect phase and the run phase. In the build phase, testbench components are created and configured. In the connect phase, the components that pass transactions between each other are connected. The UVM standard defines classes and interfaces for TLM to enable sending and receiving transactions. These interfaces are connected in the connect phase. The run phase is where the simulation time consuming part of the test is executed: drivers driving stimulus and monitors sampling the signals. [10 p. 95, 162]

The result of the test will depend on the amount and the severity of reported errors during the test run. For example, if a testbench component fails to obtain its build configuration, the test may issue a fatal error with *uvm\_fatal* macro and end the test immediately. If the scoreboard notices a disparity in the predicted and the observed response, it may raise a normal error with *uvm\_error*. The macro *uvm\_warning* can be used for minor or potential issues, and *uvm\_info* for messages of simply informational nature. These report messages are tallied in the UVM's report server, and the message counts can be printed after the simulation to inform about the test results. [10 p. 54, 429]

### 3.2 Python and SystemVerilog

To help understand the implementation details of pyuvm, this section highlights some of the key differences between Python and SystemVerilog. There are slight differences in the terminology when discussing features of Python and SystemVerilog, so let us clarify them first.

SystemVerilog class consists of properties and methods. Properties are class's data, and methods are its subroutines – functions and tasks. Together they are called class members. An object is an instance of a class. [1 p. 170–171]

In Python, a class consists of attributes. Attributes can be divided into class attributes and instance attributes. Instance attributes belong to an instance of a class, or an instance object. Instance attributes comprise of data attributes – also called instance variables – and instance methods. The class attributes are class variables and class methods. They belong to a class object. Below is an example of this attribute division. [19][20]



```

class MyClass:
    string = "Hello world!" # class variable
    def __init__(self): # class constructor
        self.name = "John" # instance variable
    def func(self): # instance method
        print("foo")

inst_obj = MyClass() # creates an instance object and calls __init__
print(inst_obj.name) # referring to an instance variable
print(MyClass.string) # referring to a class variable
f = inst_obj.func # instance method object
inst_obj.func() # instance method call
f() # equivalent to the previous line

```

The previous example shows that even functions are objects in Python. Everything except most keywords are objects. A string of characters is an object of *str* class. *MyClass* is a class object of *type* class – a metaclass that creates other classes.

SystemVerilog is a statically typed language. Before execution, a compiler needs to compile SystemVerilog code. All variables have types that are determined in compile-time. The type is given when a variable is declared. Attempting to store a value of wrong type to the variable results in a compilation error. Using a method that does not exist in the variable also leads to a compilation error.

```

int a = 10;
a = '{20, 30}; // compilation error
a.quack(); // compilation error

```

Python, on the other hand, is a dynamically typed language. Python uses an interpreter which can run source files directly without a separate compilation phase [20]. All type checking is performed in runtime [21]. The type of value stored in a variable can change, because a variable in Python is only a reference to an object that holds the value. Python interpreter determines the object type implicitly, so a type is not needed in a variable declaration. Calling a non-existent method results in a runtime error.

```

a = 10 # a points to an object of int class
a = [20, 30] # now it points to an object of list class
a.quack() # runtime error

```

In the code above, the interpreter did not check *a*'s type to see if it implements *quack* method. It simply called the method and returned an exception because the method was not found. Let us look at the next example to see the implications of this. The example code defines classes *Duck* and *Crow* and their instance methods, creates a list with instance objects of both classes, and tries to call methods *fly* and *quack* of the objects.

```

class Duck:
    def fly(self):
        print("Duck flying")

    def quack(self):
        print("Quack!")

class Crow:
    def fly(self):
        print("Crow flying")

    def caw(self):
        print("Caw caw!")

for animal in [Duck(), Crow()]:
    animal.fly()
    animal.quack()

```

The output for this example is:

```

>>> Duck flying
>>> Quack!
>>> Crow flying
>>> AttributeError: 'Crow' object has no attribute 'quack'

```

The crow worked as a duck until it had to quack. This is an example of Python's duck typing. It is a programming style where the type of the objects does not matter as long as they have the used attributes [20]. For example, functions accept all types of objects as arguments. The objects only need to have the attributes that the functions will use. This contrasts with SystemVerilog, where type compatibility is maintained with explicit argument types, type parameters and type casting.

In both languages, it is possible to use a class as a base for a new class. The new class will have all the members, or attributes, of the base class, also known as super class, in addition to the ones declared in the new class. This is known as single inheritance. In Python, a class can also inherit from more than one base class, which is called multiple inheritance [19].

```

class A: pass
class B(A): pass
class C(A): pass
class D(B, C): pass

```

Class *B* and *C* inherit the attributes of Class *A* through single heritance, but Class *D* receives the attributes of all *A*, *B* and *C* through multiple inheritance. However, since *B* and *C* share a common base class *A*, it must be searched for a method last in a method call from a Class *D* instance so that any method overrides by *B* or *C* still hold. Python uses the C3 linearization algorithm to resolve the search order as *D*, *B*, *C* and finally *A* [22].

### 3.3 pyuvm

Pyuvm is an implementation of the Universal Verification Methodology in Python. The idea of pyuvm is to provide a pythonic version of UVM that has the main features of the methodology while taking advantage of Python's characteristics and built-in features [11]. A general overview of its current implementation status of the UVM standard IEEE 1800.2 is in Table 1.

Some features are yet to be added, like the register layer, and some are not planned to be included because they are already in some form in Python. The classes listed as implemented have at least their basic functionality but may have some properties and methods missing.

Table 1. Implementation status of pyuvm

IEEE 1800.2 section	Implemented in pyuvm	Not implemented	Comment
5. Base classes	uvm_void, uvm_object, uvm_transaction, uvm_port_base	uvm_time, uvm_field_op	Python magic methods like <code>__str__</code> are used instead of UVM field operations
6. Reporting classes	-	All	The reporting system is based on Python's logging library
7. Recording classes	-	All	Python does not run in a simulator
8. Factory classes	uvm_factory	Component and object wrappers	All classes extending <i>uvm_void</i> are automatically registered with the factory without a need for wrappers
9. Phasing	Common phases	Runtime phases Phase domains Phase jumping	
10. Synchronization classes	-	All	Python's threading library provides thread synchronization, but features are not identical
11. Container classes	-	All	
12. UVM TLM interfaces	Port and export classes	Implementation classes, TLM 2	Implementation ports are not needed with Python's duck typing and multiple inheritance
13. Predefined component classes	uvm_component, uvm_test, uvm_env, uvm_agent, uvm_monitor, uvm_scoreboard, uvm_driver, uvm_subscriber	uvm_push_driver	
14. Sequence classes	uvm_sequence_item uvm_sequence	uvm_sequence_library	

15. Sequencer classes	uvm_sequencer Sequence item ports	uvm_push_sequencer	
16. Policy classes	-	All	
17. Register layer	-	All	
18. Register model	-	All	
19. Register layer interaction with the design	-	All	
Annex B Macros and Defines	-	All	
Annex C Configuration and Resource classes	uvm_config_db	uvm_resource_db	Pyuvm's <i>ConfigDB</i> is a refactored version of uvm_config_db
Annex D Convenience classes, interface and methods	Factory interface Hierarchical reporting interface	uvm_callback_iter uvm_reg_block access Callback typedefs	
Annex E Test sequences	-	All	
Annex F Package scope functionality	uvm_root	Everything else	
Annex G Command line arguments	-	All	Python's <i>os</i> module can be used to retrieve environment variables

### 3.3.1 Class definition and object creation

Let us compare SystemVerilog-UVM (SV-UVM) and pyuvm testbench code to learn how the UVM implementations are different. Below is an example definition of a custom UVM driver in both implementations.

```
// SystemVerilog
import uvm_pkg::*;

class my_driver extends uvm_driver #(my_seq_item);
    `uvm_component_utils(my_driver)
    ...
endclass

# pyuvm
import pyuvm
from pyuvm import *

class MyDriver(uvm_driver):
    ...
```

SystemVerilog uses *extends* keyword to make *my\_driver* a derived class of *uvm\_driver*. *my\_seq\_item* is an argument to *uvm\_driver*'s parameter to inform about the type of sequence item the driver is handling. Many classes in SV-UVM need to know the size of data they will

operate on at compile-time. Type parameters provide this information to create classes for different types. The macro ``uvm_component_utils`` registers `my_driver` class type to the UVM factory, so that later the factory can be used to create objects of the custom driver.

In Python, the base class is inside parentheses following the name of the derived class. There is no type parameter passed to `uvm_driver`, since due to Python's duck typing, ports do not need information about the type of objects they are handling. The factory registration macro is not needed because all classes extending from `uvm_void` are automatically registered with the factory.

SV-UVM components and objects need to define a constructor method called `new` so that new objects of that class can be created. For `my_driver` class, it would look as follows.

```
function new(string name, uvm_component parent = null);
    super.new(name, parent);
endfunction
```

The constructor method simply calls the `new` method of the base class `uvm_driver`. In Python, the constructor method is `__init__`. However, it does not need to be defined for a constructor like this since the base implementation of `__init__` already calls the constructor of the base class with the arguments. `MyDriver` does not need `__init__` because its instance variables can be declared in the build phase. The constructor can be used for non-UVM classes and UVM objects that do not use UVM phasing, or if something needs to be initialized before the phases. For example, if we had a sequence item `MySeqItem` with instance variables `data` and `addr`, its `__init__` could be defined as below.

```
class MySeqItem(uvm_sequence_item):
    def __init__(self, name):
        super().__init__(name)
        self.addr = None
        self.data = None
```

In Python, instance methods always have `self` as the first parameter, which refers to the object itself, analogous to SystemVerilog's `this`. `None` is a special singleton object in Python, and it is now used to leave `addr` and `data` without an initial value.

Unlike in SystemVerilog, instance variables in Python must be declared within one of the class's instance methods. If we declare `self.addr` outside `__init__` or other instance method, an exception will be raised because `self` is not defined outside the methods. If we declare `addr` without `self` outside an instance method, `addr` becomes a class variable, belonging to a class object and not an instance object.

To create a `my_seq_item` object, we can either call `my_seq_item`'s constructor directly or use the factory method. In SV-UVM, it would look as follows.

```
my_seq_item tr;
// Direct method
tr = new("my_seq_item_tr");
// Factory method
tr = my_seq_item::type_id::create("my_seq_item_tr");
```

Let us look at the equivalent code in pyuvvm.

```
# Direct method
tr = MySeqItem("my_seq_item_tr")
# Factory method
tr = MySeqItem.create("my_seq_item_tr")
```

The class constructor *new* creates a class instance in SV-UVM. In Python, a class type followed by parentheses calls the class constructor `__init__` and returns an instance object. *MySeqItem*'s `__init__` has a name parameter, so we pass a name argument. The *self* parameter does not need an argument – it is passed implicitly.

For the factory method in SV-UVM, UVM components and objects registered with the factory have a proxy declared as *type\_id*. We call that proxy's *create* method to create a class instance. Each registered class type has its own proxy that creates objects of that type. [10 p. 69–70]

In pyuvvm, the factory has been refactored and it does not use proxies. Instead, *uvvm\_component* and *uvvm\_object* implement *create* method that calls for the factory to return an instance object of the caller class. When a class is defined in Python, it starts existing as a class object, and its metaclass defines how that object is constructed. Pyuvvm's factory metaclass extends Python's default *type* metaclass to store the class object to a dictionary when it is constructed. The factory uses the dictionary to create instance objects of the class. Since this metaclass is the metaclass of *uvvm\_void*, all classes extending from *uvvm\_void* are automatically added to the dictionary i.e., registered with the factory.

### 3.3.2 Phasing

Pyuvvm implements the common phases, from *build\_phase* to *final\_phase*, but runtime phases, such as *configure\_phase* and *reset\_phase*, are not implemented. Many of the phasing features are left out, such as phase domains, schedules, and phase jumping. The top-level component *uvvm\_root* simply processes through a list of common phases in order, traversing through the component hierarchy and calling the phase method of the current phase in each component.

Like in SV-UVM, *uvvm\_phase* is the base class for all the phase classes in pyuvvm, but its only function is to execute the phase method of a component. The classes *uvvm\_bottomup\_phase* and *uvvm\_topdown\_phase* are the base classes for phases that traverse the component tree bottom-up and top-down, respectively. The base class for runtime phases, *uvvm\_task\_phase*, is replaced by *uvvm\_threaded\_execute\_phase* with its original coroutine-based implementation of runtime phases.

Custom phases can be derived from these base classes, but there is no official support for it. To make pyuvvm execute the custom phases requires monkey patching i.e., runtime modification, of the phasing module to add the custom phase to the list of phases, and of *uvvm\_component* to add the phase to its instance methods. In addition, custom runtime phases do not operate properly as they are all launched at the same with *run\_phase*.

Below are *build\_phase* and *run\_phase* declarations and an objection call in both SV-UVM and pyuvvm.

```
// SystemVerilog
function void build_phase(uvm_phase phase);
task run_phase(uvm_phase phase);
    phase.raise_objection(this);
# Python
def build_phase(self):
    async def run_phase(self):
        self.raise_objection()
```

In pyuvm, phases do not take *uvm\_phase* as an argument because of the simplified implementation of the class. A typical use of the phase argument in SV-UVM is to call the phase's *raise\_objection* and *drop\_objection* methods in *run\_phase* to prevent its premature termination. These methods have been delegated to *uvm\_component* in pyuvm, so we can call *self.raise\_objection()* and *self.drop\_objection()* instead.

