An implementation of a Python-based verification environment using PyUVM and cocotb

Clara Harvig Bjerrum Kasper Hesse Stefano Minigutti
Aamir Sohail Nagra João Carvalho Jacob Sander Andersen
Andrei Lavric

Abstract

In recent years, many open-source projects have emerged making hardware design and verification possible without the need for the common proprietary SystemVerilog tools. This paper presents a detailed implementation of a testbench using *PyUVM*, an implementation of the Universal Methodology Framework in Python. The use case is a testbench for a memory arbiter, verified using reusable Universal Verification Components and Constrained Random Verification to ensure all corner cases are hit. When building Universal Verification Methodology (UVM) testbenches that utilize Constrained Random Verification, the Universal Verification Component is a key element to achieving high reusability and flexibility.

To connect the testbench with the Device Under Test a Python class was implemented which provides an abstraction layer between the Universal Verification Component and the Device Under Test. An API for the Register Abstraction Layer was also developed on top of the existing Register Abstraction Layer in *PyUVM* for register programming. The use case also presents how a C-based reference model may be integrated with a Python-based testbench.

To investigate the state of open-source verification, a detailed comparison of the open-source Python libraries for constrained randomization *PyVSC*, *cocotb-coverage*, and *constrainedrandom* is presented. Here, we consider both randomization speed and the quality of the randomization result. The *cocotb-coverage* and *PyVSC* libraries also support collecting coverage which serves as an evaluation matrix for stimulus. The paper presents the current state of the coverage collection capabilities.

In the end, we conclude on the current state of open-source verifications and address the outstanding challenges in performing CRV with open-source tools.

Keywords— cocotb, pyuvm, UVM, UVC, coverage, testbench, randomization, memory arbiter, verification, open-source.

1 Introduction

The verification of digital Integrated Circuits (IC) plays a pivotal role in ensuring that the design meets specific requirements and functions, as intended during the design and manufacturing phases. Testbenches, which are often written in Hardware Description Languages (HDL) like Verilog, VHDL, and SystemVerilog, are employed to verify the implementation of the hardware.

The UVM framework, implemented in SystemVerilog, supports Constrained Random Verification (CRV) methodology through the use of constrained random stimulus generation and is used to create Verification IP (VIP) for the verification of intricate designs. One notable challenge in this process is the need for proprietary licensed tools to execute complex testbenches. However, many open-source projects have emerged, such as *Icarus Verilog* [**?**], *Verilator* [**?**], and *cocotb* [**?**], offering accessible options for hardware verification without the need for the common proprietary tools. Another challenge, worth mentioning, is the difficulty of finding human resources with HDL skills. Enabling hardware design verification with more common programming languages brings the possibility of finding more verification engineers. This fact has generated research into alternative languages for hardware description and verification.

The open-source Python library *cocotb*, offers a coroutine-based framework that can be used for crafting testbenches for digital designs, as introduced in [**?**]. This framework enables the verification environment to directly interact with design ports in the testbench, by simplifying the design instantiation and reducing its complexity. Based on *cocotb*, several related open-source libraries have emerged, allowing also the implementation of CRV and coverage collection. This also has ignited additional research into creating a Python-based UVM framework, to enable the creation of UVCs and thus increase the adoption of these open-source tools for verification.

A large part of modern digital design verification is the development of UVM testbenches, implementation of UVCs, setting up a verification environment, and writing test suites. UVCs are protocol-specific and can be implemented with several configurations, which enhances the reusability and scalability of complex designs. By using UVCs the testbench development time can be drastically reduced where several instances of standard interfaces exist.

PyUVM is an open-source Python library based on the UVM library, designed for creating UVM testbenches. It encompasses various constructs similar to those in SystemVerilog, including the UVM ConfigDB, TLM 1.0 ports, factory methods, and more. *PyUVM* uses *cocotb* as the bridge between Python and the simulator, allowing one to interact with the simulator and schedule simulation events.

The remainder of this paper is structured as follows. Section 2 presents the implementation of a Python-based UVM testbench for a memory arbiter using *PyUVM*. It covers how to connect the Device Under Test (DUT) and testbench, and the generic UVCs that were implemented. It also showcases how a reference model, implemented in C, may be integrated with a Python-based testbench. Next, Section 3 evaluates and compares various open-source Python libraries that implement constrained randomization, and Section 4 presents a comparison of some Python libraries that implement coverage collection. In Section 5 we enumerate several challenges that were found while developing the testbench using open-source tools that must be addressed before open-source verification becomes a viable alternative to the closed-source tools. Finally, in Section 6 we present the conclusions of this work, and future work ideas are presented.

The toolchain used for the implementation and verification of the work described in this paper is described below:

- Icarus Verilog (v11.0), the primary simulator.
- Python (v3.12), the primary Python version.
- *cocotb* (v1.8.1), a Python library used for verification.
- *PyUVM* (v2.9.1), an implementation of UVM in Python.
- *cocotb-coverage* (v1.1.0), a Python library for constraint solving and coverage collection.

- *PyVSC* (v0.8.6), a Python library for constraint solving and coverage collection.
- *constrainedrandom* (v1.1.2), a Python library used to facilitate constrained randomization.

This document should not be seen as an introductory tutorial to the UVM framework, *cocotb*, or *PyUVM*. It is assumed that the reader already has some knowledge of these topics. If this is not the case, we refer them to the following resources [**?**, **?**, **?**].

2 Python-based UVM testbench for a Memory Arbiter

The use case presented describes the implementation of a Python-based testbench for a multi-master memory arbiter as shown in Figure 2.1. A memory arbiter is a device used in a shared memory system to decide, for each memory cycle, which client will be allowed to access the shared memory. The device uses two data transfer protocols: the SyoSil Data Transfer Protocol (SDT) is used by the external clients to make requests to access the memory, as explained in Section 2.2; and the Advanced Peripheral Bus (APB) for allowing direct operations into the registers. The memory arbiter has a default static priority for the master clients connected to the arbiter, but the priority can be dynamically changed by writing into the two registers handling the priority, the "Control" and the "Dynamic Priority" registers. The arbiter should always observe the latest priority set.

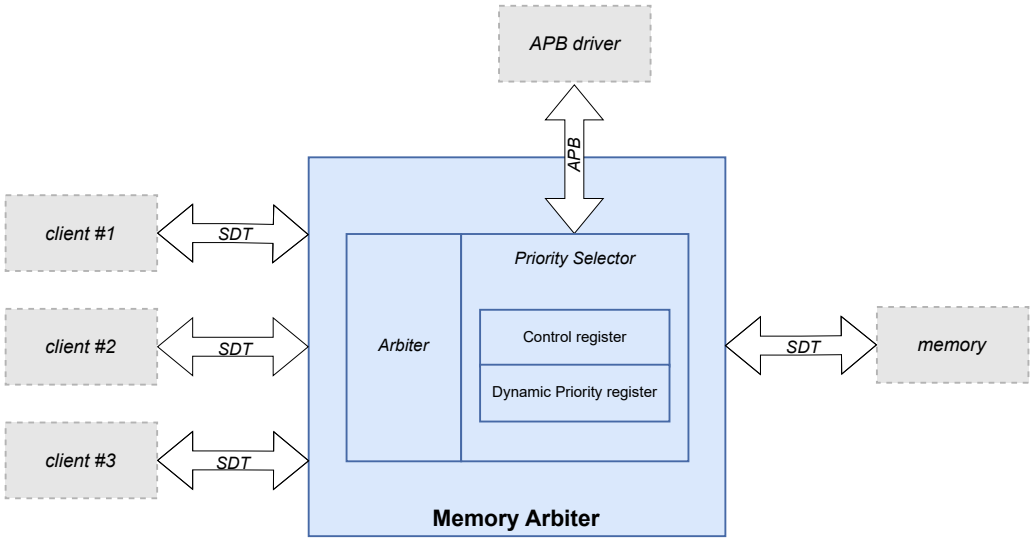

Figure 2.1: Block diagram of the memory arbiter

The aim of the presented use case is to verify the memory arbiter, implemented in SystemVerilog and also described as the DUT, which is pursued through the creation of a testbench along with a reference model for checking results. Constrained randomization has been used to generate random tests and coverage has been implemented to observe the quality of the testing for the DUT.

