A Python based Design Verification Methodology

Ankitha[1], Dr. H. V. Ravish Aradhya* [2]
[1] *Student, R.V. College of Engineering, Bengaluru*
[2] *Professor & Associate PG Dean, R.V. College of Engineering, Bengaluru*
[1]*ankitha.ec17@rvce.edu.in,* [2] *ravisharadhya@rvce.edu.in*

Abstract: *While the UVM-constrained random and coverage-driven verification methodology revolutionized IP and unit-level testing, it falls short of SoC-level verification needs. A solution must extend from UVM and enable for vertical (IP to SoC) and horizontal (verification engine portability) reuse to completely handle SoC-level verification. To expedite test-case generation and use rapid verification engines, it must also provide a method to collect, distribute, and automatically amplify use cases. Opting a Python based Design Verification approach opens the door to various such merits. Cocotb is a very useful, growing methodology which can be used for the same.*

This paper elaborates on the application of cocotb, an open source framework hosted on Github which is based on CO-routine and CO-simulation of Testbench environment for verifying VHDL/Verilog RTL using Python. It employs equivalent design-reuse and functional verification concepts like UVM, however is implemented in Python, which is much simpler to understand and that leads to faster development and reduces the turn around time.

Keywords: *system-on-chip, design verification, universal verification methodology, intellectual property, design under test*

1. Introduction

Modern system-on-chip (SoC) designs have been evolving towards heterogeneous compositions of general purpose and specialized computing fabrics as Dennard scaling has ended and Moore's law has slowed. This heterogeneity makes the already difficult work of SoC design and verification much more difficult. Multiple generations of open-source hardware modelling frameworks have attempted to address the growing complexity of hardware design and verification. Comprehensive, productive, and open-source verification procedures that decrease the labour necessary to build completely validated hardware blocks are a critical missing component in the open-source hardware ecosystem.

Verification of open-source hardware has numerous substantial hurdles as compared to closed-source hardware. Closed source hardware, for starters, is typically owned and maintained by firms with specialized verification teams. These verification engineers often have a lot of expertise with constraint-based random testing using commercial SystemVerilog simulators utilizing a universal verification methodology (UVM). Open source hardware teams, on the other hand, typically use an agile test-driven design method borrowed from the open-source software community, in which the designer is also responsible for writing the tests. Furthermore, due to the high learning curve and limited support in existing open-source tools, open-source hardware teams seldom employ the UVM-based method. Instead of replicating closed-source hardware testing frameworks, the open-source hardware industry deliberately needs an alternate way for verifying open-source hardware.

The top-down approach offered by UVM does not work well for complex multimedia IP blocks like image signal processing pipeling, video codec, neural processing unit etc. due to the algorithmic/system architecture complexity. An SoC chain can contain

more than 20 blocks, which a verification testbench is expected to handle. There is a need for SoC DV to be able to take a portion of the IP DV environment and be able to re-run valid semi-randomized scenarios at SoC level. To fully address SoC-level verification, a solution must extend from UVM and allow for vertical (IP to SoC) reuse and horizontal (verification engine portability) reuse. A solution must provide a way to capture, share, and automatically amplify use cases to speed test-case creation and leverage fast verification engines.

2. Background

Design Verification is a process in which a design is compared against a given design specification before tape-out. This happens along with the development of the design and can start from the time the design architecture definition is completed. The main goal of verification is to ensure functional correctness of the design. However, with increasing design complexities, the scope of verification is also evolving to include much more than functionality. This includes verification of performance and power targets, security and safety aspects of design and complexities with multiple asynchronous clock domains. Simulation of the design model (RTL) remains the primary vehicle for verification while a lot of other methodologies like formal property verification, power-aware simulations, emulation/FPGA prototyping, static and dynamic checks, etc. are also used for efficiently verifying all aspects of design. The Verification process is considered very critical as part of design life cycle as any serious bugs in design not discovered before tape-out can lead to the need of newer steppings and increasing the overall cost of design process.

2.1. Functional Verification

The process of demonstrating the functional correctness of a design in relation to the design specifications is known as functional verification. Functional verification does not confirm the correctness of the design specification and instead assumes that it is correct. It is one of the most difficult steps in the IC design cycle and the primary cause of IC re-spin.
The main objectives are: Functional correctness of individual IPs, Internal module communication, External module communication, End to end functional paths, Clock and reset circuits, Power up and down sequence, Complete integration of all IPs.
Different types of Functional Verification methods are shown in Figure 1.