### 3.3.3 Configuration database

As mentioned in Section 3.1, a configuration database is used in UVM to share configuration information between testbench components. It has *set* method to store information and *get* method to retrieve information. For example, in SystemVerilog, to pass a configuration object from a test class down to its environment's agent, we could do as follows.

```
class my_test_base extends uvm_test;
    my_env env;
    ...
    function void build_phase(uvm_phase phase);
        my_agent_configuration cfg;
        env = my_env::type_id::create("env", this);
        cfg = my_agent_configuration::type_id::create("my_cfg");
        uvm_config_db #(my_agent_configuration)::set(
            this, "env.agt", "cfg", cfg);

class my_agent extends uvm_agent;
    my_agent_configuration cfg;
    ...
    function void build_phase(uvm_phase phase);
        if(!uvm_config_db #(my_agent_configuration)::get(
            this, "", "cfg", cfg)) begin
            `uvm_fatal(get_type_name(), "...")
        end
```

The configuration database is parameterized, so we need to pass the configuration object type to the parameter in *set* and *get* methods. The first argument in *set* is the context, which is now the test class object, and the second argument is the instance name. Together they create the full hierarchical path to the component the configuration object is passed onto. The third argument is a name for the value in the database, and the fourth argument is the actual value [10 p. 404]. When the agent retrieves the configuration object, the *get* is wrapped in an if-statement. If retrieving the object fails, the function call returns zero, which makes *`uvm\_fatal* raise a fatal error. If the error is left uncaught at this point, using the non-existent configuration object later in the execution will lead to a segmentation fault that is less obvious to debug.

In pyuvm, the configuration database functions similarly but the calls are simplified. Below is the equivalent code in pyuvm.

```

class MyTestBase(uvm_test):
    def build_phase(self):
        self.env = MyEnv("env", self)
        self.cfg = MyAgentConfiguration.create("my_cfg")
        ConfigDB().set(self, "env.agt", "cfg", self.cfg)

class MyAgent(uvm_agent):
    def build_phase(self):
        self.cfg = ConfigDB().get(self, "", "cfg")

```

The configuration database is named *ConfigDB* in pyuvvm. It is a singleton object, therefore the constructor call *ConfigDB()* is used to get the database object. Type parameters are not used. The arguments of *set* method are the same as in SystemVerilog but *get* has only three arguments because the call directly returns the configuration object, which is assigned here to the instance variable *self.cfg*. The *get* is not wrapped in an if-statement like in SystemVerilog. As explained in Section 2.3, Python exceptions determine the simulation result in cocotb. The same is true for pyuvvm. Exceptions have the same result as *uvvm\_fatal*: they fail the test and end the test execution. If *get* cannot find the configuration object in the database, it raises *UVMConfigItemNotFound* error with a message informing about the missing object, prints a traceback pointing to the *get* call, and ends the simulation.

### 3.3.4 Interfacing with the DUT

In SV-UVM, to write and read bus signal values of the DUT, a SystemVerilog interface – “physical” interface – is typically connected to the DUT. UVM components do not directly use the physical interface because hierarchical references to signals would limit the reusability of the testbench [23]. Instead, the components use a virtual interface as a handle to the real interface. Through the virtual interface handle, they can assign or read signals of the physical interface and call the interface methods [23].

Let us go through an example procedure to pass a virtual interface to a driver. The top testbench module assigns the physical interface to a virtual interface variable in the configuration database as shown below.

```

module top();
    ...
    my_interface my_if(); // the physical interface
    initial begin
        uvm_config_db #(virtual my_interface)::set(
            null, "uvm_test_top", "my_vif", my_if);
    end

```

The test class creates a configuration object *cfg*, gets the virtual interface from the database and stores the interface to a virtual interface variable in *cfg*. *set* method to store *cfg* in the database and to set it accessible to the agent is omitted as this was shown in Subsection 3.3.3.

```

// In test class
my_agent_configuration cfg;
if (!uvm_config_db #(virtual my_interface)::get(
    this, "", "my_vif", cfg.my_vif))
    `uvvm_fatal("...")

```

The agent retrieves the configuration object like previously in Subsection 3.3.3 and stores the virtual interface handle to the database for the driver.



```
// In agent class
uvm_config_db#(virtual my_interface)::set(
    this, "drv", "my_vif", cfg.my_vif);
```

Finally, the driver obtains the virtual interface handle from the configuration database and assigns it to its property *my\_vif*. Through *my\_vif*, the driver has access to the members of the physical interface.

```
// In driver class
class my_driver extends uvm_driver #(my_seq_item);
    virtual my_interface my_vif;
    ...
    if (!uvm_config_db#(virtual my_interface)::get(
        this, "", "my_vif", my_vif))
        `uvm_fatal(...);
```

There is no strict equivalent to a SystemVerilog interface in pyuvvm or cocotb. Section 2.3 showed that cocotb uses *dut* object to access signals in its hierarchy. We could build an interface-like class that directly references *dut* to read and assign values.

```
class MyInterface:
    def __init__(self):
        self.dut = cocotb.top
        ...
    async def reset(self):
        self.dut.aresetn.setimmediatevalue(0)
        ...
```

However, this interface has a reusability problem. Say we are building a verification intellectual property (VIP) for AHB-Lite protocol. AHB-Lite, according to its specification, uses a clock signal named *HCLK* to synchronize the bus. In the DUT, there is a port *DomClk* that needs to connect to *HCLK*. If the top module, *cocotb.top*, is the DUT, *MyInterface* will only find *DomClk* – not *HCLK*. We must find a way to read *DomClk* through *HCLK* variable so that the VIP is design-agnostic.

We could create a SystemVerilog interface whose signal names match with the variable names that *MyInterface* uses, instantiate the interface along with the DUT in a top testbench module, and connect the interface to the DUT. *MyInterface* would then point to the SystemVerilog interface instead of the top module, *cocotb.top*.

Cocotb's external bus package *cocotb-bus* solves the problem in Python using dynamic attributes [23]. The advantage of this approach is that the signal names do not have to match between the HDL entity and the Python interface. The following example shows this solution in a simple version that uses the configuration database.

The VIP will use a configuration object to configure the interface. This object contains a dictionary that maps AHB-Lite signal names to custom names. By default, the standard and custom names are the same. The object also has a handle to the entity containing the bus signals. This could be the top testbench module, an interface or a modport. Here the default is the top module, *cocotb.top*. Lastly, it has a placeholder variable for the interface handle.

```

class AHBLiteConfiguration(uvm_object):

    def __init__(self, name="ahb_lite_cfg"):
        super().__init__(name)
        self.ahb_signals = {"HCLK": "HCLK", "HRESETn": "HRESETn",
                            ...
        }
        self.bus_entity = cocotb.top
        self.ahb_if = None

```

The base test class creates the configuration object and maps the signal names on the HDL side to the default names in the signal dictionary. The test also instantiates the interface, passing the bus entity and the signal dictionary to its constructor, and assigns the handle to *ahb\_if* in the configuration object. Finally, it stores the object in the configuration database.

```

class MyTestBase(uvm_test):

    def build_phase(self):
        self.cfg = AHBLiteConfiguration.create("ahb_lite_cfg")
        self.cfg.ahb_signals["HCLK"] = "DomClk"
        ...
        self.cfg.ahb_if = AHBLiteInterface(
            self.cfg.bus_entity, self.cfg.ahb_signals)
        ConfigDB().set(self, "env.ahb_lite_agent", "cfg", self.cfg)

```

The agent sets the configuration object for the driver as was shown in Subsection 3.3.3. The driver retrieves the configuration object and assigns the interface handle to its instance variable.

```

class AHBLiteDriver(uvm_driver):

    def build_phase(self):
        ...
        self.cfg = ConfigDB().get(self, "", "cfg")
        self.ahb_if = self.cfg.ahb_if

```

The interface class assigns the HDL signal names to the generic names using the Python method *getattr(object, name)*. The method searches a given object for an attribute with a given name and returns a handle to the attribute. In this case, *getattr* searches *bus\_entity*, which is equal to *cocotb.top*, for an attribute with the name *DomClk*, stored at the dictionary's key *HCLK*, and returns a handle to it. The handle is assigned to the instance variable *HCLK*. Now we can read and drive *DomClk* with *self.HCLK* without hardcoding hierarchical signal paths or names in the interface.

```

class AHBLiteInterface:

    def __init__(self, bus_entity, ahb_signals):
        self.HCLK = getattr(bus_entity, ahb_signals["HCLK"])
        ...
    async def monitor_write(self, ...):
        ...
        await RisingEdge(self.HCLK)
        ...

```

### 3.3.5 Transaction-level modelling

To deliver transactions from one component to another, UVM TLM 1.0 includes three types of ports: ports, exports and impls – short for implementation ports. Ports initiate transaction requests. Exports forward the request to its implementation. Impls implement the used TLM method to execute the transfer. TLM methods include, among others, *put* to send transactions, *get* to receive transactions, and *peek* to obtain transactions without consuming it. The three methods have a blocking version which blocks the thread until the method succeeds, and a nonblocking version which tries to execute the method but may fail if the target component is not ready to do so. An example port connection is shown in Figure 5. [10 p. 120]

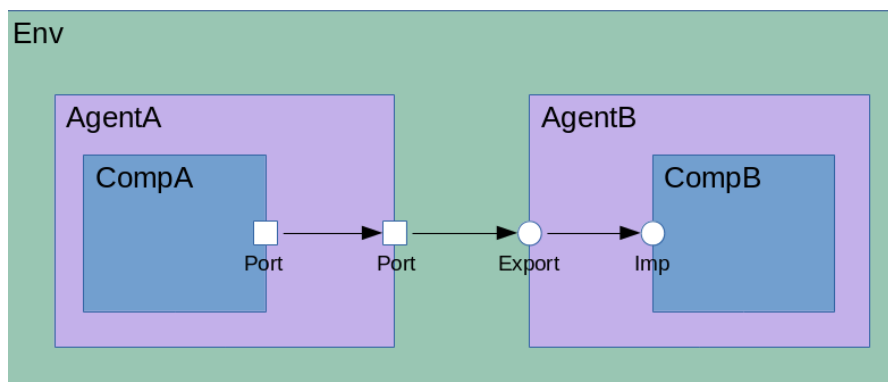


Figure 5. An example port connection in SV-UVM.

For example, *CompA* is sending a transaction to *CompB*. Let us say the transfer needs to block the thread's execution until the transfer is finished, so *CompA* has a *uvm\_blocking\_put\_port* and it calls the port's *put* method to request a transfer.

```

class CompA extends uvm_component;
    uvm_blocking_put_port #(my_seq_item) m_put_port;
    ...
    function void build_phase(uvm_phase phase);
        m_put_port = new("m_put_port", this);
        ...
    task run_phase(uvm_phase phase);
        ...
        m_put_port.put(item);
    endtask
endclass

```

Ports are connected in *connect\_phase*. Below is an example of *Env* connecting *AgentA*'s port to *AgentB*'s export.

```

class Env extends uvm_env;
    ...
    function void connect_phase(uvm_phase phase);
        agent_a.put_port.connect(agent_b.put_export);
        ...
    endfunction
endclass

```

Through the connected ports and exports, the request reaches *CompB*'s *imp*, which implements the *put* method. When *CompA* calls the *put* method, the *put* task in *CompB* gets called.

```

class CompB extends uvm_component;
  uvm_blocking_put_imp #(int, ed_monitor) put_imp;
  ...
  function void build_phase(uvm_phase phase);
    put_imp = new("put_imp", this);
    ...
  task put(my_seq_item item);
    `uvm_info("Trace",
      $sformatf("Received item:addr:%0h, data:%0h",
        item.addr, item.data), UVM_HIGH)
    foo(item.addr, item.data);
  endtask

```

An example port connection in pyuvvm is pictured in Figure 6. Pyuvvm only has two of the three main types of ports: ports and exports. It does not have imps because duck typing and multiple inheritance renders them unnecessary. Ports either initiate transfers or forward them towards an export. An export implements the TLM method to execute the transfer.

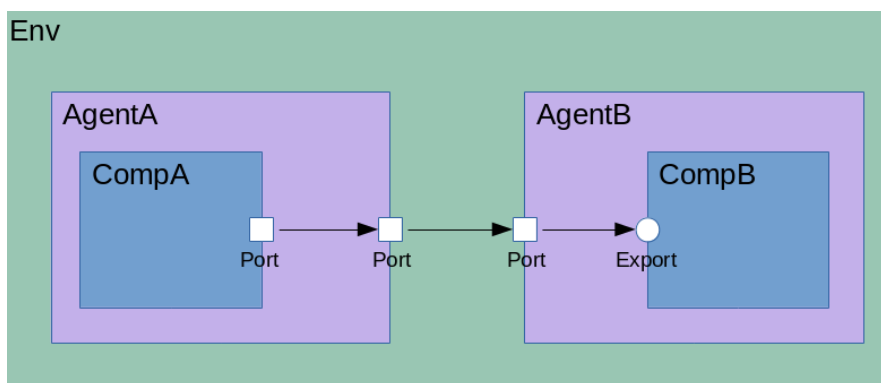


Figure 6. An example port connection in pyuvvm.