The UVM testbench has been fully implemented in Python using *PyUVM*. The testbench follows the same steps as the ones implemented in SystemVerilog, as will be described. The following will give an overview of the UVM testbench setup using *PyUVM*, a description of the developed UVCs, testbench components, and the verification approach.

2.1 Testbench Architecture

The testbench is created by using the UVM base classes from *PyUVM* and extending them to the presented use case. An overview of the testbench setup can be seen in Figure 2.2. The top entity is the `uvm_test` class containing the `uvm_environment`, a configuration object, a top sequence, and interfaces for connecting to the DUT. These interfaces are Python objects implemented to facilitate the connection between the memory arbiter and the testbench, further explanation in Section 2.4.

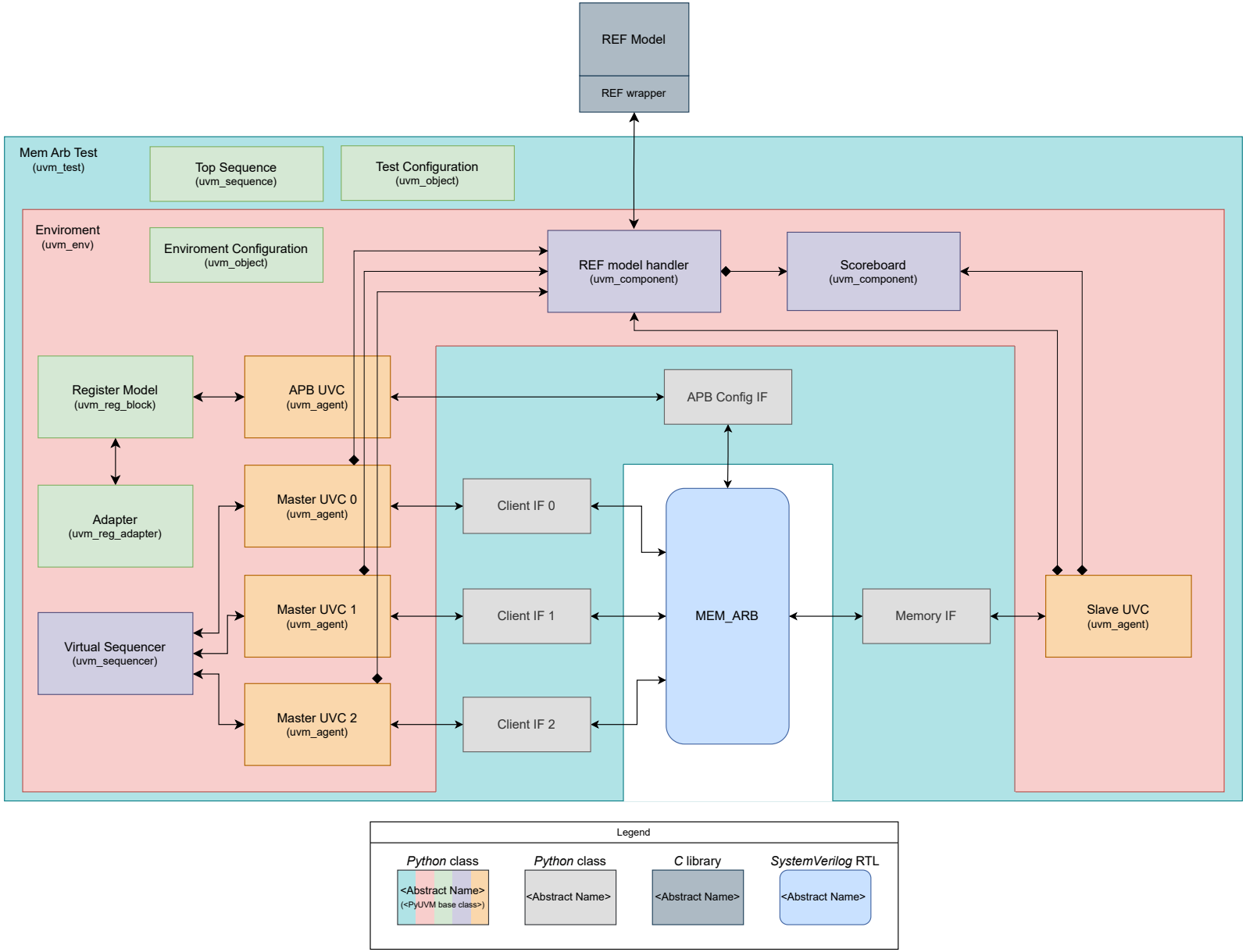

Figure 2.2: Diagram of the memory arbiter testbench

The environment consists of three master UVCs and a single slave UVC responsible for handling data between the testbench and the DUT through the interfaces. The three master UVCs act as clients, while the slave UVC acts as a memory and handles the transactions. The environment also contains the APB-UVC responsible for writing to the registers in the DUT. To further handle writing to registers, the environment has a register model and an adapter for keeping track of the registers within the DUT. Additionally, the environment contains a virtual sequencer, a scoreboard, and a reference model handler.

2.2 SDT protocol

The UVC uses the SDT which is a simple synchronous data transfer protocol that may be used on, e.g., a bus between a processing core and a peripheral. It uses a common address line and separate read/write data lines. Once a request is served, it may be acknowledged at the earliest one clock cycle later. A timing diagram outlining the behavior of the protocol is shown in Figure 2.3.

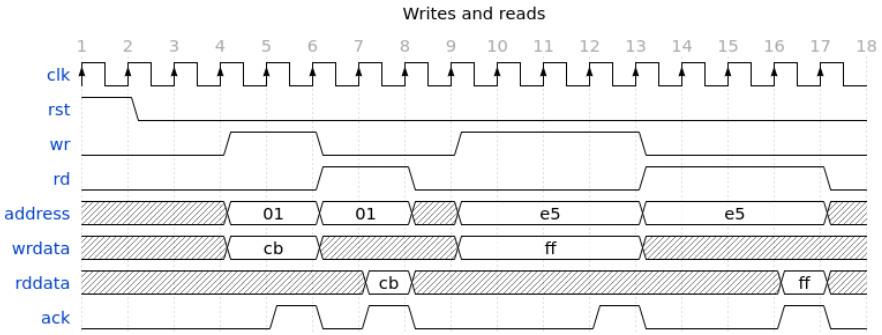

Figure 2.3: Example read and write behavior in the SDT protocol

2.3 Universal Verification Components

The testbench contains several UVCs for handling transactions within the testbench. The purpose of developing general UVCs is to promote reuse across verification projects. This project uses the SDT-UVC which is a UVM agent that has been developed to handle the SDT protocol.