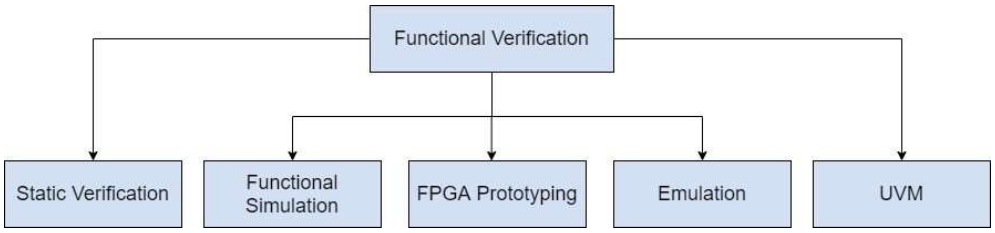

Figure 1. Types of Functional Verification

1) *Static Verification:* It is the process of checking a design against some predefined rules without running it. It enables validation of design at an early stage, without any stimulus or setup, and is thus performed early in the IC design cycle, that is, as soon as the RTL code is available. It doesn't do any timing checks. The earlier a bug is discovered, the easier it is to fix it. The goal of static verification is to decrease the verification effort at the RTL level.

2) *Functional Simulation:* The process of simulating a design's functional behaviour in software is known as functional simulation. It is not useful in software development because

it does not account for the timing delays of internal logic or interconnects. The goal of simulation is to validate the individual IPs or blocks of the IC. Functional simulation does not allow for system-level verification.

3) FPGA Prototyping: FPGA prototyping is the process of testing the functionality of an integrated circuit (IC) on FPGAs. With the increasing complexity of ICs and the increasing demand to reduce IC time to market, FPGA prototyping remains a critical solution. The goal of FPGA prototyping is to ensure that the design works as expected when driven with live data and that all of its external interfaces are operational.

4) Emulation: Emulation, also known as pre-silicon validation, is the process of testing the system's functionality on a hardware device known as an emulator. An emulator can handle both system-level and RTL designs (written in C, C++, or SystemC) (in Verilog or VHDL). Simulators take much longer to run than emulators. A design that takes days to simulate will only take hours to emulate. Emulation is used to find issues in system level design using live data, to verify system integration and to develop embedded software.

5) Universal Verification Methodology (UVM): UVM is a well-defined set of coding guidelines with a well-defined testbench structure. It's written in SystemVerilog and comes with a SystemVerilog base class library for creating advanced reusable verification components. It was created with significant guidance and input from Mentor by the Accellera Systems Initiative, an EDA standards body. IPs are extremely complex, and fully verifying them takes time. The standard test benches are not reusable, so verification engineers must build them from scratch. Due to time constraints, a verification methodology is highly recommended. UVM has a fixed testbench architecture, which makes the testbench highly reusable and saves time.

2.2. Switching to Python

SystemVerilog is a fairly complex programming language. The SystemVerilog specification is almost a thousand pages long. There are 221 keywords in the language, compared to 83 in C++. It's a powerful tool, but it takes some time to master. UVM has comparable concerns with complexity. There are numerous ways to accomplish the same task. Again, highly powerful, but difficult to master.

Ergo, SV-UVM is powerful but complicated. So hardware description languages are kept for designing whereas for developing testbenches, a high-level, general-purpose language with object oriented programming is considerably more beneficial. Thus, cocotb was created.

3. Design Verification using cocotb

Cocotb automatically connects to a variety of HDL simulators (such as Icarus, Modelsim, Questasim, and others) and allows you to control the signals in your design straight from Python. The whole testbench may be written in Python, and automation and randomization are simple to implement, resulting in increased productivity.

Cocotb does not necessitate the use of any additional RTL code. In the simulator, the top level is instantiated as the Design Under Test. Python is used to provide stimulation to the DUT's inputs and monitor the outputs. Given that it does not necessitate knowledge of HDLs, it can be of great help to those who are unfamiliar with it. Python is also an object-oriented scripting language. Cocotb has certain significant advantages over HDL testing techniques since it uses Python for verification:

- Python is an extremely productive language that allows one to write code quickly.

- Python makes it simple to connect to other languages.
- Python contains a large library of pre-existing code that can be reused.
- Python is an interpreted language, which means that tests can be modified and re-run without having to recompile the design or exit the simulator GUI.
- Python is widely used; significantly more engineers are familiar with it than SystemVerilog or VHDL.

3.1. Architecture of cocotb

A normal cocotb testbench does not necessitate any additional RTL code. Without any wrapper code, the Design Under Test (DUT) is instantiated as the simulator's toplevel. Cocotb applies stimuli to the DUT's inputs (or lower in the hierarchy) and monitors the outputs directly from Python. Cocotb acts as a bridge between the simulator and Python as shown in Figure 2 [9]. Verilog Procedural Interface (VPI) or VHDL Procedural Interface (VHDLPI) is used (VHPI).