Let us go through the same example as above. Pyuvvm also has the *uvm\_blocking\_put\_port* class, but no type parameterization is needed. *put* is awaited because it is a blocking call.

```

class CompA(uvm_component):
  def build_phase(self):
    self.put_port = uvm_blocking_put_port("put_port", self)
  async def run_phase(self):
    ...
    await self.put_port.put(item)

```

Port-to-port connections have a similar syntax to SV-UVM. Here *Env* has declared *AgentA* and *AgentB* objects and now connects their ports.

```

class Env(uvm_env):
  ...
  def connect_phase(self):
    self.agent_a.put_port.connect(self.agent_b.put_port)

```

Pyuvvm expects *uvm\_\*\_export* classes to implement the needed TLM methods for the connected port. Therefore, *CompB* can inherit an export class and then implement the methods. In a simple case where *CompB* does not need to inherit from classes other than *uvm\_component*,

the export class can be *CompB*'s only base class, because all *uvm\_\*\_export* classes are originally derived from *uvm\_component*.

```
class CompB(uvm_blocking_put_export):
    ...
    async def put(self, item):
        ...
```

In a case where *CompB* has a custom base class to inherit from, called *BaseCompB* for example, Python's multiple inheritance can be utilized.

```
class CompB(BaseCompB, uvm_blocking_put_export):
    ...
    async def put(self, item):
        ...
```

*AgentB*'s put port is connected directly to *CompB* object itself.

```
class AgentB(uvm_agent):
    def build_phase(self):
        self.comp_b = CompB("comp_b", self)
        self.put_port = uvm_blocking_put_port("put_port", self)
        ...
    def connect_phase(self):
        self.put_port.connect(self.comp_b)
```

UVM TLM also includes first-in, first-out (FIFO) classes. FIFO classes provide a transaction buffer between two connected components. The classes have exports and implementations for the TLM methods *put*, *get* and *peek*. [10 p. 130–133]

Below is an example of using FIFO's nonblocking *get* method to retrieve a transaction from the FIFO if there is one available. The main difference of FIFOs between pyuvm and SV-UVM is that they are not type parameterized in pyuvm. The example also highlights the difference in the nonblocking *try\_get* methods. Python does not allow assigning a function's return value into its argument like SystemVerilog does with *output* keyword in function or task parameters. In SV-UVM, *try\_get* returns the transaction in the function argument and a success bit as the function's return value. In pyuvm, *try\_get* returns a tuple with the success bit as the first value and the transaction as the second.

```
// SystemVerilog
class my_predictor extends uvm_component;
    uvm_tlm_analysis_fifo #(ahb_lite_seq_item) ahb_fifo;
    ...
    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        ahb_fifo = new("ahb_fifo", this);
        ...
    endfunction
    task run();
        ahb_lite_seq_item ahb_tr;
        if(ahb_fifo.try_get(ahb_tr)) begin
            ...
```

```
# Python
class MyPredictor(uvm_component):
    def build_phase(self):
        self.ahb_fifo = uvm_tlm_analysis_fifo("ahb_fifo", self)
        ...
    async def run(self):
        success, ahb_tr = self.ahb_fifo.try_get()
        if success:
            ...
```

### 3.3.6 Reporting system

Pyuvvm relies on cocotb's logging system to substitute for UVM's reporting system. Each UVM component extending from *uvm\_report\_object* has its own logger object that can print messages. The severity of the messages can range from trace messages to critical errors. In SV-UVM, messages have both severity and verbosity levels, and messages can be omitted depending on the verbosity. In pyuvvm, messages do not have a verbosity level but messages below a certain severity can be filtered. Below is an example of the pyuvvm logging system in contrast to SV-UVM's reporting system.

```
// SV-UVM:
// `uvm_info has message tag, message and verbosity parameters
`uvm_info("TRACE", $sformatf(
    "AHB read: addr: 0x%0h, data: 0x%0h", rd_tr.addr, rd_tr.data),
    UVM_HIGH);
// Verbosity controls which messages are ignored instead of severity
// Set verbosity level for one component
set_report_verbosity_level(UVM_FULL);
// Set verbosity level for the component hierarchy
set_report_verbosity_level_hier(UVM_LOW);

# pyuvvm:
# Display INFO level message formatted in
# a similar style to SystemVerilog-UVM
self.logger.info(
    f"AHB read: addr: {hex(rd_tr.addr)}, data: {hex(rd_tr.data)}")
# Messages below level DEBUG are not displayed for this component
self.set_logging_level(DEBUG)
# Set logging level for this component and
# all components below it in hierarchy
self.set_logging_level_hier(ERROR)
# Set default logging level for all UVM components in the testbench
uvm_report_object.set_default_logging_level(CRITICAL)
```

In its current state, the logging system is lacking major features of the UVM standard. Log messages do not have functionality beyond displaying information, and the functionality cannot be easily changed as *set\_\*\_action* methods do in SV-UVM. The method *logger.error()*, for example, does not affect the test result like *`uvm\_error* does by default. Macros like *`uvm\_error* do not exist in pyuvvm which is an issue for UVM objects and interfaces that exist outside the UVM hierarchy. There is also no equivalent of *uvm\_report\_server* that can be queried for message counts.

One way to print messages in the UVM format from UVM objects and interfaces is to use UVM root object's logger. Another way is to create a *uvm\_report\_object* type object and use

its logger. The second method allows to add a name tag to the messages but comes at the cost of creating the report object.

```
class MyInterface:
    def __init__(self, ...):
        # Option #1
        self.logger = uvm_root().logger
        # Option #2
        self.logger = uvm_report_object("my_interface").logger
```

For the lack of *uvm\_error* and *uvm\_report\_server*, a custom handler class was used as a temporary solution in the testbench of this work. In Python, handlers are objects that process messages of the loggers they are attached to. The function of the custom handler, *ReportHandler*, was to keep count of logging messages in the testbench. The main part of the code is below. It uses *Singleton* metaclass from *pyuvm* to ensure that all loggers share a common *ReportHandler* object. The object collects log message counts of different log levels to a dictionary, which can be used in a test's report phase to print the message counts and to fail the test if there are errors.

```
import logging
from pyuvm.utility_classes import Singleton

class ReportHandler(logging.Handler, metaclass=Singleton):
    def __init__(self):
        super().__init__()
        self.log_counts = {
            "DEBUG": 0,
            "INFO": 0,
            "WARNING": 0,
            "ERROR": 0,
            "CRITICAL": 0,
        }
    def emit(self, record):
        if record.levelname not in self.log_counts:
            self.log_counts[record.levelname] = 1
        else:
            self.log_counts[record.levelname] += 1
```

Besides the added dependency, the main weakness of this solution is that the handler needs to be added to every logger in the testbench. The shortest method to do this is to use *uvm\_component*'s function *add\_logging\_handler\_hier* from the topmost test class to add the handler to every logger in the UVM hierarchy. However, this function can only be called after the hierarchy is built in the build phase, so any messages prior to that are not counted. The alternative is to add the handler in every component individually. The handler also needs to be added to the logger of *uvm\_root* and to any loggers outside the UVM component hierarchy if used.

### 3.3.7 UVM Register Abstraction Layer

One major feature of UVM not introduced in Section 3.1 is the UVM register layer, also known as the UVM Register Abstraction Layer (RAL). It is used to create an object-oriented model of the registers and memories in the DUT within the testbench. The model allows to execute read and write operations on the DUT registers and memories on a more abstract level

than transactions, and it improves vertical reuse of testbenches. With the register model, registers are addressed in a verification environment by name and not by physical address, so addresses can change without requiring changes in the environment and tests. UVM class library also defines a set of test sequences for the register model to verify register functionality of the DUT. [18 p. 75–78]

UVM RAL is not yet available in pyuvm. Only rudimentary implementations of register model classes *uvm\_reg\_block*, *uvm\_reg\_map*, *uvm\_reg* and *uvm\_reg\_field* are included in the release version. The classes can be used to build a basic register model, but they do not define methods to map the model to the registers of the DUT, nor do they provide frontdoor or backdoor access to the registers.

### 3.3.8 Defining test cases

In Section 2.3, *@cocotb.test()* decorator was used to mark a test coroutine for cocotb to simulate. For pyuvm tests, *@pyuvm.test()* is used for the same purpose. The test decorator accepts arguments to modify the execution, such as the timeout limit in the example below. The decorator also makes the call to *run\_test* method of *uvm\_root* automatic. The *run\_test* method is explicitly called in SV-UVM from the top testbench module to execute a UVM test.

```
@pyuvm.test(timeout_time=6000, timeout_unit="us")
class MySimpleTest(MyTestBase):
```

Multiple tests can be defined in the same module. The environment variable *MODULE* is used to specify the module or modules where the tests are located. Unlike in SV-UVM, all tests within the specified modules are run sequentially in a single simulation. Any singleton objects, including the configuration database and the factory, are cleared between the tests.

SV-UVM uses *+UVM\_TESTNAME* command-line parameter to select the UVM test to run from the compiled test classes. Cocotb has an environment variable *TESTCASE* to select specific test cases from the modules, but it is not compatible with the current version of pyuvm's test decorator. One alternative way is to place the tests in separate modules and select them with the environment variable *MODULE*. Another way is to define the test class without the pyuvm test decorator and to instead define an additional test coroutine with *@cocotb.test* decorator. The coroutine will call *uvm\_root* to start the test using the specified test class. This coroutine can be used with *TESTCASE*.

```
@cocotb.test(timeout_time=6000, timeout_unit="us")
async def my_simple_test(_):
    await uvm_root().run_test(MySimpleTest)
```



## 4 TESTBENCH SUPPLEMENTATION

While UVM provides a class library of testbench components and objects, macros and other utilities for functional verification, many critical verification features are part of the SystemVerilog language itself. This chapter focuses on two of such features, functional coverage and constrained random verification, and explores two Python packages that add the features in a Python testbench. One of the packages, *PyVSC*, provides a solution primarily in Python, while the other, *pyquesta*, provides a bridge to pass objects between Python and SystemVerilog, allowing to randomize objects and collect coverage with SystemVerilog.

### 4.1 Functional coverage

The goal in functional verification is to validate a design, and this requires comprehensive tests. To follow the progress towards this goal and to minimize redundancy in verification, coverage is used as a metric to track which portions of the design have been exercised and which are still untested. Some types of coverage can be automatically collected, such as measuring lines of code that have been executed in the HDL model. Functional coverage is a manually defined metric to measure the percentage of exercised features or capabilities of the design as specified by the design specification and enumerated in the test plan [1 p. 553].

The test plan can include UVM tests where a sequence or a set of sequences execute a feature in the RTL model. Another type of items to include in the test plan are coverage groups. *covergroup* is a SystemVerilog construct that encapsulates the data and temporal requirements of input stimuli and output response to cover a feature [25]. During a simulation, *covergroup* receives samples of input or output data which are compared with its coverage points. Coverage points include bins which are associated with sampled values or value transitions. If the sample “hits” a bin i.e., it has a value or completes a value transition associated with a bin, the bin is covered. [1 p. 554–558]

In the following example, a UVM component *my\_reg\_cov\_collector* has a coverage group with two coverage points. *cov\_addr* has a bin for each configuration register’s address in the DUT, and *cov\_op* has bins for read and write operations. There is also a cross coverage *cov\_addr\_x\_op* of the two coverage points, which has a bin for each combination of address and operation bin. The coverage collector is connected to a monitor’s analysis port. The monitor reads the interface bus signals, packages the values to a sequence item and sends the item to the analysis port subscribers. *my\_reg\_cov\_collector* samples the item, and if the item’s *addr* or *op\_type* value is found in a bin, the bin is covered.

```

class my_reg_cov_collector extends uvm_subscriber #(my_seq_item);
  `uvm_component_utils(my_reg_cov_collector)

  my_seq_item my_tr;

  covergroup my_reg_cg;
    cov_addr: coverpoint my_tr.addr {
      bins my_reg_mask = { 32'h1000 };
      bins my_reg_type = { 32'h1004 };
      //...

    cov_op: coverpoint my_tr.op_type {
      bins read = { 0 };
      bins write = { 1 };

    cov_addr_x_op: cross cov_addr, cov_op;
  endgroup

  function void write(my_seq_item t);
    my_tr = t;
    my_reg_cg.sample();
  endfunction

```

Functional coverage poses two problems for Python testbenches. Coverage constructs like *covergroup* are not part of the Python language, and they are outside the scope of pyuvvm and cocotb. The other issue is simulator integration. With SystemVerilog, simulators can collect coverage information from coverage groups and present the accumulated results at the end of a simulation. The coverage results can be saved to a database format and merged with results from other simulations to keep track of a project's verification progress. A functional coverage solution for Python must also be able to save the results in a format that electronic design automation (EDA) tools support.