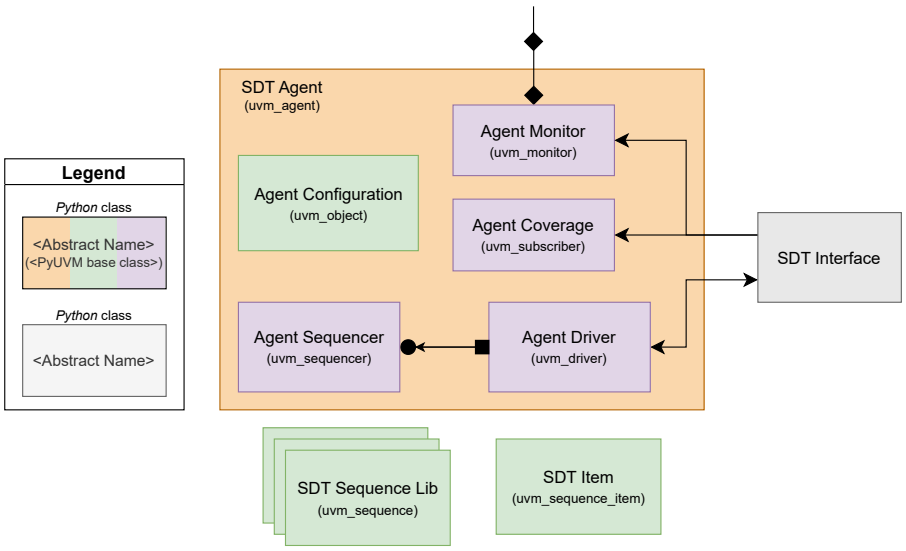

Figure 2.4: SDT UVC

The SDT agent contains a driver, monitor, coverage collection, and sequencer component, as well as a configuration object. Furthermore, the UVC has an instance of the SDT interface, as well as SDT items and sequences.

Everything within the agent is protocol-specific and designed to handle the SDT protocol. The SDT interface ensures that signals from the DUT, regardless of their name, can be connected to the SDT-UVC. The SDT item is used for sending transactions in the testbench and both the driver and the monitor are designed to handle these specific transactions. The driver in the SDT agent can be configured using the agent configuration object as either a master or a slave, allowing the UVC to be used as both in the testbench.

2.4 DUT and testbench connections

By design, *cocotb* always has a reference to the DUT which is accessible through the global variable `cocotb.top`. This allows the connection between the testbench and the DUT by accessing signals directly from the top level of the DUT using this constructor, e.g., `cocotb.top.<signal_name>`. This is a handle to the signal object which can be written to and read from using the object `cocotb.top.<signal_name>.value`.

As mentioned before, a Python object was implemented to facilitate the connection between the memory arbiter and the testbench, through the *cocotb* constructor. This interface is represented as "Client IF X" in Figure 2.2. This object contains the members for all the signals available in the protocol, operating as a typical SystemVerilog interface class. The interface also contains members for the clock and reset signals, which must be set when instantiated. From that point on, all inputs and outputs are probed through this interface, decoupling it from the DUT.

The interfaces for the UVC are instantiated in the base test and stored in the configuration object using UVM ConfigDB, during the build phase. The signals are later connected, during the connect phase of the base test, storing the signals objects for the correct DUT ports in each UVC.

2.5 Reference model

To validate the behavior of the DUT, a common approach is to implement a software reference model. The reference model receives the same inputs as the DUT and generates outputs that are defined to be correct. The correctness of the DUT is then checked by comparing the outputs of the DUT and the reference model in the scoreboard.

Using a C reference model is facilitated by the Python/C API [**?**] that allows one to call Python functions from C, or access C functions from Python. A C library, named "REF wrapper", for the reference model was created to call all functions from Python. The function `add_item_wrapper` is used to send an item from an agent to the reference model. As shown in Listing 2.1, this function unpacks its arguments, extracts fields of the passed object, and then calls the function `add_item` defined on the reference model library.

The library must be compiled to a shared library object, which can be done with any C-compiler e.g. GCC. Once the shared library object is created it can be accessed in Python using the `import` statement. Each method from the shared library, to be available in Python, must be defined as a `PyObject*`, as shown in the example.

Integrating the reference model in the testbench was done by creating the module "REF model handler", see Figure 2.2. This handler is a UVM component that handles the transactions to and from the reference model through the shared library, the "REF wrapper". The handler is also a subscriber of the SDT agent, such that it can send identical sequence items to the reference model as sent by the drivers to the DUT. When the handler receives the processed items from the reference model, then the items are sent through an analysis port to the scoreboard for comparison.

```c
#include <Python.h>
#include "ref_model.h"

//Send an item from master to reference model
static PyObject* add_item_wrapper(PyObject* self, PyObject* args){
    PyObject* item;
    int queue;

    //Parse args as a tuple of Object and int (Oi)
    if (!PyArg_ParseTuple(args, "Oi", &item, &queue)){
        return NULL;
    }

    // Access the fields of the Python object
    PyObject* access_obj    = PyObject_GetAttrString(item, "access");
    PyObject* addr_obj      = PyObject_GetAttrString(item, "addr");
    PyObject* data_obj      = PyObject_GetAttrString(item, "data");

    // Extract the values from the Python objects
    int access  = PyLong_AsLong(access_obj);
    int addr    = PyLong_AsLong(addr_obj);
    int data    = PyLong_AsLong(data_obj);

    //Call reference model
    add_item(access, addr, data, queue);

     // Clean up
    Py_DECREF(access_obj);
    Py_DECREF(addr_obj);
    Py_DECREF(data_obj);

    Py_RETURN_NONE;
}
```

Listing 2.1: "REF wrapper" library. Using the Python-C interface to call a function from a C library

2.6 Register model

The dynamic priority of the clients connected to the memory arbiter is dependent on writing to registers, which requires a register model. The Register Abstraction Layer (RAL) in UVM facilitates the implementation of register models for the testbench. At the point of development of the current testbench, the RAL available in *PyUVM* was not fully implemented. To overcome the issue, it was created a new *PyUVM* base classes for the RAL, and the existing base classes were extended with the required functionality. The existing implementation was extended such that simple front-door access was possible but without implementing the full functionality. The implementation was written in the same structure as the source code for the SystemVerilog UVM implementation.

A working register model was created by extending the base classes and writing the register model containing two registers, the dynamic priority register (DPrio Register) and the control register, with respectively one and two fields, as well as a register map. To use the register model in the testbench an adapter was needed for converting between register items and bus items. As the register uses the APB protocol, the adapter was responsible for converting APB items to and from register items. Integrating the register model in the testbench facilitated reads and writes to the registers in the memory arbiter thus dynamically changing the priority of the arbiter.

2.7 Testbench tests

The CRV methodology implemented for the verification of the DUT, generates random constrained stimuli to be sent to the DUT. A base test has been created by extending the `uvm_test`. Its purpose is to configure all the components and properly connect them. To evaluate and check different behaviors, three random constrained tests have been derived from the base test. The goal of the first test was

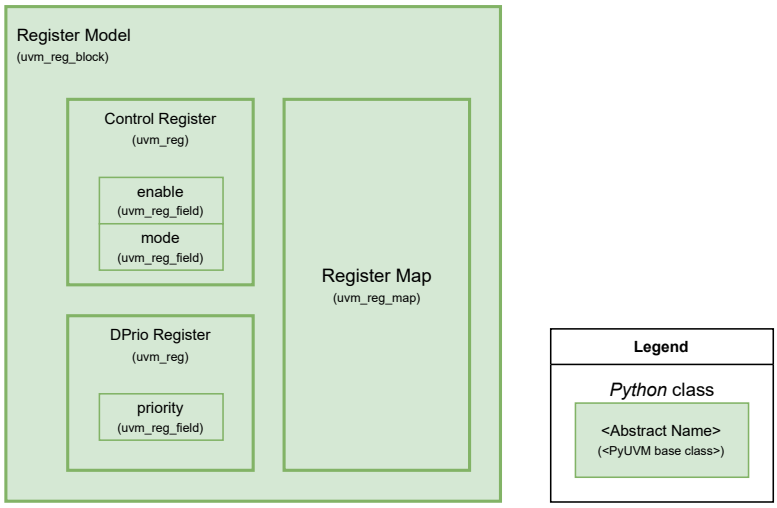

Figure 2.5: Block diagram of the register model

to validate the connections between the several components within the hierarchy. As a consequence, each master device connected to the DUT initialized a write and read sequence to random addresses within the memory. The desired randomization functionality was achieved using the *cocotb-coverage* library as will be explained in the next section. The second test evaluates the influence of the static priorities within the execution of different sequences initiated by different master devices. Similarly, this test was implemented using the *cocotb-coverage* library. The third test evaluates the dynamic prioritization of the arbiter by invoking different possible scenarios established by the priority and scheduling policies. In the case of this test, the *PyVSC* library was used due to the capabilities offered for randomizing dynamic objects.