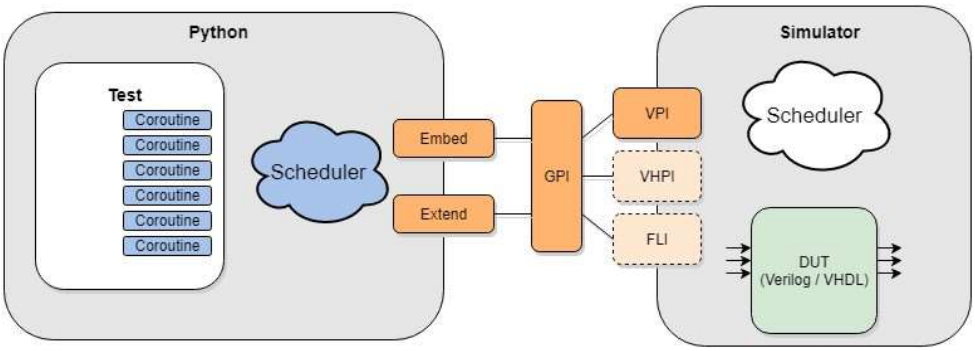

Figure 2. Architecture of cocotb

A test is merely a Python function. The await keyword indicates when control of execution should be returned to the simulator. A test can start numerous coroutines, permitting separate execution flows.

Python testbench code has the ability to:

- Traverse the DUT hierarchy and update values.
- Wait for the simulation timer to run out.
- Wait for a signal's rising or falling edge.

3.2. Design Methodology

The cocotb framework is made to be a goal-directed design verification tool. The following steps are included in the python based verification flow.

1) Capture the IP-level actions needed to create a desired use case, if not already captured.
2) Compose the desired use case in text format.
3) Use cocotb for vector generation:

cocotb allows constrained randomization through which all the parameters of the IP core can be randomized.

4) Verify the resulting vectors on a golden reference:

These vectors can be run on a C test design and the validity of vectors can be checked.

5) Review coverage results: Gcov and Lcov reports are then used to review the coverage results.

3.3. Cosimulation

It is the independent simulation of the design and testbench. Communication is accomplished using VPI/VHPI interfaces, which are represented by cocotb 'triggers'. The simulation time does not advance while the Python function is running. When a trigger is delivered, the testbench suspends execution until the triggered condition is met before restarting execution.

Some triggers availed are:

- Timer(time, unit): Waits for a given amount of simulation time to pass before acting.
- Edge(signal): Waits for a signal's state to change (rising or falling edge).
- RisingEdge(signal): Waits for a signal's rising edge.
- FallingEdge(signal): Waits for a signal's falling edge.
- ClockCycles(signal, num): Waits for a certain number of clocks to cycle (transitions from 0 to 1).

Sample RTL code of a 2:1 Multiplexer:

```
// example_mux.v
module example_mux( output wire out1, input wire in1, input wire
wire1, input wire wire2);
    // Switch between inputs depending on value of readout mode.
    assign out1 = in1 ? wire1 : wire2;
endmodule
```

Sample cocotb code for a 2:1 Multiplexer:

```
# mux_tester.py
import cocotb
from cocotb.triggers import Timer
from cocotb.result import TestFailure
@cocotb.test()
def mux_test(dut):
    dut.L0_i <= 0
    dut.we_lp_i <= 0
    dut.readout_mode_i <= 1
    dut.L0_i <= 1
    yield Timer(1, "ns")
    if dut.we_lp_muxed_o != 1:
        raise TestFailure("Failure!")
    dut.readout_mode_i <= 0
    yield Timer(1, "ns")
    if dut.we_lp_muxed_o != 0:
        raise TestFailure("Failure!")
```

The following are some critical points in the testbench code:

- The decorator *@cocotb.test()* declares a function as a test.
- The hierarchy is represented by the variable dut.
- The expression *dut.L0_i=0* is shorthand for assigning an RTL variable.
- Timer(1,'n') waits for the simulator to progress by 1 ns.
- If the MUX is not operational, *raise TestFailure* fails the test.

3.4. Modification of Hierarchy

As Python and RTL are co-simulated, it is simple to climb the hierarchy. Any internal signal may be read or changed by the Python testbench. It simplifies the modelling of single-event upset.

Example code below shows how value of internal signal could be read (and changed):

```
import cocotb
from cocotb.triggers import RisingEdge
@cocotb.test()
def test(dut):
    yield RisingEdge(dut.clk)
    # Accessing value of internal signal.
    current = int(dut.submodule.important.value)
    # Changing it.
    dut.submodule.important <= (not current)
    yield RisingEdge(dut.clk)
```

It is still allowed to have RTL testbench components. Creating a top-level Verilog or VHDL logic involves instantiating the actual design being tested, as well as other components for testing and use a trigger interface. However, it is not feasible to call operations directly but it is still helpful for low-level testing, assertions, and so on.