There are currently two Python packages that add functional coverage to Python testbenches: *PyVSC* and *cocotb-coverage* [26][27]. PyVSC was chosen for the thesis because of its active development, syntactic similarity with SystemVerilog, and the ability to save results in Unified Coverage Interoperability Standard (UCIS) database and interchange formats, which simulators can read.

PyVSC provides coverage collection and constrained random verification features in Python. This section will focus on the former, while the next one focuses on the latter.

Let us repeat the example above using PyVSC. The decorator `@vsc.covergroup` marks *my\_reg\_cov\_collector* class as a coverage group. The constructor calls *with\_sample* – one of the methods the decorator adds to the class – and passes a dictionary of the attributes to cover from samples. The method accepts either a dictionary or keyword arguments.

```

import vsc
from my_common import my_regs

@vsc.covergroup
class my_reg_cov_collector(uvm_subscriber):
    def __init__(self, name, parent):
        super().__init__(name, parent)
        self.with_sample(
            dict(
                addr=vsc.bit_t(32),
                op_type=vsc.bit_t(1)
            )
        )

```

Continuing the constructor definition, we next declare the two coverage points and their coverage cross. The bins are specified in a dictionary passed to the parameter *bins* of *vsc.coverpoint*. One of the available bin types is *vsc.bin*. It can cover a value or a value range. The address bins declarations are made simpler by having the register name-address pairs stored in a dictionary *my\_regs* and using Python's dictionary comprehension to create the bins in one line.

```

        self.cov_addr = vsc.coverpoint(
            self.addr, bins={
                k: vsc.bin(v) for (k, v) in my_regs.items()
            }
        )
        self.cov_op = vsc.coverpoint(
            self.op_type,
            bins={
                "read": vsc.bin(0),
                "write": vsc.bin(1)
            }
        )
        self.cov_addr_x_op = vsc.cross([self.cov_addr, self.cov_op])

```

Aside from the constructor, *my\_reg\_cov\_collector* needs to define *write* function where *sample()* is called so that bus transactions sent to its analysis export are sampled.

```

def write(self, tr):
    self.sample(tr.addr, tr.op_type)

```

There are a few different options for viewing the results in PyVSC. One is *report\_coverage()* which writes the results in a stream. The function *write\_coverage\_db()* saves the results in either UCIS XML interchange format or a coverage database format using the given library that implements UCIS C API [28]. These functions can be called in, for example, a UVM test's report phase to report and save the coverage for the test.

```

def report_phase(self):
    vsc.report_coverage(details=False)
    vsc.write_coverage_db("cov.xml")
    vsc.write_coverage_db(
        "cov.ucdb", fmt="libucis", libucis="libucis.so")

```

## 4.2 Constrained random verification

As design complexity grows, it becomes increasingly infeasible to manually generate sufficient stimuli to verify the functionality of the design. Constrained random verification is a methodology that uses random stimulus to find bugs in the design. Constraints limit the range of random values to a set of legal values or values that test a specific feature. The testbench predicts the response to the random input stimulus using a reference model or other techniques and compares the prediction against the observed response. Functional coverage measures how effectively the random stimulus explores the space of possible inputs. [29]

SystemVerilog has constrained random stimulus generation built into the language. In a Python testbench, for some simple cases, Python's pseudo-random number generation in its standard library *random* can be sufficient. The testbench in this work used PyVSC's constrained random verification features as one option for random stimulus generation. PyVSC uses satisfiability modulo theories (SMT) solver Boolector to solve constraints and adds classes and methods similar to SystemVerilog constructs and patterns to control the stimulus generation.

Let us examine a short example in both languages for comparison. A sequence class *my\_read\_seq* defines a sequence for a read operation from an address *addr* of *size* bytes. The variable *reg\_idx* is for selecting an address from an array of register addresses. In SystemVerilog, random variables in a class are declared with an additional *rand* or *randc* keyword before the data type. The keyword *constraint* is used to place constraints on the randomization. Here *inside* keyword is used to limit *size* to values one, two and four.

```
class my_read_seq extends uvm_sequence #(my_seq_item);
    rand logic[31:0] addr;
    rand int size;
    rand int reg_idx;
    constraint allowed_size { size inside {1, 2, 4}; }
```

New values for the random variables in an object are selected with *randomize()* method. Additional constraints can be added if the method is followed by *with* keyword. The two constraints here make the address to be selected from *cfg\_regs*, which holds an array of configuration register addresses.

```
task cfg_reg_read();
    my_read_seq rd_seq;
    rd_seq = my_read_seq::type_id::create("rd_seq");
    rd_seq.randomize() with {
        reg_idx inside {[0:$size(cfg_regs)-1]};
        addr == cfg_regs[reg_idx];
    };
endtask
```

In PyVSC, randomized classes are marked with *@vsc.randobj* decorator. Random attributes are declared as objects of one of the random data types in PyVSC. In this example, we use *rand\_bit\_t* for an arbitrary-width unsigned data type and *rand\_uint8\_t* for an 8-bit unsigned integer. Constraint statements are defined in a method decorated with *@vsc.constraint*.

```

@vsc.randobj
class MyReadSeq(uvm_sequence):
    def __init__(self, name):
        super().__init__(name)
        self.addr = vsc.rand_bit_t(32)
        self.size = vsc.rand_uint8_t()
        self.reg_idx = vsc.rand_uint8_t()

    @vsc.constraint
    def allowed_size(self):
        self.size.inside(vsc.rangelist(1, 2, 4))

```

Randomization with additional constraints takes advantage of Python's context manager protocol. The keyword *with* wraps the constraint statements with methods that prepare the object for randomization and then randomize it with the added constraints. The keyword *as* assigns the name *it* to the prepared *rd\_seq* object.

```

async def cfg_reg_read(self):
    rd_seq = MyReadSeq.create("rd_seq")
    with rd_seq.randomize_with() as it:
        it.reg_idx in vsc.rangelist(vsc.rng(0, len(cfg_regs)-1))
        it.addr == cfg_regs[reg_idx]

```

### 4.3 Passing objects between Python and SystemVerilog

As functional verification in Python is still at an early stage, there can arise situations where it would be beneficial to be able to borrow SystemVerilog's features in a Python testbench. For example, in functional coverage and constrained random verification, PyVSC has not yet reached feature parity with SystemVerilog.

For users of Questa Sim, there is an option to exchange objects with SystemVerilog from a Python testbench using the Python package pyquesta. It is a collection of resources being developed for Python programmers using Questa [30]. Currently, it only contains SVConduit, which is the object exchange resource. In short, SVConduit enables one-way or two-way transfer of an object of a defined type from Python testbench to a SystemVerilog module via the SystemVerilog direct programming interface (DPI). DPI is an interface that allows inter-language calls between SystemVerilog and a foreign programming language [1 p. 938]. SVConduit needs the user to define the exchange object type and the behaviour associated with the object in the SystemVerilog module [30]. Building the rest of the exchange framework is automated with a script included in the package.

The following example shows how to use SVConduit to solve constraints of a sequence item in SystemVerilog with Questa Sim and send the item to a Python testbench.

The class of the exchange object is defined in YAML format. It needs to include a class name and properties along with their data types.

```

MyReadSeqSVCItem:
  addr:
    uint
  size:
    uchar
  reg_idx:
    uchar

```

A script packaged with SVConduit creates a Python module and a SystemVerilog package that define a Python and a SystemVerilog class based on the YAML file. The classes include pre-defined serialization and deserialization methods for transferring data between them. Below is the skeleton of the created SystemVerilog class.

```
class MyReadSeqSVCItem;
  rand int unsigned   addr;
  rand byte unsigned  size;
  rand byte unsigned  reg_idx;
  function new(...); // Initialize members from data buffer
  function string serialize(); // Serialize members
endclass
```

The SystemVerilog package also defines functions *sv\_put()* for sending objects from Python to SystemVerilog, *sv\_get()* for the reverse, and *sv\_transport()* for bidirectional transfer. These functions are not pre-defined since their implementation depends on the use case. We only need *sv\_get()* for this example, so the other two are defined as dummy functions.

```
function string sv_get();
  MyReadSeqSVCItem obj;
  string obj_str;
  obj = new();
  void'(obj.randomize() with {
    size inside {1, 2, 4};
    reg_idx inside {[0:$size(cfg_regs)-1]};
    addr == cfg_regs[reg_idx];
  });
  obj_str = obj.serialize();
  return obj_str;
endfunction
```

The function *sv\_get* selects random values for the object properties based on the constraints, serializes the properties in a string-type buffer and returns the buffer. SVConduit implements a C library which uses the C layer of DPI. A *get* function in the C library calls *sv\_get* and receives the returned buffer from the SystemVerilog module. The C function, in turn, is called from a Python implementation of *get*, which populates the attributes of a Python class instance of a given type with the buffer data and returns the object.

Now we can import SVConduit and the Python class to the testbench code and call *get* to receive a randomized *MyReadSeqSVCItem* object from SystemVerilog.

```
from pyquesta import SVConduit
from MyReadSeqSVCItemMod import MyReadSeqSVCItem

async def cfg_reg_read_svc(self):
  rd_seq = MyReadSeq.create("rd_seq")
  rd_seq_svc_item = SVConduit.get(MyReadSeqSVCItem)
  rd_seq.addr = cfg_regs[rd_seq_svc_item.reg_idx]
  rd_seq.size = rd_seq_svc_item.size
```

## 5 TEST SETUP

To test the simulation and verification performance of a Python UVM testbench, the same testbench and test cases were developed in both SystemVerilog and Python. Functional equivalence between the testbenches was maintained to the extent that was possible so that their performance could be compared. The following sections give an overview of the DUT and the testbenches.

### 5.1 The device under test

A small-scale IP with simple functionality was selected as the DUT to ease the effort of developing two testbenches. The DUT was an AHB-Lite slave, and it had registers that could be accessed via the bus. Some of the input and output ports were outside the control of the bus and were driven by a separate interface in the testbench. The relevant parts of the IP are shown below in Figure 7.

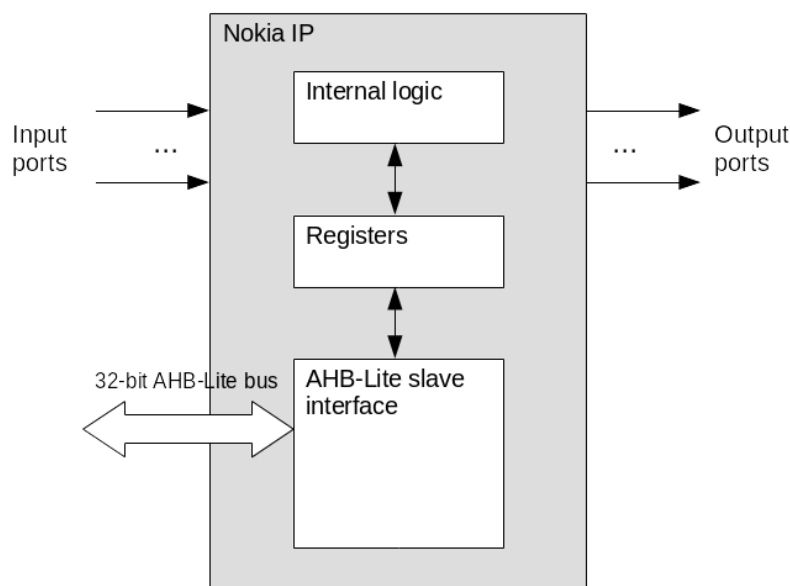


Figure 7. The IP used as the DUT for the performance tests.

### 5.2 The testbenches

The general testbench architecture for both the SystemVerilog and the Python UVM testbench is in Figure 8, with the slight distinction that interfaces belonged to the base test class in the Python version and to the top testbench module in the SystemVerilog version.

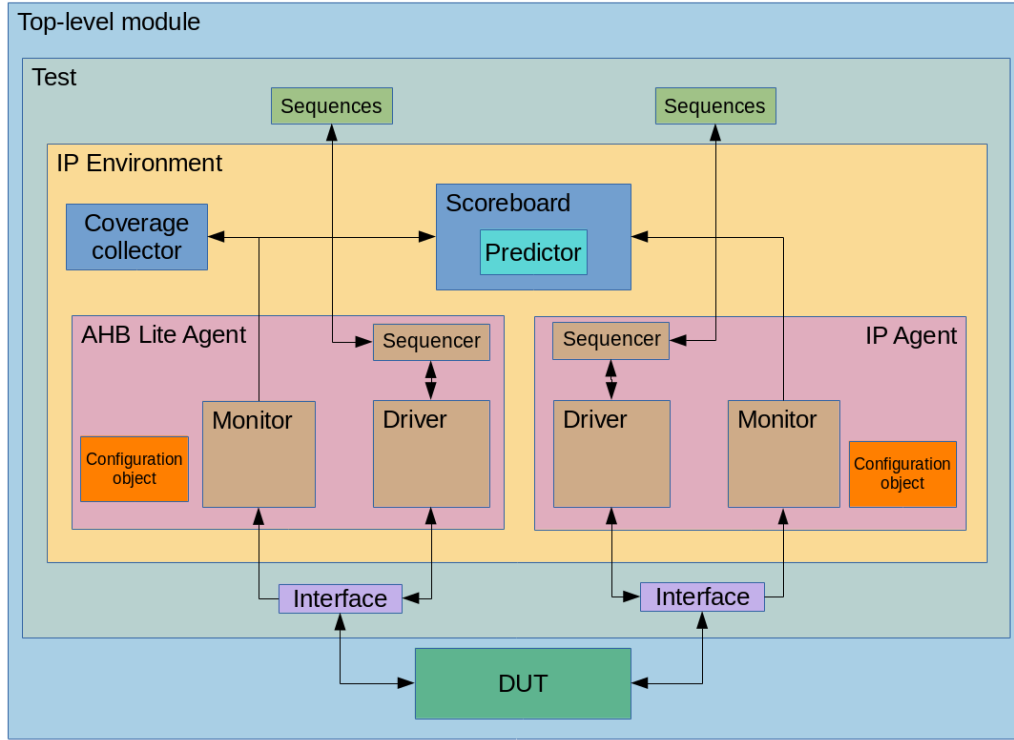


Figure 8. The UVM testbench for the performance tests.