2.8 Randomization

An essential part of all hardware verification is CRV. Using this methodology, the exploration of the state space is much more efficient from the perspective of time, as well as the code reusability. When conducting verification, it is crucial to narrow down the inputs to a specific subset of all possible inputs. This allows for guided testing and helps identify any holes in coverage.

In the testbench the generation of random stimuli was primarily applied at the SDT protocol level. It is desirable that both the sequences and sequence items should be randomizable to achieve efficient generation of random tests. The SDT sequence item is composed of three randomizable members: type of operation (read or write), address, data. These allow to read or modify the content of any valid address.

The randomization scheme used for both sequence items and sequences was implemented with the help of the *cocotb-coverage* [**?**], and it is used in all random constrained tests that were previously mentioned. The random sequences require hierarchical randomization, meaning that when the sequences are randomized so are the sequence items it contains. Since there are no built-in capabilities for randomizing nested objects, it is up to the user to implement it. A solution to overcome this problem will be discussed further in Section 3.1.

Another randomization requirement is represented by the procedure in which memory access is granted by the memory arbiter using the dynamic priority registers. Thus, to change dynamically the priority of each master device connected to the DUT, it was necessary to define a list of all valid permutations. For the case $n = 3$, the priority selection process is given by choosing one of the 6 permutations.

In addition to the priority mechanism, a scheduling mechanism was also defined and implemented. Thus, the sequences of two or multiple different clients can be started in parallel (the order of memory access is established by priority levels) or sequentially (the order is given by the scheduling levels). The

randomization scheme of this mechanism involves the definition of a list of variable lengths between 1 and the maximum number of connected devices, and the content of the list represents the scheduling order associated with each device (which must be unique values in the range $[0 : nrdevice - 1]$). The randomization schemes for priority and scheduling policies were implemented using the *PyVSC* [**?**] library, because it offers support in the randomization of dynamic objects, and was primarily used in the third test.

2.9 Functional coverage

Functional coverage is an important measure when evaluating the quality of the verification process. In the context of the memory arbiter testbench, two different tools (*cocotb-coverage* and *PyVSC*) were utilized to gather coverage data. Within the SDT UVC, a coverage component was implemented to determine the functional coverage. A covergroup for the transaction item has been created to collect data for the operation type as well as data and address values. It is composed of three coverpoints (one for each member of the transaction). Addresses and data are sampled into four equally sized bins, each covering one-quarter of the address or data range. Additionally, a cover cross was created by combining the access type, address, and data coverpoints. This ensures that all types of transactions have been exercised. A second covergroup was defined to collect information about the acknowledgment delay used by slave devices.

Regarding the memory arbiter, the current priority configuration and active requests were sampled. A cover cross was then defined using these two coverpoints. This comprehensive approach ensures that all possible combinations of priority configurations and input toggles have been tested, providing high confidence that the module operates as expected.

Besides *PyUVM* other Python libraries were needed for developing the testbench, mainly to support constrained randomization and functional coverage. The need for integrating randomization and coverage within the testbench led to the investigation of the available libraries that will be described in the following sections. A further description and comparison of the coverage collection capabilities of *PyVSC* and *cocotb-coverage* is presented in Section 4.

3 Constrained randomization

To use CRV in *PyUVM* and *cocotb*-based testbenches, additional Python plugins provide the means for performing randomization. The libraries considered in this paper are *cocotb-coverage* [**?**], *PyVSC* [**?**] and *constrainedrandom* [**?**]. *cocotb-coverage* and *PyVSC* both support randomization and coverage collection features whereas *constrainedrandom* only supports randomization.

To perform a preliminary analysis of the functionality of each of the three libraries mentioned earlier, two classes have been defined: simple and complex. The main focus was on how the randomizable members can be defined, the options available to create, activate, and deactivate constraints, as well as randomization mechanisms for nested objects.

3.1 Using *cocotb-coverage* for randomization

The Python library *cocotb-coverage* facilitates constrained randomization like the one applied in SystemVerilog. Listing 3.2 shows a simple class with three randomizable fields and a constraint on their values.

These random members are declared by calling the `add_rand` method inherited from `crv.Randomized`, along with the domain over which the variables are defined. As shown, the domain may also consist of non-integers.

An object with random member variables can be randomized by calling the `randomize` method on the object. To add additional constraints, the function `randomize_with` may be used, similar to the `with` keyword in SystemVerilog. An example implementation is shown in Listing 3.2.

```python
from cocotb_coverage.crv import Randomized

class cocotb_coverage_simple(Randomized):
  def __init__(self):
    ...
    # Declare fields as randomizable
    self.add_rand("op", [ReadWriteEnum.RD, ReadWriteEnum.WR])
    self.add_rand("addr", list(range(256)))
    self.add_rand("data", list(range(256)))

    self.add_constraint(lambda addr, op: addr < 128 if op == ReadWriteEnum.RD else addr >
    ↪  128)
    self.add_constraint(lambda addr, data: addr != data)
```

Listing 3.2: Using *cocotb-coverage* to define and randomize

A limitation of *cocotb-coverage* is the requirement that the domain over which a random variable is defined must be given as a parameter to `add_rand`. In particular, this complicates hierarchical randomization where a random variable may itself be an object, potentially with nested random variables, as the domain cannot be explicitly given.

One option is to resolve this issue by overloading the `randomize` and `randomize_with` methods to manually call `randomize` on all members that are random objects, as shown in Listing 3.3. Here, the methods `randomize` and `randomize_with` both call `self.ccs.randomize()` after randomizing the current object. If multiple nested objects exist, these must be manually added to the list of objects in `randomize` and `randomize_with`. To solve the issue of passing additional constraints to nested objects, an additional method `randomize_ccs_with` is created. This randomizes the parent object and then randomizes the nested object with constraints passed in the method. While this approach works, it is limited by the fact that individual methods have to be created for each additional random variable that should accept additional constraints.

```python
class cocotb_coverage_complex(Randomized):
  def __init__(self):
    ...
    self.ccs = cocotb_coverage_simple()
    # Note no call to self.add_rand() for ccs

  def randomize(self):
    """Extending the randomize function to randomize hierarchy"""
    super().randomize()
    self.ccs.randomize()

  def randomize_with(self, *constraints):
    """Extending the randomize_with function to randomize hierarchy
    with inline constraints for this object"""
    super().randomize_with(*constraints)
    self.ccs.randomize()

  def randomize_ccs_with(self, *constraints):
    """Creating new randomize_with function to randomize hierarchy
    with inline constraints for nested item"""
    super().randomize()
    self.ccs.randomize_with(*constraints)
```

Listing 3.3: Using *cocotb-coverage* to randomize nested objects

An alternative is to first randomize the parent object, and then directly call `parent.child.randomize_with()` to randomize nested objects. However, this does not serve as an idiomatically similar way of randomizing objects as the `with` clause in SystemVerilog.