The SystemVerilog AHB-Lite agent hierarchy, interface, and base sequences for driving and monitoring the 32-bit AHB-Lite bus were part of Nokia in-house VIP and were rewritten in Python for the Python testbench. The SystemVerilog AHB-Lite VIP was the only reused part of the testbench. The IP agent was for the DUT's input and output ports outside the AHB-Lite bus. Both agents included a configuration object for switching between active and passive mode and other configurations.

The scoreboard was fed input and output data from both monitors, and it compared predicted and observed responses of the DUT on each clock cycle. The predictor component within the scoreboard generated the predictions based on the input data. There was no reference model of the design available to use with the testbench, so the predictor was developed to model the DUT behaviour at an abstract level in SystemVerilog and Python. Its purpose was to verify the response to random stimuli and to help identify bugs in the testbenches.

The operation of the predictor component involved constantly manipulating bit vectors to mirror the register values of the DUT and to create predicted output data. For these bit vectors, cocotb's *BinaryValue* type was first used in the Python testbench. With the initial implementation, connecting this component slowed down the simulation considerably. Performance analysis of the testbench revealed that methods of the *BinaryValue* type were consuming a major portion of the simulation time. In response, all bit vectors in the predictor were switched to *bitarray* type from the Python package *bitarray*. The *bitarray* type is an array of Boolean values with its functionality implemented in C [31]. Performance gain of the type over *BinaryValue* was not officially measured and recorded, as this was outside the scope of this work, but *bitarray* was clearly faster in this case.

The coverage collector component was a *uvm\_subscriber* with a coverage group, collecting coverage data from AHB-Lite bus operations. There were two different versions of the coverage groups. The first one was used for performance tests and included coverpoints for DUT register addresses, transaction sizes and their cross coverage. The second was made for a coverage test



and included a subset of the bins of the first version. This subset is described more in detail in Section 6.3. In the Python testbench, the coverage collector was implemented with PyVSC as shown in Section 4.1.

The Python testbench used three different methods for constrained randomization. Sequence items in IP agent sequences only had one constraint to limit a field's value to a range of integers, so Python's built-in *random* library was sufficient to randomize the items. Sequence items in AHB-Lite agent sequences used two different, switchable randomization schemes which were compared in the performance tests.

One of the randomization schemes used PyVSC's constrained random features. An example of randomizing sequences with PyVSC was in Section 4.2. Sequences were decorated with `@vsc.randobj` like in the example but the sequence items were not. This is because the decorator had a drastic performance impact on the creation of UVM objects. While the impact to the total execution time is small in many use cases, the tests in this work created only a few sequences but up to 2.5 million sequence items. Therefore, it was far better to randomize sequences and copy the randomized values to sequence items.

The other randomization scheme used pyquesta's SVConduit to transport a sequence item from Python to SystemVerilog with the required information for randomization. The SystemVerilog module would then randomize the sequence item fields and send it back to Python. While Section 4.3 showed an example of unidirectional transfer of sequence items between Python and SystemVerilog with `sv_get` function, this method used `sv_transport` for two-way object exchange.

The interfaces in the Python testbench were implemented in Python as shown in Subsection 3.3.4. They were not functionally equivalent to the SystemVerilog interfaces because they did not have clocking blocks. Currently, cocotb does not offer a direct alternative to SystemVerilog's clocking blocks. *Timer* trigger in cocotb could be used to add arbitrary delay to driving output signals, but it is uncertain if negative skew can be added to the sampling of input signals. Missing this functionality in the Python testbench did not affect test results but adds some error in the codebase comparison.

Since the Python testbench did not use SystemVerilog interfaces, the top testbench module only contained top-level signals and an instantiation of the IP module. Using a top testbench module in such case is not necessary with cocotb, but here its main function was to wrap the IP module, written in VHDL, in a SystemVerilog module. This enabled simulation with Synopsys VCS because cocotb does not currently support VHPI for VCS. Cocotb's default Makefile for VCS also required changes to enable compilation of VHDL files. Probing internal signals of the DUT was still not possible with this arrangement, but access to top-level signals was enough for the tests. The modifications to the Makefile are presented in Appendix 1.

## 6 PERFORMANCE AND CODEBASE COMPARISON

This chapter covers the performance and codebase comparison of the Python and SystemVerilog testbenches introduced in Chapter 5. Lines of code and file size comparison gives one estimate for the effort put into developing the two testbenches. The performance tests measured the simulation performance difference of the testbenches in terms of execution time and memory consumption. Lastly, the coverage test measured the efficacy of SystemVerilog's and PyVSC's random stimulus generation in increasing functional coverage.

### 6.1 Code size comparison

Codebase statistics for the Python and SystemVerilog testbenches are in Table 2. The Python testbench had 30% less lines of code and 20% smaller total file size than the SystemVerilog testbench. Lines of code excludes comment lines and blank lines. SVConduit files were excluded from the comparison since this was an extra feature that did not have a counterpart in the SystemVerilog testbench.

Table 2. Comparison of Python and SystemVerilog testbench codebase

Testbench	Lines of code	Comment lines	Blank lines	Total file size (kB)
Python	1634	237	223	82
SystemVerilog	2325	134	428	103

In SystemVerilog testbench, method prototypes were declared inside a class body with *extern* keyword, and methods were defined outside the body. In Python testbench, methods were defined inside classes without the prototypes because Python does not have them. This is one reason for the less lines of code in the Python testbench. Another reason is the lack of variable declarations in Python. New variables can be assigned at any point while in SystemVerilog variables need to be declared first before assigning a value to it. SystemVerilog classes also had registration macros and constructor definitions which were not needed in the Python testbench as explained in Subsection 3.3.1.

Smaller differences include Python's dictionary comprehension which saved effort of creating bins separately as shown in Section 4.1. No equivalent method of defining bins out of an associative array was found for SystemVerilog. Also, as mentioned in Section 5.2, Python interfaces did not have clocking block functionality of the SystemVerilog interfaces. Clocking blocks added 51 lines of code to the SystemVerilog testbench.

Aside from the differences in implementation, there are several factors that contributed to a difference in lines of code and code size but had no correlation or had an inverse correlation with development effort. In Python testbench, a code formatter tool enforced a maximum line length of 88 characters, whereas in the SystemVerilog testbench, the line length cap was self-imposed and not strictly followed. In effect, the code formatter increased lines of code but reduced development effort. The SystemVerilog testbench commonly had lines with only brackets or the keyword *end* of a *begin-end* block, while in the Python testbench, lines with only brackets were less common. The Python testbench had more comment lines, but it is partly explained by the line length restriction. Comments were mirrored between the testbenches wherever it was applicable. The SystemVerilog testbench had more blank lines, some of which had whitespace characters, increasing the file size. On the other hand, the Python testbench used four spaces for indentation, while the SystemVerilog testbench used only two.

## 6.2 Performance tests

Identical UVM tests were created for both the Python and SystemVerilog testbench to measure memory use and execution time. The tests were executed on a Linux RHEL7 server. The simulators used in the tests were Synopsys VCS Release version T-2022.06\_SP1\_Full64 and Siemens QuestaSim 2022.2\_1 Revision 2022.05. The Python version was 3.7.4. A list of Python packages used in the Python testbench, and their versions, is in Table 3.

Table 3. Python packages used in the Python testbench

Python package	Version
pyuvvm	2.9.0
cocotb	1.7.1
PyVSC	0.7.9
pyquesta	2.5.1
bitarray	2.6.0

For the metrics, CPU time and peak virtual memory size (VSZ) during simulation phase, as reported by the simulators, were used. CPU time is the time spent by the CPU executing the processes spawned by the simulator. Virtual memory size is the memory assigned to all the contributing processes. VSZ is a pessimistic metric for memory consumption since it includes memory that is swapped out, but it was the primary metric in the simulators.

### 6.2.1 Idle tests

The idle tests compare the cost of issuing signal value changes from the Python testbench versus the SystemVerilog testbench. The scoreboard, coverage collector and monitors were uninstantiated, and the drivers only reset the DUT in the beginning but were otherwise idle. A clock signal with a period of 100 ns was generated by a *Clock* class of *cocotb.clock* module in the Python testbench and by a forever looping *initial* block in the top testbench module in the SystemVerilog testbench.

In the first variation of the test, the clock signal was generated for only one clock cycle to measure the baseline for memory use. In the second variation, the clock signal was generated for 150 ms.

Figure 9 shows the peak virtual memory size during the simulation phase of the idle tests. With VCS, the Python testbench had 35% higher VSZ than the SystemVerilog testbench and remained steady regardless of simulation length. With Questa Sim, the SystemVerilog testbench simulation consumed 119 MB of VSZ regardless of test length, but with the Python testbench, VSZ started at 588 MB and rose all the way up to 1320 MB. This could indicate some simulator specific issue since test length had no effect on memory consumption with VCS.

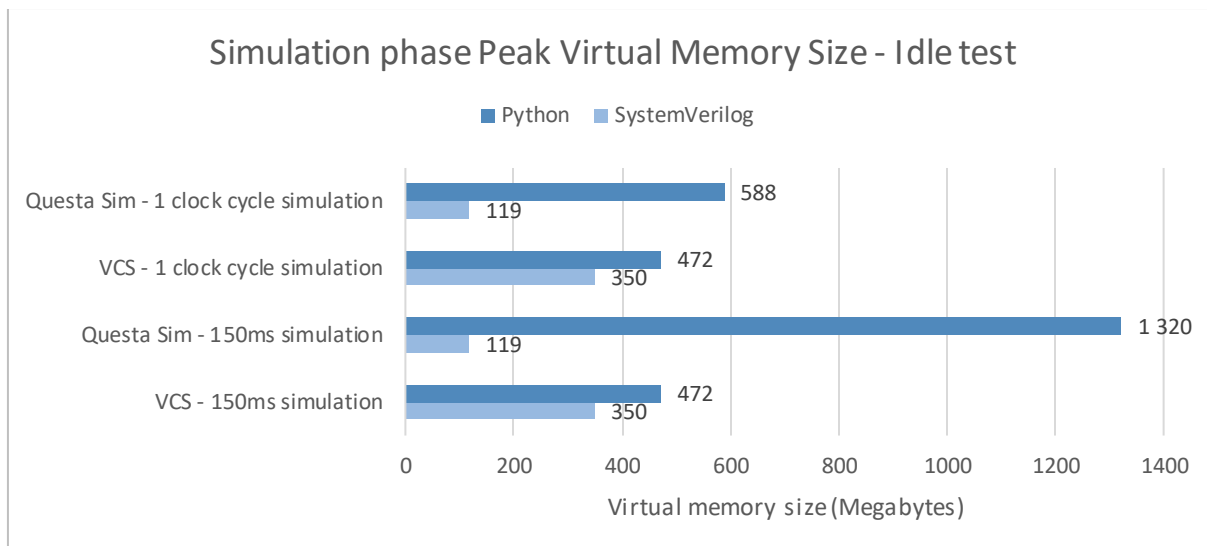


Figure 9. Peak virtual memory size during simulation phase for idle tests.

Figure 10 shows simulation phase CPU time for the 150 ms idle tests. There was a massive difference between the SystemVerilog and Python testbenches, with the Python test taking more than a hundred times longer to simulate.

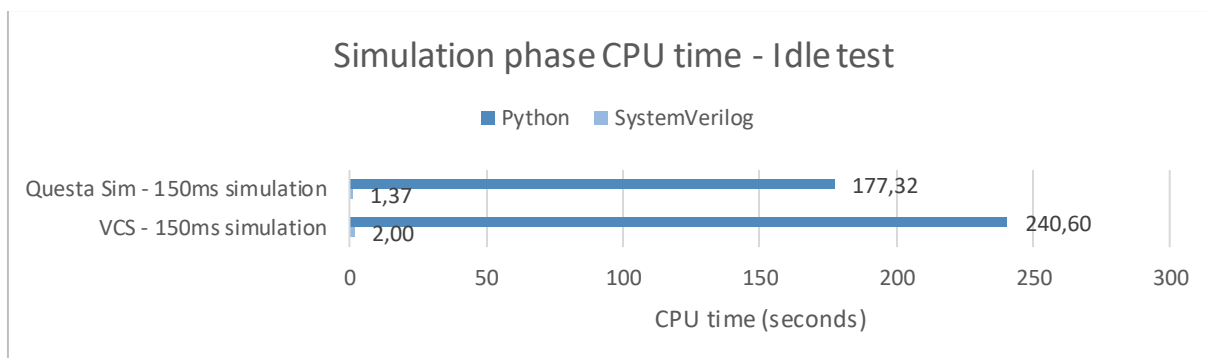


Figure 10. Accumulated CPU execution time during simulation phase for idle tests.