3.2 Using *PyVSC* for randomization

The library *PyVSC* also adds support for constrained randomization and coverage collection. The library relies heavily on Python decorators to add support for CRV. Python decorators are tools that allow the extension or to modify the behavior of a function or class. Instead of adding random variables with a method, objects with random variables must be annotated with the `@randobj` decorator. In addition, the library provides the user with different constructors for defining the random members (e.g. `vsc.rand_bit_t`). An implementation of the classes presented for *cocotb-coverage* is shown in Listing 3.4, now using *PyVSC*.

By using the datatypes `vsc.rand_bit_t`, randomizable bit-vectors of a certain length are specified. Constraints are defined through functions annotated with `@vsc.constraint`. No additional boilerplate is required to add support for randomizing nested objects as opposed to *cocotb-coverage*. This comes at the cost of using *PyVSC*s domain specific language (DSL) for describing constraints, requiring e.g. the use of `with vsc.if_then` and `with vsc.else_then` to express relatively simple conditional relationships.

```
@vsc.randobj
class vsc_simple():
  def __init__(self):
    self.op = vsc.rand_enum_t(ReadWriteEnum)
    self.addr = vsc.rand_bit_t(8) #8-bit signal
    self.data = vsc.rand_bit_t(8)

  @vsc.constraint
  def c_addr_op(self):
    with vsc.if_then(self.op == ReadWriteEnum.RD):
      self.addr < 128
    with vsc.else_then:
      self.addr > 128

  @vsc.constraint
  def c_addr_data_neq(self):
    self.addr != self.data


@vsc.randobj
class vsc_complex():
  def __init__(self):
    ...
    self.vs = vsc.rand_attr(vsc_simple())
```

Listing 3.4: Using *PyVSC* to randomize and add constraints

3.3 Using *constrainedrandom* for randomization

The Python library *constrainedrandom* uses a syntax that is similar to *cocotb-coverage*. Implementations of the simple and complex example classes are shown in Listing 3.5.

The object to be randomized must extend the class `RandObj`. Random variables are added to the object by using the method `add_rand_var` and passing the domain of the random variable.

It is possible to add nested random objects by passing a function as the domain of the random variable, the result of the function being a randomized object. This is done to support nested randomization of the `simple` member of the complex class. This allows for hierarchical randomization but does not support passing constraints to child objects. By adding extra functions to change the behavior of `init_conrand_simple` it is possible to change this behavior, but the implementation is not intuitive.

```
from constrainedrandom import RandObj

class conrand_simple(RandObj):
  def __init__(self):
    ...
    self.add_rand_var("op", domain=ReadWriteEnum)
    self.add_rand_var("addr", domain=range(255))
    self.add_rand_var("data", domain=range(255))

    # Constraint with lambda function
    self.add_constraint(lambda op,addr:
      addr < 128 if (op == ReadWriteEnum.RD) else addr > 128, ("op", "addr"))

  # Constraint with predefined function
  def addr_data_neq(addr, data):
    return addr != data
    self.add_constraint(addr_data_neq, ("addr", "data"))

class conrand_complex(RandObj):
  def __init__(self):
    ...
    # Workaround to support randomization of nested objects
    # Uses 'init_conrand_simple' as the function for creating simple object
    def init_conrand_simple():
      x = conrand_simple()
      x.randomize()
      return x
    self.simple = self.add_rand_var("simple", fn=init_conrand_simple)
```

Listing 3.5: Using *constrainedrandom* to randomize and add constraints

3.4 Comparison of randomization libraries

As was emphasized in the previous section, using CRV grants benefits by exploring state space and closing coverage holes faster. The first randomization mechanism analyzed in this project was represented by the *cocotb-coverage*, built on top of the *cocotb* framework. This library is a good choice for educational purposes, as well as for small testbenches where the randomization structure does not have a high degree of difficulty. However, due to its limitations such as the impossibility of randomizing dynamic objects, as well as the boilerplate required for randomizing nested objects, the switch to the *PyVSC* library was made. The latter is a much more complete solution than *cocotb-coverage*, being in this case the most suitable for the use case. Another analyzed library was represented by *constrainedrandom*, a library that offers flexibility in defining and adding constraints. However, the performance decreases drastically when the complexity increases (e.g. variable defined on a higher domain which will increase state-space). Unlike *cocotb-coverage* and *PyVSC*, the library does not offer coverage capabilities.

To compare the three libraries for constrained randomization, some tests have been performed to compare the runtime and the distributions of the random numbers generated by each library.

3.4.1 Randomization speed

Two tests were run to measure the speed at which the libraries perform constrained randomization. In the first test (named basic), a random object with 4 random variables `a, b, c, and d` was created, using the first three constraints shown in Figure 3.1. The domain over which these four random members were defined was increased gradually from 8 bits to 16 bits. In the second test (named complex) one additional constraint was added, namely constraint 4 in Figure 3.1. In this specific case, the solver requires more time to find a more restrictive solution than in the first case, because the number of constraints has been increased, and the last constraint added implies that 3 of the 4 random variables are in a certain relationship.

The time required to create and randomize 100 objects in the basic or complex scenario using

```
constraint 1: a < c
constraint 2: d % 2 = 0
constraint 3: b > 0
constraint 4: (c + d) % b = 0
```

Figure 3.1: Constraints used in the *basic* and *complex* tests

PyVSC and *constrainedrandom* libraries is shown in Figures 3.2 and 3.3.

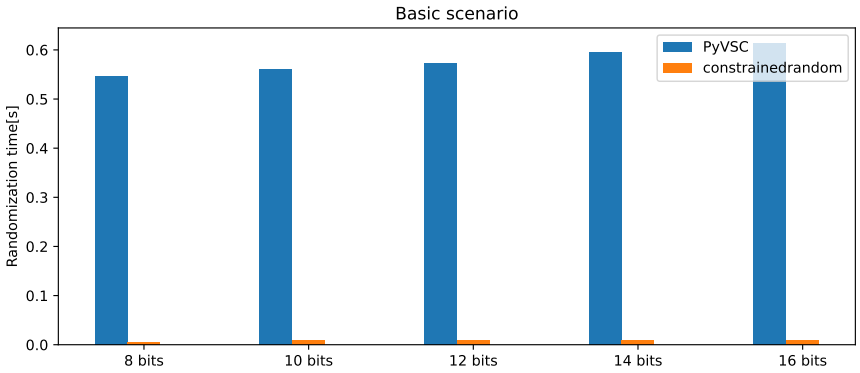

Figure 3.2: Runtime for the basic scenario with *PyVSC* and *constrainedrandom*

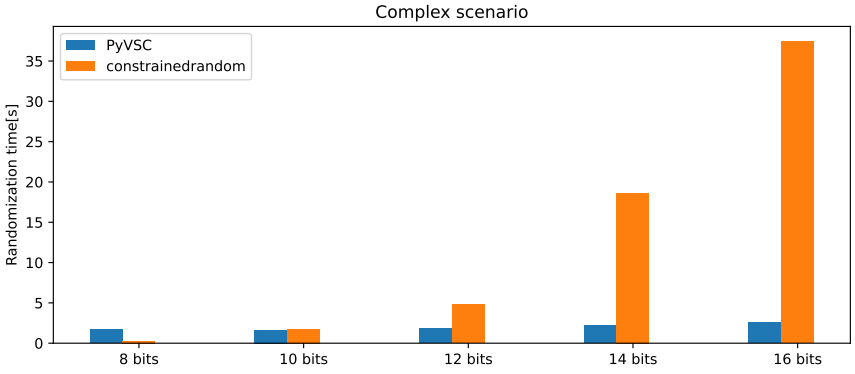

Figure 3.3: Runtime for the complex scenario with *PyVSC* and *constrainedrandom*

The results of the speed tests show that the *constrainedrandom* library is by far the fastest at resolving basic constraints, averaging a 70x speedup compared to *PyVSC*. However, the situation is reversed when trying to solve more complex constraints. As shown in Figure 3.3 the creation and randomization time of the 100 objects required by the *constrainedrandom* increases exponentially with the definition range of the randomized variables (from 0.26s for 8 bits to 37.42s for 16 bits). In the case of the *PyVSC* library, the variation is very small, taking an average of 2s to provide a solution.

3.4.2 Randomization quality

To compare the quality of random numbers generated, two separate tests were used. These tests have only been applied to *PyVSC* and *constrainedrandom* since the time required to create and randomize 10 objects in the basic (8 bits) for *cocotb-coverage* was 77s.

The first test creates 50000 elements with two random integer variables a, b that can take any value in the range `[0:500]`. These were then randomized with and without the further constraint

`a < b`. This was done to observe the distribution of values generated both with and without applying the constraint. The results of these tests are shown in Figures 3.4 and 3.5.

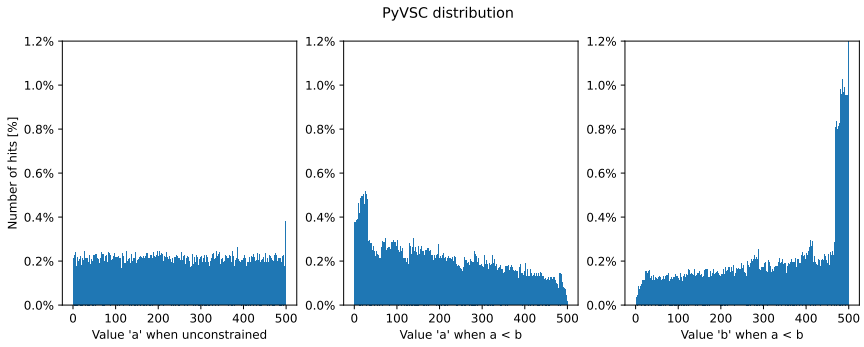

Figure 3.4: Distribution of random numbers generated when using *PyVSC*

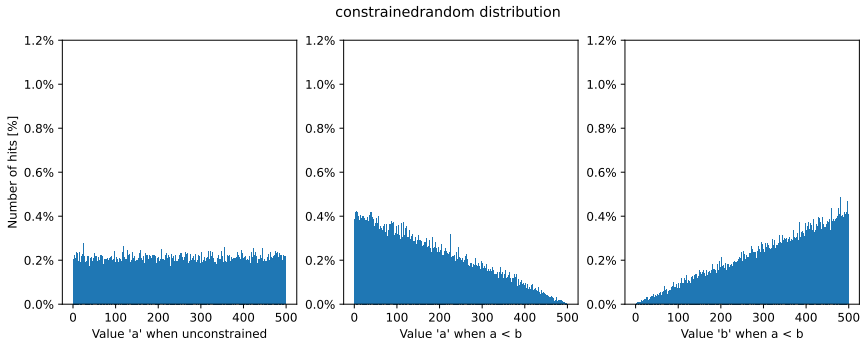

Figure 3.5: Distribution of random numbers generated when using *constrainedrandom*

From Figures 3.4 and 3.5 it is evident that the probability distribution of *PyVSC* is heavily skewed towards low values for a and high values for b when the constraint is activated. This is not the desired behavior and would indicate that *constrainedrandom* is the preferred tool to use.

The second test goes beyond the distribution plots shown and attempts to determine the type of solver implemented by the randomization libraries. The importance of the type of solver is illustrated by the following example: Consider a random variable with two randomizable members x and y, each defined on the interval $[0:9]$. The random variable has the single constraint $x \leq y$. A sequential solver will first solve for one of x or y, fixing that value and then solving for the next value. This means that not all combinations of x and y will have the same probability. If e.g. the value of x is fixed before the value of y, we find that the probability of a sequential solver hitting the value $(3, 5)$ is given by $P(y = 5|x = 3) = \frac{1}{10} \cdot \frac{1}{7} = 0.01428$. In practice, the values of the random members will be skewed towards the upper limit of their ranges. A generic example of this is shown in Table 1 where both values are defined on a domain of size $[n] = \{0, 1, \ldots, n-1\}$.

On the other hand, a parallel solver will solve for both x and y at the same time. This ultimately leads to each pair (x, y) being equally likely, which is usually the desired behavior. A generic example is presented in Table 2.

The tests were run by declaring a random variable with random members x and y, both taking values in the domain $[0:9]$ with the constraint that $x \leq y$. 10000 iterations were run on both *PyVSC* and *constrainedrandom*, and the occurence of each (x, y) combination were computed. The results of this are shown in Tables 3 and 4. The cells have been highlighted green if the number of hits into a specific bin was within $\pm 10\%$ of the expected value (1.64% and 2.00% in this case).

From the figures, it is obvious that *constrainedrandom* seems to be a parallel solver, whereas *PyVSC* does not seem to be a parallel solver with many bins outside the expected range. On the other hand, it is worth mentioning that the engine used does not look like a serial solver either.

Y \X	0	1	...	n-1
0	$\frac{1}{n} \cdot \frac{1}{n}$			
1	$\frac{1}{n} \cdot \frac{1}{n}$	$\frac{1}{n} \cdot \frac{1}{n-1}$		
⋮	⋮	⋮	⋱	
n-1	$\frac{1}{n} \cdot \frac{1}{n}$	$\frac{1}{n} \cdot \frac{1}{n-1}$	...	$\frac{1}{n} \cdot \frac{1}{1}$

Table 1: Generic table of probabilities assigned by a sequential solver when resolving $x \leq y$

Y \X	0	1	...	n-1
0	$\frac{2}{n \cdot (n+1)}$			
1	$\frac{2}{n \cdot (n+1)}$	$\frac{2}{n \cdot (n+1)}$		
⋮	⋮	⋮	⋱	
n-1	$\frac{2}{n \cdot (n+1)}$	$\frac{2}{n \cdot (n+1)}$	...	$\frac{2}{n \cdot (n+1)}$

Table 2: Generic table of probabilities assigned by a parallel solver when resolving $x \leq y$

Y \X	0	1	2	3	4	5	6	7	8	9
0	1.82									
1	1.71	1.93								
2	1.95	1.70	1.84							
3	1.84	1.74	1.89	1.91						
4	1.93	1.77	1.79	1.78	2.00					
5	1.64	1.83	1.94	1.87	1.71	1.76				
6	1.87	1.61	1.80	2.00	1.83	1.94	1.95			
7	1.53	1.84	1.75	1.67	1.89	1.74	1.71	1.72		
8	1.71	1.93	1.81	1.94	1.93	1.78	1.89	1.93	1.83	
9	1.44	1.83	1.81	1.78	1.94	1.88	1.42	2.07	2.10	1.78

Table 3: Percentage of generated pairs (x, y) that mapped to specific combination using *constrainedrandom*. Values within 0.9 and 1.1× the expected value are highlighted green, and values outside are highlighted red.