### 6.2.2 Write tests

In the write tests, a register in the DUT was written to via the AHB-Lite bus. A write sequence created transactions that were sent to the AHB-Lite driver to drive the write operations through the bus interface. The IP driver remained idle but monitors in both agents and the scoreboard were enabled. The test looped for 500 000 bus writes, taking 150 ms of simulation time.

In the first variation of the test, the coverage collector was disabled. This removed the effect of PyVSC to focus on the performance of pyuvvm and cocotb. In the second variation, the coverage collector was enabled to see the performance impact of PyVSC's coverage functionality. In the third variation, the coverage collector was once again disabled but every write operation produced a UVM info message in the SystemVerilog testbench and a logging message in the Python testbench. In the first two variations, message verbosity and severity settings were adjusted such that only a few messages were produced during the simulation. This variation highlights the performance impact of UVM reporting system versus Python's logging system.

Figure 11 shows peak virtual memory sizes during the write test simulations. Questa Sim once again displayed a very high VSZ with the Python testbench, with 1450MB compared to 237MB of SystemVerilog. The cause for this issue is not known apart from that it was only present in long simulations. For VCS, the Python testbench had 35% higher VSZ than the SystemVerilog testbench. All the test variants had identical memory consumption. If components were included in the compilation, whether they were instantiated or not had no effect on simulation phase VSZ.

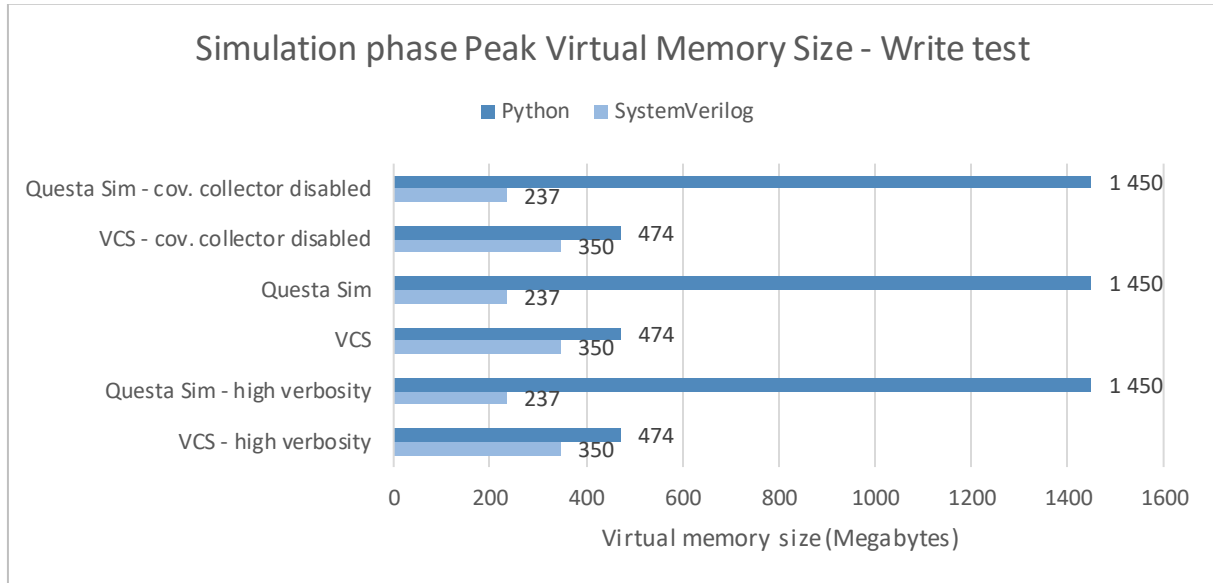


Figure 11. Peak virtual memory size during simulation phase for write tests.

Figure 12 shows the CPU execution time for the write test simulations. Execution times were 12 to 16 times longer with the Python testbench compared to the SystemVerilog testbench on Questa Sim, and 8 to 10 times longer on VCS. The difference in execution time was an order of magnitude smaller than in the idle tests but still significant.

The coverage collector implemented with PyVSC contributed to a 10% increase in execution time, while the impact of the SystemVerilog coverage collector was only two to three percent. In absolute numbers, the difference was 88 to 115 seconds increase in the Python testbench execution time compared to 1 to 3 seconds with the SystemVerilog testbench. Even in relative numbers, PyVSC's functional coverage had a greater impact on performance than its SystemVerilog counterpart.

High verbosity increased simulation time on Questa Sim by 15 seconds with SystemVerilog testbench and 20 seconds with Python testbench. On VCS, the difference was larger – 21 seconds with the SystemVerilog testbench to 39 seconds with the Python testbench.

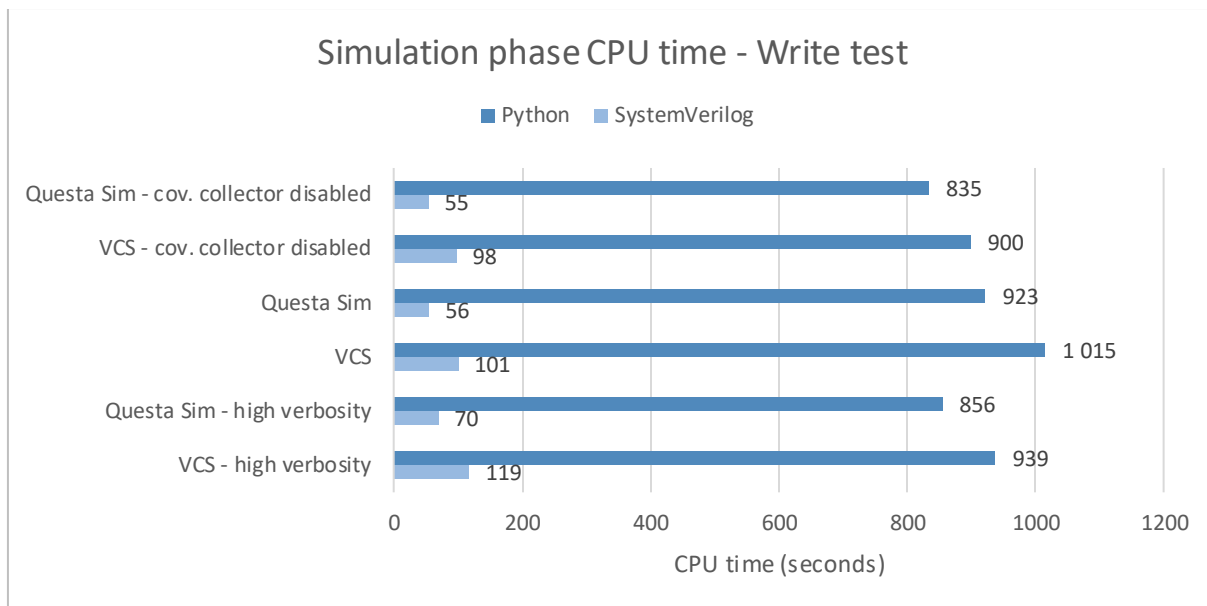


Figure 12. Accumulated CPU execution time during simulation phase for write tests.

### 6.2.3 Random stimulus tests

In the random stimulus tests, both the AHB-Lite agent and the IP agent were driving random stimulus to the DUT. All the testbench components were enabled.

Randomized data was written to a selection of 28 registers in the DUT via AHB-Lite bus in a loop. The order in which the registers were written to was shuffled after each sequence loop. In addition to randomizing the data, the sequence would randomly select a transaction size of one, two or four bytes. Based on the transaction size, the sequence randomized an offset for the address: zero for four-byte transaction, zero or two for two bytes, and from zero to three for one-byte transaction. At the end of each loop, two non-random register writes were issued to clear a register in the DUT.

While the AHB-Lite agent was applying random stimulus to the registers, the IP agent was driving randomized data to one of the DUT's inputs on every fourth clock cycle in a separate thread.

In total, the register write sequence looped 5000 times, taking approximately 45 ms of simulation time.

There were two variations of the Python test. In the first variation, the random stimulus for AHB-Lite bus in the Python testbench was generated using PyVSC. The IP agent's stimulus was generated using Python's built-in random library, since the randomizing function simply selected a value from a range of integers for the data.

In the second variation, instead of PyVSC generating the random stimulus, the AHB-Lite sequence items were transported via SVConduit to a SystemVerilog module which randomized the item fields. After randomization, the items were sent back to the Python testbench.

Figure 13 shows the peak VSZ during the random stimulus tests. Memory consumption on Questa Sim with the Python testbench follows the trend of previous tests. It is now 700 MB less than in the 150 ms write test but 260 MB higher than in the one clock cycle idle test, suggesting a linear increase in VSZ with increasing simulation length. The VSZ difference of SVConduit and PyVSC is insignificant. VSZ difference of Python and SystemVerilog testbenches on VCS supports the earlier results, with the Python testbench having 30% higher VSZ in this test.

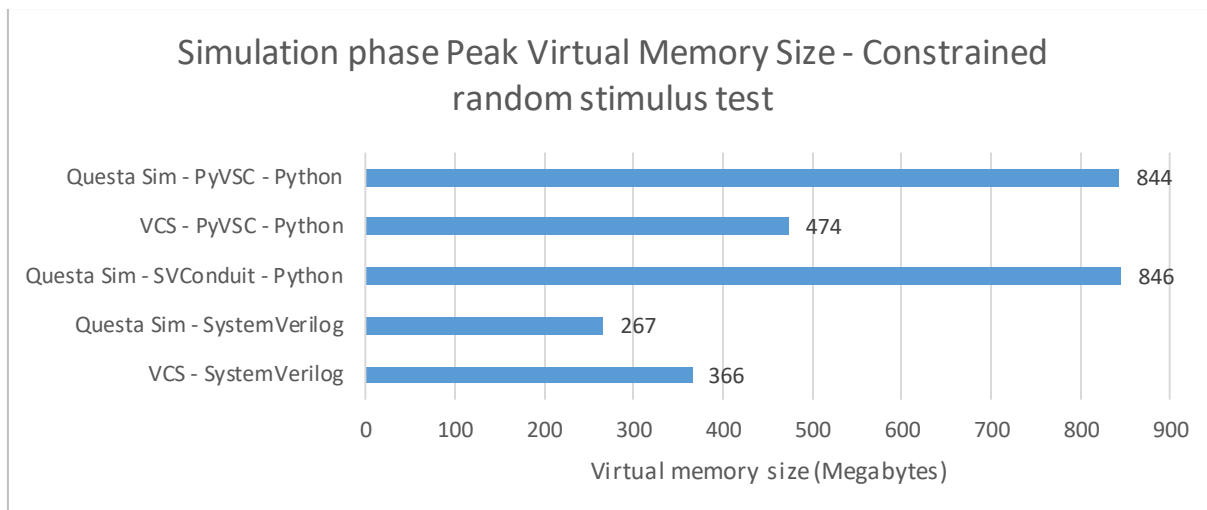


Figure 13. Peak virtual memory size during simulation phase for random stimulus tests.

Simulation phase CPU execution times of the random stimulus tests are in Figure 14. SVConduit version of the test took almost 60% less time to execute than the test with PyVSC. The SVConduit version of the Python testbench was still 9 times slower than the SystemVerilog testbench, however.

While analysing the results, a bug was discovered in a SystemVerilog sequence which caused one of the non-random register writes to not be issued. This reduced the execution time of the SystemVerilog tests by approximately 2%, which makes a negligible difference to the overall comparison.

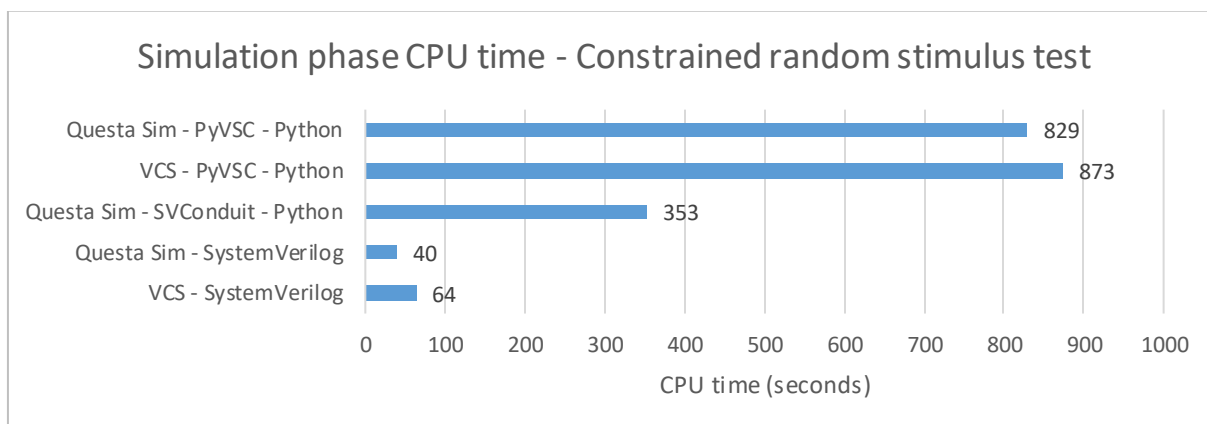


Figure 14. Accumulated CPU execution time during simulation phase for random stimulus tests.