Y \X	0	1	2	3	4	5	6	7	8	9
0	2.55									
1	2.59	2.65								
2	2.08	3.00	2.43							
3	2.20	2.04	1.80	2.15						
4	1.47	2.48	1.79	1.94	2.00					
5	1.83	1.51	1.36	1.82	1.68	2.22				
6	1.90	1.90	1.13	1.63	1.50	2.10	2.33			
7	1.59	1.69	0.89	1.05	1.37	1.43	1.88	2.36		
8	1.09	1.63	1.19	1.25	1.39	1.55	1.56	1.77	3.54	
9	1.06	0.94	1.23	1.21	1.57	1.46	1.85	1.68	2.35	3.34

Table 4: Percentage of generated pairs (x, y) that mapped to specific combination using *PyVSC*. Values within 0.9 and 1.1× the expected value are highlighted green, and values outside are highlighted red.

4 Comparison of coverage collection tools

When verifying hardware designs, coverage collection is an important metric that guides further verification efforts. Functional coverage is a user-defined metric that refers to the features that were covered by an individual or a set of tests. It depends on the signal values sampled during the simulation and can be used to verify that interesting combinations of signals have been tested.

Both *cocotb-coverage* and *PyVSC* may be used to sample functional coverage in a Python-based testbench.

4.1 Using *cocotb-coverage* for coverage collection

cocotb-coverage does not operate with a concept of covergroups like in SystemVerilog. Instead, covergroups are implicitly defined, based on the hierarchical name of the coverpoints. Coverpoints are defined using the `coverage_section` construct, which creates a decorator that must be attached to another function. It may be a simple sampling function, or it may be attached to, e.g., a method in a monitor. Whenever the decorated method is called, coverage is automatically sampled.

```
import cocotb_coverage.coverage
import random

range_relation = lambda val_, bin_ : bin_[0] <= val_ <= bin_[1]
cov_groups = coverage_section (
  CoverPoint(
    name = "top.cg.cp_op",
    vname = "op",
    bins = ["RD", "WR"]
  ),
  CoverPoint(
    name = "top.cg.cp_addr",
    vname = "addr",
    bins = [(0, 127), (128, 255)],
    bins_labels = ["lo", "hi"],
    rel = range_relation
  ),
  CoverPoint(
    name = "top.cg.cp_data",
    vname = "data",
    bins = [(i,i+63) for i in range(0, 256, 64)],
    bins_labels = [f"rng[{i}]" for i in range(4)],
    rel = range_relation
  ),
  CoverCross("top.cg.X", items=[
    "top.cg.cp_op",
    "top.cg.cp_addr",
    "top.cg.cp_data"
  ])
)

@cov_groups
def sample_fun(op, addr, data):
  pass
```

Listing 4.6: Using *cocotb-coverage* to define a covergroup for the class shown in Listing 3.2

The `range_relation` is a function that defines how to interpret the ranges given as tuples. In the example depicted in Listing 4.6, it defines that the tuples denote an inclusive range of all values between the smaller and greater numbers. More complex relations may be used to, e.g., obtain coverage on transitions of a given signal.

4.2 Using *PyVSC* for coverage collection

PyVSC uses decorators to indicate which classes and methods are used to implement covergroups and coverpoints. An example of a simple covergroup is shown in Listing 4.7.

As shown, *PyVSC* uses a function `with_sample` to define the member variables corresponding to the sampled signals. The variables defined here are also added as members of the covergroup, and may be referenced in coverpoints. Once a covergroup is instantiated, sampling is performed by calling `sample` on it, passing the sampled signal values as parameters.

PyVSC supports dumping coverage data to a Unified Coverage Interoperability Standard (UCIS) file, allowing the coverage data to be read by other coverage analysis tools that support the UCIS format. It also provides a tool for visualizing coverage data. While it also provides a method for

```
import vsc

@vsc.covergroup
class vsc_covergroup():
  def __init__(self, name):
    self.options.name = name

    # Specifying parameters to the 'sample' method. Creates local fields
    self.with_sample(
      op = vsc.enum_t(ReadWriteEnum),
      addr = vsc.bit_t(8),
      data = vsc.bit_t(8)
    )

    # No explicit bins - automatically generated for all values
    self.cp_op = vsc.coverpoint(self.op)

    # Explicit bins, each covering half of the range
    self.cp_addr = vsc.coverpoint(self.addr, bins={
      "lo" : vsc.bin([0, 127]),
      "hi" : vsc.bin([128, 255])
    })

    # Bin array, creating 4 bins covering
    # an equal proportion of the sample space
    self.cp_data = vsc.coverpoint(self.addr, bins={
      "rng" : vsc.bin_array([4], [0, 255])
    })

    # Cover point cross
    self.cp_cross = vsc.cross([self.cp_op, self.cp_addr, self.cp_data])
```

Listing 4.7: Using *PyVSC* to define a covergroup for the class shown in Listing 3.4

merging multiple coverage reports, at the time of writing, this tool does not correctly merge coverage reports.

5 Discussion

The presented use case describes the implementation of a Python-based UVM testbench, using the *PyUVM* implementation of the UVM framework. While the presented design is non-trivial and has been successfully verified, it is not a large, complex design with many modules and interfaces. As such, this discussion cannot determine whether *PyUVM* and *cocotb* are sufficient for evaluating a design of a larger size and scope. It does, however, show that a Python-based testbench can be implemented, leveraging the UVM framework. In the end, the implementation of this project allowed us to generate some considerations that will be described below.

5.1 Python-based testbenches

The presented work uses some of the available open-source tools for the implementation of a testbench in Python used for design verification. The current state of the tools is mature enough to allow the implementation of a testbench in Python. However, there are still several limitations when implementing a Python-based testbench. These limitations result mainly as a consequence of the nature of an open-source project, which depends on the contribution of its community and does not follow a strict roadmap for the development or release of specific needed features.

The experience gained during the implementation of the presented use case allows us to state that using Python to implement direct tests as well as testbenches is quite straightforward, mainly because *cocotb* facilitates the interface with the HDL simulator, but also because of the simple syntax of Python and the several libraries available.

For writing further advanced testbenches *PyUVM* offers great support of the UVM framework in Python. As *PyUVM* is a Python implementation of the SystemVerilog UVM, the development of testbenches follows the same steps thus allowing the knowledge of UVM in SystemVerilog to be applied in Python as well. Furthermore, using Python offers some advantages such as no parameterization of classes is needed and the possibility of double inheritance for classes. As stated before, based on the current use case, *PyUVM* proved to allow the implementation of a Python-based testbench following the UVM framework.

It should also be noted the existence of another implementation of the UVM framework in Python. The *UVM-Python* library [**?**] is a direct translation of the SystemVerilog implementation to Python. However, it has not been evaluated in this project. *PyUVM* was chosen for the project since it seems to be more actively developed.

When using SystemVerilog and UVM for verification and implementing testbenches, some features are part of SystemVerilog and not UVM. Some of these features are not available using Python and *PyUVM* as they exist in between SystemVerilog and UVM, e.g. assertions and interfaces. To tackle the missing interfaces problem we instead presented a generic connection functionality that entirely decouples the ?? and the DUT. This is done by creating an interface class in Python and storing signal from the DUT available through *cocotb*. Assertions when implementing Python-based testbenches remain a limitation and will be discussed further in the next section.

The UVM factory is an important feature that improves the efficiency and flexibility of the UVM testbenches. UVM offers macros for the registration process of classes derived either from `uvm_component` or from `uvm_object`. The major benefit of this mechanism consists of replacing one object of a certain class with an object of a derived class. Similarly, *PyUVM* implements the same functionality for the factory overrides, which increases the reusability in the testbench as it is possible to create base tests and sequences.

5.2 Limitations

This section will describe the main limitations of the available open-source tools found during the implementation of the presented use case. Will also be explained how those may have influenced

the conclusions presented in the next section. Also, some ways to address these limitations in future verification projects.