### 6.3 Coverage test

The coverage test compared the coverage accumulation rate of the two testbenches. AHB-Lite write operations were randomized from a range of 120 register addresses and three data sizes. The Python testbench used PyVSC for randomizing the operations. The coverage collector contained a covergroup with cross bins that covered all 360 possible combinations. The coverage percentage of the cross bins was measured as a function of write operations. Each repetition of the test had a different random seed. The tests were executed on Questa Sim only.

The coverage results are in Figure 15. The circles and crosses represent individual measurements, and the curves are polynomial fitted curves to aid visualization. The coverage in Python testbench remained behind SystemVerilog's by 1.6% on average, with the difference starting to show after 400 operations. This means that as the write operation count increased, PyVSC tended more often to produce random values that were already covered than SystemVerilog. The raw test result data is available in Appendix 2.

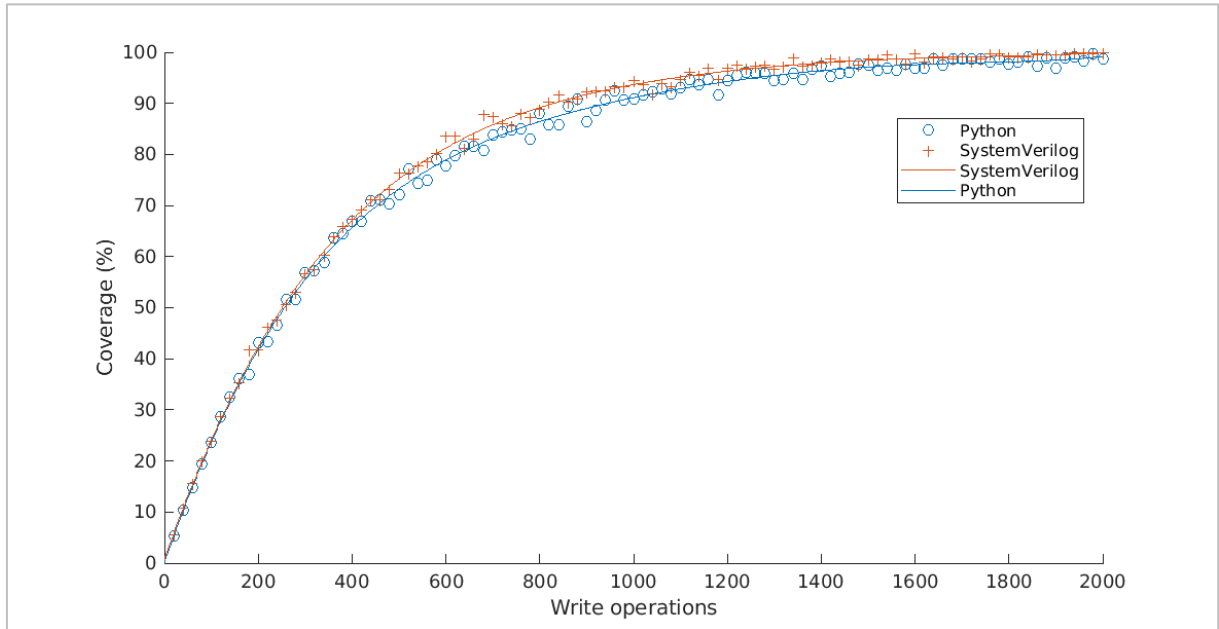


Figure 15. Coverage as a function of write operations for the Python and SystemVerilog testbench in the coverage test.



## 7 DISCUSSION

SystemVerilog provides a rich set of verification features, but Python with *cocotb* was a viable alternative to it for writing testbenches. *PyVSC*'s constrained random verification and functional coverage sufficiently substituted their SystemVerilog counterparts for the test cases in this work. *SVConduit* was successfully used to borrow constrained random features of SystemVerilog in Python. The Python package *bitarray* provided bit vectors with high performance to the predictor component.

As the focus of this thesis was on UVM, it did not cover in detail what SystemVerilog features are available in *cocotb* and other Python libraries, and what are still missing. The novel capabilities that Python may offer for verification were not explored either because the testbench in this work had to be implemented in both languages. These are important topics to cover in future studies.

*Pyuvm* was viable for building a UVM testbench for IP-level verification. Its current features enabled to build a configurable UVM environment with an AHB-Lite VIP, an IP specific agent and a scoreboard with a predictor. A coverage collector implemented with *PyVSC* was also incorporated as a UVM component in the environment.

The incomplete UVM RAL makes subsystem and top-level verification with *pyuvm* challenging at this point of its development. In a typical system-on-chip (SoC), many IPs together form a subsystem, and the complete SoC at top-level consists of many subsystems. The design is verified at each level. UVM RAL helps to reuse verification code from one level to the next. A lower level UVM register model is easy to integrate to a higher level testbench, and the stimulus code can be reused as well when it only refers to the register names in the model, while the model links the names to their actual addresses.

Performance test results of the Python testbench also raised a concern for top-level verification of complex SoCs, where simulation times can become a bottleneck for productivity even with SystemVerilog. Driving the clock signal from the Python testbench was two orders of magnitude slower than from the SystemVerilog testbench. The write tests and random stimulus tests took 8 to 21 times longer to run in Python than in SystemVerilog. Moving the clock signal generation from Python to SystemVerilog would help alleviate the performance issues, but it may not always be feasible. Memory consumption was also notably higher with the Python testbench, but the results only show the difference for small testbenches. More data is needed to see how the memory use scales for large verification environments.

The hybrid solution of randomizing AHB-Lite sequence items in SystemVerilog with *SVConduit* was faster than the *PyVSC* solution to a surprising degree, even though the sequence items had to cross over the Python-SystemVerilog language interface in both directions. The motivation behind using *SVConduit* was to test the feasibility of utilizing SystemVerilog in a Python environment, but it also proved to be a method to improve the performance of the testbench. A future study could explore if an *SVConduit* module is able to pass objects between a Python and a SystemVerilog UVM testbench running at the same time. Enabling the reuse of SystemVerilog legacy VIPs with a Python testbench would facilitate a transition to Python as a verification language.

Implementing the most performance critical data types and algorithms in C is also one way to improve the performance of a Python testbench. Switching the data type of bit vectors in the predictor component from *cocotb*'s *BinaryValue* type to the *bitarray* library's *bitarray* type, implemented in C, resulted in faster simulation execution.

It should be noted that a more complex DUT than the one used in this work would demand more from the simulator to calculate the state of the model, which would affect the performance

difference between the testbenches. The testbenches themselves were also light to execute outside the transaction-level traffic handling. As a result, the simulations had to be extended by looping the test sequences from thousands to hundreds of thousands of times to reach long enough execution times for comparison. Therefore, the test results may not give an accurate estimate for the performance in computationally demanding test scenarios at subsystem and top level.

Lack of need for testbench compilation in Python is an advantage which was not shown in the performance tests due to the simplicity of the testbench. The compilation took less than a second for the SystemVerilog testbench. For large verification environments, a full compilation can consume substantial amount of time.

The coverage test revealed that PyVSC did not produce as uniform distribution of random values as SystemVerilog. While designing the random stimulus test, it was noticed that with PyVSC, certain registers were written to far more often than others when the test selected the target register randomly as in Section 4.2. Therefore, the sequence was changed to go through the registers in order and the coverage test was added to measure coverage accumulation. An update to PyVSC addressed this issue but some difference in uniformness remained as can be seen from the test results.

In addition to pyuvvm, there is another Python package that implements UVM in Python, called `uvm-python`. Its repository page states that it already includes UVM RAL with frontdoor and backdoor access, and TLM 2.0, which are not yet included in `pyuvvm` [32]. Based on its class reference, `uvm-python` is a faithful port of the SystemVerilog UVM implementation to Python without a strong attempt to simplify the implementation to make it more pythonic like `pyuvvm` [33]. A comparison of `uvm-python` and `pyuvvm` could be considered for a future topic of study.

Let us summarize the advantages of using Python as a hardware verification language as opposed to SystemVerilog. Python is the most popular programming language and using it for verification would allow businesses to draw from a much larger pool of talent than the people who know SystemVerilog. As a dynamically typed language with features like duck typing, reflection and multiple inheritance, Python gives a degree of flexibility and conciseness to the code that is not possible with SystemVerilog. Since Python is a dynamic programming language, a Python testbench does not need to be compiled before running a test, which can potentially save a lot of time during development. And with the thriving open-source community around Python, there are hundreds of thousands of shared packages to utilize when coding with Python.

The disadvantages, of course, cannot be ignored either. SystemVerilog was standardized by IEEE almost 18 years ago in 2005. Accellera released the first version of UVM standard and its SystemVerilog class library in 2011. SystemVerilog and its UVM implementation have reached a level of maturity that Python packages for verification do not currently offer. The packages need more time to develop to improve quality and performance and to bring them closer to feature parity with SystemVerilog and SV-UVM. As the work is mostly being done by volunteers, the timeline to reach that point is uncertain, and the development may stagnate without financial support. And although Python's dynamicity was mentioned as an advantage, there are also merits to statically typed languages like SystemVerilog. A separate code compilation phase enables compiler optimization, which can explain some of the observed difference between SystemVerilog and Python simulation execution times. Also, compilation can detect errors before they become run-time errors. In addition, explicitly declared variable types convey the programmer's intent, although type hints are available in modern Python. [21]

Considering all the above, four main use cases are suggested for a Python UVM testbench based on pyuvvm in its current state. The first is to verify designs with sufficiently low complexity, with the register count perhaps in the dozens, where slower simulation speed and the lack of UVM RAL can be managed. The DUT in this thesis is a good example of a design for which a Python testbench is a viable option. The second is to use it in any cocotb based verification environment. Naturally, all Python testbenches can benefit from UVM's features – the structure, the component reusability and the TLM implementation, to name a few. The third use case is to use it with open-source HDL simulators. The two prominent free and open-source HDL compilers/simulators that support SystemVerilog, Icarus Verilog and Verilator, do not currently support SV-UVM. Pyuvvm enables UVM on these EDA tools. The fourth use case is to use it for research. Python as a hardware verification language is not well studied and discovering novel ways to use it for RTL verification could bring great value to the semiconductor industry.

## 8 SUMMARY

Testbenches for functional verification of integrated digital circuits are typically written in SystemVerilog, and it is also the language for the Universal Verification Methodology as defined in the IEEE standard. But recently, Python has been emerging as a new language for functional verification. *cocotb* provides a framework to write testbenches in Python and to connect the testbench with an HDL simulator. Enabled by *cocotb*, Python libraries are being developed for different verification areas, including UVM.

In this thesis, the viability of Python for UVM based verification with *pyuvvm* and other available libraries was studied. *Pyuvvm* is an implementation of UVM in Python. Its key features currently include base classes, reporting system based on Python's logging library, factory classes, phasing, TLM 1.0, predefined component classes, sequences, sequencer, and sequence items. A notable feature still under development is the UVM RAL, which limits the capability of *pyuvvm* especially for subsystem and top-level verification of SoCs.

*Pyuvvm* was successfully used to develop a UVM testbench for an AHB-Lite slave IP component, and a matching testbench was also built in SystemVerilog for comparison. In addition to *pyuvvm* and *cocotb*, the Python testbench used *PyVSC* for functional coverage and constrained random stimulus, and *bitarray* for calculations with bit vectors. With *SVConduit*, the Python testbench also had the ability to exchange objects with a SystemVerilog module, which was used as an alternative option for randomizing sequence items.

Based on the codebase comparison, developing testbenches may be faster in Python than SystemVerilog. The Python testbench had 30% less lines of code and 20% smaller total file size than the SystemVerilog testbench. Features of Python and *pyuvvm* lead to shorter code lines and less repetitive code like factory registration macros and constructors with only a call to the constructor of the base class. The results include some error due to code formatting and minor differences in functionality. Differences in language syntax also affected line count without comparable effect on development effort.

The Python testbench did not fall far behind in functional coverage accumulation, but it had performance issues. A batch of UVM tests were executed for both testbenches on Siemens Questa Sim and Synopsys VCS for performance comparison. Clock signal generation in the Python testbench was two orders of magnitude slower than in the SystemVerilog testbench. For tests involving more of the testbench with AHB-Lite write operations and random stimulus generation, execution times were 8 to 21 times longer in the Python testbench. In terms of memory consumption, the Python testbench had 30–35% higher VSZ on VCS. On Questa Sim, there were indications of a memory leak with the Python testbench in longer simulations, where VSZ could be several times higher than with the SystemVerilog testbench.