Using Transaction-Level Modelling (TLM)

PyUVM has support for TLM 1.0 connections, supporting the use of TLM ports, exports and imp's. Because Python uses 'duck typing', there is no need for the large set of parameterized ports seen in the SystemVerilog implementation of UVM, making TLM connections simpler and less verbose when using *PyUVM* compared to their usage in SystemVerilog.

Unfortunately, *PyUVM* does not support TLM 2.0. That means that features such as the generic payload and bidirectional sockets are not part of the current *PyUVM* implementation. It does not appear that this is currently a feature in progress by the *PyUVM* development group.

Assertions

One area where open-source verification does not yet support the full feature set of SystemVerilog is in assertions. SystemVerilog supports both immediate assertions (using the values of signals 'right now') and concurrent assertions (clocked assertions, using current and previous signal values), but there is no way of realizing both types of assertions using open-source tools.

On one hand, it could be argued that the simulators should support the SystemVerilog constructs for assertions. Verilator has support for immediate assertions with the `assert` keyword as well as limited support for concurrent assertions with the `always @(posedge clock)` syntax. This is because the `##` and `|->` constructs are not supported. Likewise, Icarus Verilog has support for immediate assertions but does not support concurrent assertions.

On the other hand, it could be argued that *cocotb* could provide the support for immediate and concurrent assertions, given that it serves as the entry point to verification. It is possible to use the Python `assert` construct to perform immediate assertions in *cocotb* testbenches, but there is no syntax for expressing concurrent assertions. As such, the support for immediate and concurrent assertions using open-source tools is very limited.

Register Models and Register Abstraction Layer

Register models and RAL is an integral part of the verification of hardware, as registers are almost always present and often will contain a large number of registers. This is a clear limitation of open-source verification as *PyUVM* does not contain a finalized version of the RAL, which would be needed for testing of the hardware. Even with a finalized version of the *PyUVM* RAL implementation, there is still the need for autogenerated register models as available for SystemVerilog, as writing these by hand would be tedious and error-prone.

Simulation tools

Using *cocotb* and Python for verification can be done using both proprietary and open-source simulators. Using open-source simulators such as Icarus Verilog causes limitations as the simulator does not support all features of either Verilog or SystemVerilog. This might cause the need for rewriting of an otherwise compilable SystemVerilog DUT, to be able to simulate with one of the open-source simulators.

Coverage collection

Althought coverage reports can be generated when running a single simulation with both *PyUVM* and *cocotb-coverage*, the results when running multiple simulations are not as expected. The tool is not able to merge the coverage reports collected from the various runs, resulting in an incorrect result where the coverage shows the results from the last run only.

6 Conclusion

This paper presented the implementation of a verification methodology using *PyUVM*, an open-source library that allows the implementation of the UVM framework in Python. Reusability and scalability is a big challenge in testbench development for functional verification. A UVC was implemented for the SDT protocol using the *PyUVM* library, which was used to verify the presented use case, i.e. the memory arbiter. To address the DUT-testbench connectivity scalability challenge, a Python object was implemented to reproduce the functionality of a SystemVerilog interface. This object can also be adopted for the integration of other UVCs by simply passing the DUT port names to the object, creating a generic connection functionality that entirely decouples the UVC and the DUT. The other key aspects of verification such as register programming and reference model-based scoreboard were also demonstrated in this paper. A C-based reference model was implemented and integrated with the Python environment. This integration was relatively straightforward and demonstrated the possibility of reusing existing reference models with Python-based testbenches, instead of porting it to Python, which can reduce verification effort for complex DUTs. A RAL API was also implemented for register programming to present the RAL model concept using the existing support of *PyUVM* register model. Coverage model implementation enables one to do coverage collection and use those numbers to signoff verification. The coverage database was also analyzed with one of the open-source libraries used, to visualize the numbers properly.

This paper also presents a detailed comparison of randomization libraries such as *cocotb-coverage*, *PyVSC*, and *constrainedrandom*. This comparison provides the verifier with some performance benchmarks that can be used while selecting the suitable randomization library. Based on the comparison shown in Section 3.4 of the randomization modules, the *cocotb-coverage* has the worst performance of the three tools presented. Between the two remaining tools, *PyVSC* and *constrainedrandom*, the choice of a constrained randomization module depends on the use case. For non-complex constraints, *constrainedrandom* is much faster than *PyVSC*, whereas *PyVSC* is faster at solving complex constraints. This comes at the cost of *PyVSC* skewing the distribution of the generated random numbers, which might prolong verification efforts as some input combinations become less likely to be generated.

Future work could implement a UVM testbench using the presented libraries to verify a more complex design. The development of more efficient coverage analysis tools that allow merging the coverage database can be one big step towards open-source verification.

Contents

List of Figures

List of Listings

Glossary

SyoSil SyoSil ApS. 4

cocotb *COroutine based COsimulation TestBench.* Python library that implements a verification framework to connect a Python module with a HDL simulator. 2, 3, 7, 10, 13, 19, 20

cocotb-coverage Python library that can be used for functional coverage and constrained randomization. 1, 2, 9–14, 16, 17, 20, 21

constrainedrandom Python library for creating and solving constrained randomization problems. 1, 3, 10, 12–16, 21

Icarus Verilog Free and open-source compiler implementation for the IEEE-1364 Verilog HDL, also providing a simulation environment. 2, 20

PyUVM Python library that implements the UVM framework in Python. 1–4, 8, 10, 19–21

PyVSC Python library that implements random verification-stimulus generation and coverage collection. 1, 3, 9, 10, 12–18, 21

Acronyms

APB Advanced Peripheral Bus. 4, 6, 8

API Application Programming Interface. 21

CRV Constrained Random Verification. 2, 8–10, 12, 13

DSL Domain Specific Language. 12

DUT Device Under Test. 2, 4, 6–9, 19–21

HDL Hardware Description Languages. 2, 19

IC Integrated Circuits. 2

RAL Register Abstraction Layer. 8, 20, 21

SDT SyoSil Data Transfer Protocol. 4, 6, 7, 9, 10, 21

TLM Transaction-Level Modeling. 20

UCIS Unified Coverage Interoperability Standard. 17

UVC Universal Verification Component. 2, 4, 6, 7, 10, 21

UVM Universal Verification Methodology. 1–4, 6–8, 19–21

VHDL Very High-Speed Integrated Circuit Hardware Description Language. 2

VIP Verification IP. 2

References

[1] Stephen Williams, "Icarus Verilog." [Online]. Available: https://github.com/steveicarus/iverilog

[2] Verilator Development Team, "Verilator GitHub Repository." [Online]. Available: https://veripool.org/

[3] Cocotb Contributors, "cocotb." [Online]. Available: https://github.com/cocotb/cocotb/

[4] R. Salemi, *Python for RTL Verification: A complete course in Python, cocotb, and pyuvm.* Boston Light Press, 2022.

[5] A. B. Mehta, *ASIC/SoC Functional Design Verification.* Springer, 2018.

[6] R. Salemi, *The UVM Primer.* Boston Light Press, oct 2013.

[7] Python Software Foundation, "Python/C API Reference Manual." [Online]. Available: https://docs.python.org/3/c-api/index.html

[8] Marek Cieplucha, et. al, "cocotb-coverage." [Online]. Available: https://github.com/mciepluc/cocotb-coverage/

[9] Matthew Ballance, et. al, "PyVSC." [Online]. Available: https://github.com/fvutils/pyvsc

[10] Imagination Technologies, "constrainedrandom." [Online]. Available: https://github.com/imaginationtech/constrainedrandom

[11] Poikela, T. and others, "uvm-python: UVM 1.2 ported to Python," 2020. [Online]. Available: https://github.com/tpoikela/uvm-python