With the large performance gap and the UVM RAL missing in *pyuvvm*, SystemVerilog UVM may still be the superior option for verifying complex designs like SoCs. However, the libraries used in this work already enable advanced verification methodologies with Python and open-source HDL simulators, and the ecosystem for Python RTL verification continues to evolve. In its current state, a Python UVM testbench is a viable option for verifying simple designs and is also a promising subject of research.

## 9 REFERENCES

- [1] IEEE. (2017) IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language, IEEE Std 1800-2017, p. 1182–1183, 953, 170–171, 553–558, 748, 938
- [2] Cass S. (2022) Top Programming Languages 2022. IEEE Spectrum (Accessed 20.5.2023). URL: <https://spectrum.ieee.org/top-programming-languages-2022>
- [3] Python Software Foundation. PyPI - The Python Package Index (Accessed 19.3.2023). URL: <https://pypi.org/>
- [4] Potential Ventures, SolarFlare communications. cocotb (Accessed 4.5.2022). URL: <https://www.cocotb.org/>
- [5] Potential Ventures, Solarflare Communications, cocotb contributors. cocotb, Github repository (Accessed 4.5.2022). URL: <https://github.com/cocotb/cocotb>
- [6] cocotb documentation: Simulator support (Accessed 27.12.2022). URL: [https://docs.cocotb.org/en/stable/simulator\\_support.html](https://docs.cocotb.org/en/stable/simulator_support.html)
- [7] Ashmanskas W.J., Dandoy J.R., Dressnandt N.C, Keener P.T., Kroll J., Lipeles E., Lu S., Newcomer F.M., Nikolica A., Rosser B.J., Thomson E. (2023) Verification of simulated ASIC functionality and radiation tolerance for the HL-LHC ATLAS ITk Strip Detector. Journal of instrumentation, Vol. 18, 2023.
- [8] Foster H., Wilson Research Group, Siemens EDA. (2022) 2022 Wilson Research Group IC/ASIC functional verification trends, p. 11–12
- [9] Foster H., Wilson Research Group, Siemens EDA. (2022) 2022 Wilson Research Group FPGA functional verification trends, p. 13–14
- [10] IEEE. (2020) IEEE Standard for Universal Verification Methodology Language Reference Manual, IEEE Std 1800.2-2020, p. 7, 12, 15–16, 18, 54, 69–70, 95, 120, 130–133, 162, 173–175, 177, 194, 404, 429
- [11] Salemi R., Siemens. pyuvm, Github repository (Accessed 5.5.2022). URL: <https://github.com/pyuvm/pyuvm>
- [12] IEEE. (2019) IEEE Standard for VHDL Language Reference Manual, IEEE Std 1076-2019, p. 355
- [13] cocotb contributors. cocotb documentation: GPI Library Reference (Accessed 2.5.2022). URL: [https://docs.cocotb.org/en/stable/library\\_reference\\_c.html](https://docs.cocotb.org/en/stable/library_reference_c.html)
- [14] cocotb contributors. cocotb wiki: cocotb internals (Accessed 09.08.2022). URL: <https://github.com/cocotb/cocotb/wiki/cocotb-Internals>
- [15] Python Software Foundation. Python 3.7.14 Documentation: Extending and Embedding the Python interpreter (Accessed 27.12.2022). URL: <https://docs.python.org/3.7/extending/index.html>
- [16] Knuth D. (1997) The Art of Computer Programming, Volume 1: Fundamental Algorithms, Third Edition. p. 186–187, 193–195
- [17] cocotb contributors. cocotb documentation: Coroutines and Tasks (Accessed 12.03.2023). URL: <https://docs.cocotb.org/en/stable/coroutines.html>
- [18] Accellera. (2015) Universal Verification Methodology (UVM) 1.2 User's Guide. p. 1–2, 4, 7–8, 34, 38, 40–41, 58–59, 70, 75–78
- [19] Python Software Foundation. The Python Tutorial: Classes (Accessed 10.08.2022). URL: <https://docs.python.org/3/tutorial/classes.html>

- [20] Python Software Foundation. Python 3.7.14 Documentation: Glossary (Accessed 09.08.2022). URL: <https://docs.python.org/3.7/glossary.html>
- [21] Tratt L. (2009) Dynamically typed languages. *Advances in Computers*, vol. 77, p. 149–184, July 2009
- [22] Rossum G.V. Method Resolution Order (Accessed 09.08.2022). URL: <http://python-history.blogspot.com/2010/06/method-resolution-order.html>
- [23] Siemens. (2021) Universal Verification Methodology UVM Cookbook. p. 75
- [24] Cocotb maintainers, Potential Ventures, SolarFlare communications. cocotb-bus, Github repository (Accessed 30.11.2022). URL: <https://github.com/cocotb/cocotb-bus>
- [25] Piziali, A. (2008) Functional Verification Coverage Measurement and Analysis. First Edition. p. 39–40
- [26] Ballance M. PyVSC, Github repository (Accessed 30.11.2022). URL: <https://github.com/fvutils/pyvsc>
- [27] Cieplucha M., Pleskacz W.A. Cocotb-coverage, Github repository (Accessed 30.11.2022). URL: <https://github.com/mciepluc/cocotb-coverage>
- [28] Ballance M., Contributors. PyVSC coverage. PyVSC documentation (Accessed 28.05.2023). URL: <https://fvutils.github.io/pyvsc/coverage.html>
- [29] Spear, C., Tumbush, G. (2012) SystemVerilog for Verification: A Guide to Learning the Testbench Language Features. p. 169–170, 172
- [30] Salemi R., Siemens. pyquesta (Accessed 30.11.2022). URL: <https://pypi.org/project/pyquesta/>
- [31] Schnell I. bitarray, Github repository (Accessed 30.11.2022). URL: <https://github.com/ilanschnell/bitarray>
- [32] Poikela T. uvm-python, Github repository (Accessed 29.05.2023). URL: <https://github.com/tpoikela/uvm-python>
- [33] Poikela T. uvm-python Class Reference (Accessed 29.05.2023). URL: [https://uvm-python.readthedocs.io/en/latest/uvm\\_1.2\\_class\\_reference.html](https://uvm-python.readthedocs.io/en/latest/uvm_1.2_class_reference.html)

## 10 APPENDICES

- Appendix 1    The modified cocotb's Makefile for VCS to enable VHDL compilation
- Appendix 2    Table of coverage test results

## Appendix 1 The modified cocotb's Makefile for VCS to enable VHDL compilation

```
#####
# Copyright (c) 2013 Potential Ventures Ltd
# Copyright (c) 2013 SolarFlare Communications Inc
# All rights reserved.
#
# Redistribution and use in source and binary forms, with or without
# modification, are permitted provided that the following conditions are met:
# * Redistributions of source code must retain the above copyright
#   notice, this list of conditions and the following disclaimer.
# * Redistributions in binary form must reproduce the above copyright
#   notice, this list of conditions and the following disclaimer in the
#   documentation and/or other materials provided with the distribution.
# * Neither the name of Potential Ventures Ltd,
#   SolarFlare Communications Inc nor the
#   names of its contributors may be used to endorse or promote products
#   derived from this software without specific prior written permission.
#
# THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND
# ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
# WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
# DISCLAIMED. IN NO EVENT SHALL POTENTIAL VENTURES LTD BE LIABLE FOR ANY
# DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
# (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
# LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
# ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
# (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
# SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
#####

# Based on Colin Marquardt's work at:
# https://github.com/cocotb/cocotb/commit/302e122fe04530ab7c7c292ba05f20573723c1d6

include $(shell cocotb-config --makefiles)/Makefile.inc

ifeq ($(TOPLEVEL_LANG),vhdl)

# Unchanged part omitted

RTL_LIBRARY ?= $(TOPLEVEL)

$(SIM_BUILD)/synopsys_sim.setup:
    @mkdir -p $(SIM_BUILD)/work
    @mkdir -p $(SIM_BUILD)/$(RTL_LIBRARY)
    -@rm -f $@
    @echo "WORK > $(RTL_LIBRARY)" >> $@
    @echo "$(RTL_LIBRARY) : ./$(RTL_LIBRARY)" >> $@

# Compilation phase
$(SIM_BUILD)/simv: $(SIM_BUILD) $(SIM_BUILD)/synopsys_sim.setup $(VERILOG_SOURCES) \
    $(VHDL_SOURCES) $(SIM_BUILD)/pli.tab $(CUSTOM_COMPILE_DEPS)
    LD_LIBRARY_PATH=$(LIB_DIR):$(LD_LIBRARY_PATH) TOPLEVEL=$(TOPLEVEL)
ifeq ($(VERILOG_SOURCES),)
    cd $(SIM_BUILD) && \
        vlogan -sverilog -nc -l sv_comp.log -work $(RTL_LIBRARY) $(PLUSARGS) $(EXTRA_ARGS) \
        $(GPI_ARGS) $(COMPILE_ARGS) $(SV_COMPILE_ARGS) $(VERILOG_SOURCES)
endif
```



```

ifneq ($(VHDL_SOURCES),)
    cd $(SIM_BUILD) && \
        vhdlan -nc -l vhdl_comp.log -work $(RTL_LIBRARY) $(PLUSARGS) $(EXTRA_ARGS)
$(GPI_ARGS) \
    $(COMPILE_ARGS) $(VHDL_COMPILE_ARGS) $(VHDL_SOURCES)
endif
cd $(SIM_BUILD) && \
$(CMD) -top $(TOPLEVEL) $(PLUSARGS) +acc+1 +vpi -P pli.tab -V -notice \
-timescale=$(COCOTB_HDL_TIMEUNIT)/$(COCOTB_HDL_TIMEPRECISION) \
$(EXTRA_ARGS) $(ELAB_ARGS) -debug -load $(shell cocotb-config --lib-name-path vpi vcs) -l
elab.log

# Execution phase
$(COCOTB_RESULTS_FILE): $(SIM_BUILD)/simv $(CUSTOM_SIM_DEPS)
    -@rm -f $(COCOTB_RESULTS_FILE)

-PYTHONPATH=$(LIB_DIR):$(PWD):$(PYTHONPATH)
LD_LIBRARY_PATH=$(LIB_DIR):$(LD_LIBRARY_PATH) \
    MODULE=$(MODULE) TESTCASE=$(TESTCASE) TOPLEVEL=$(TOPLEVEL)
TOPLEVEL_LANG=$(TOPLEVEL_LANG) \
    $(SIM_CMD_PREFIX) $(SIM_BUILD)/simv +define+COCOTB_SIM=1 $(SIM_ARGS)
$(EXTRA_ARGS) | tee $(SIM_BUILD)/sim.log

$(call check_for_results_file)

# Unchanged part omitted

```

Appendix 2 Table of coverage test results

Write operations	Cross Coverage (%), Python	Cross Coverage (%), SystemVerilog
20	5.28	5.55
40	10.28	10.83
60	14.72	15.55
80	19.44	20.00
100	23.61	23.88
120	28.61	28.61
140	32.50	32.22
160	36.11	35.27
180	36.94	41.66
200	43.06	41.66
220	43.33	46.11
240	46.67	47.50
260	51.67	50.55
280	51.67	53.05
300	56.94	56.66
320	57.22	57.50
340	58.89	60.27
360	63.61	63.88
380	64.44	65.83
400	66.94	67.22
420	66.94	69.16
440	70.83	71.11
460	71.11	71.11
480	70.28	73.05
500	72.22	76.38
520	77.22	76.38
540	74.44	77.77
560	75.00	78.61
580	78.89	80.27
600	77.78	83.61
620	79.72	83.61
640	81.67	81.11
660	81.67	83.05
680	80.83	87.77
700	83.89	87.50
720	84.44	86.11
740	84.72	85.55
760	85.00	88.05
780	83.06	87.22
800	88.06	88.88
820	85.83	90.27
840	85.83	91.66

860	89.44	90.27
880	90.83	90.83
900	86.39	92.22
920	88.61	92.50
940	90.56	92.50
960	92.50	93.33
980	90.56	93.33
1000	90.83	94.44
1020	91.67	93.61
1040	92.22	91.94
1060	92.78	93.88
1080	91.94	93.33
1100	93.06	94.72
1120	94.72	96.11
1140	93.61	95.55
1160	94.72	96.94
1180	91.67	94.72
1200	94.44	96.94
1220	95.56	97.50
1240	96.11	96.94
1260	95.83	97.22
1280	95.83	97.50
1300	94.44	96.66
1320	94.72	97.22
1340	95.83	98.88
1360	94.72	97.22
1380	96.67	97.50
1400	97.22	98.05
1420	95.28	98.61
1440	95.83	98.33
1460	96.11	98.33
1480	97.78	97.50
1500	97.50	98.61
1520	96.39	98.61
1540	96.94	99.44
1560	96.39	98.61
1580	97.78	97.77
1600	96.94	99.72
1620	96.94	98.05
1640	98.61	99.16
1660	97.50	99.16
1680	98.61	98.61
1700	98.61	99.16
1720	98.61	98.33
1740	98.61	98.61

1760	98.06	99.72
1780	98.61	99.72
1800	97.78	99.16
1820	98.06	99.16
1840	99.17	99.16
1860	97.22	99.72
1880	98.89	99.44
1900	96.94	99.44
1920	98.89	99.44
1940	99.17	100.00
1960	98.33	100.00
1980	99.72	99.72
2000	98.61	100.00