Cocotb can be used for post-synthesis simulations too. The wrapper approach can be used to load timing constraints (SDF) files on demand.

3.5. Coroutines

Cocotb employs a multitasking cooperative architecture. Tests, like regular Python, can invoke other methods and functions. Coroutines are required if such procedures are to use simulation time. Coroutines in cocotb are just functions that follow two rules:

1) The *@cocotb.coroutine* decorator is to be used.
2) Include at least one yield statement that results in another coroutine or trigger.

Sample code :

```
import cocotb
from cocotb.triggers import RisingEdge
@cocotb.coroutine
def test_helper(dut):
    dut.member <= 1
    yield RisingEdge(dut.clk)
@cocotb.test()
def test(dut):
    yield test_helper(dut)
```

3.6. Forking Coroutines

Coroutines can also be forked such that they run concurrently. This enables the development of something similar to a Verilog always block. Monitors can be started and run in the background to create sophisticated testbenches.

Sample code:

```
import cocotb
from cocotb.triggers import RisingEdge
@cocotb.coroutine
def always_block(dut):
    while True:
```

```
        yield RisingEdge(dut.clk)
        # Do something.
@cocotb.test()
def test(dut):
    # Start clock.
    thread = cocotb.fork(always_block(dut))
```

3.7. Joining Forked Coroutines

A forked coroutine, unlike always blocks, can be joined by calling *.join()*. It returns a trigger that wait until the coroutine completes its execution. It is also possible to kill a coroutine immediately by using the command *.kill()*.

Sample code:

```
import cocotb
from cocotb.triggers import RisingEdge, Timer
@cocotb.coroutine
def always_block(dut):
    while True:
        yield RisingEdge(dut.clk)
        # Do something.
@cocotb.test()
def test(dut):
    # Start clock.
    thread = cocotb.fork(always_block(dut))
    yield thread.join()
```

3.8. Communication with Coroutines

It is vital to communicate information across forked coroutines when developing sophisticated testbenches. There are a few options for doing this:

1) Using Event() trigger: A coroutine can yield *event.wait()* to block until another coroutine calls *event.set()* Data can be passed between coroutines by setting *event.data*.

2) Using classes: Functions in classes can be made coroutines and forked. The class will be accessible from both the main and the forked coroutine.

3) Combining the above two techniques: This can create advanced testbench components like drivers and monitors.

Sample code:

```
import cocotb
from cocotb.triggers import RisingEdge, FallingEdge
class SimpleDriver:
    def __init__(self, dut):
        self.dut = dut
        self.value = 0
    @cocotb.coroutine
    def drive(self):
        while True:
            yield RisingEdge(self.dut.clk)
            self.dut.data <= self.value
@cocotb.test()
def test(dut):
    driver = SimpleDriver(dut)
    cocotb.fork(driver.drive())
    yield FallingEdge(dut.clk)
    driver.value = 1
```

This is a basic example of a cocotb driver with coroutines. It makes use of a Python class (*SimpleDriver*). When the drive function is activated, it sets a port on the DUT to *self.value* on each clock. This flag may then be set outside of the coroutine, from the test.

4. Coverage

Code Coverage testing determines how much code is tested. Code coverage is a metric that describes the extent to which the program's source code has been tested. It is given by the Eqn. 1:

$$Code\ Coverage = \frac{Number\ of\ lines\ of\ code\ executed}{Total\ number\ of\ lines\ of\ code} * 100\ \% \quad (1)$$

There are several coverage types, which are as follows:

4.1. Statement coverage/ Line coverage

Statement coverage, often known as line coverage, is the most simple to comprehend sort of coverage. Statement coverage measures how many statements/lines are covered in the simulation.

4.2. Block/ Segment coverage

The nature of the statement and block coverage seems to be similar. The distinction is that block coverage takes into account branching blocks of if/else, case branches, wait, while, for, and so on. The dead code(lines which never get executed) is revealed by analyzing block coverage.

4.3. Conditional coverage

Conditional coverage, also known as expression coverage, shows how variables or expressions in conditional statements are assessed. Only expressions using logical operators are taken into account. Conditional coverage is the ratio of number of cases checked to the total number of instances present.

4.4. Branch coverage

Branch coverage, also known as decision coverage, reports the true or false of conditions such as if-else, case, and ternary operator statements. Decision coverage for an 'if' statement will report if the 'if' statement is examined in both true and false instances, even if a 'else' statement does not exist.

4.5. Toggle coverage

It ensures how many times variables and nets are toggled (flipping between logic high and logic low).Toggle coverage is just the ratio of toggled nodes to total nodes.

4.6. Path coverage

Due to conditional statements such as if-else, a different path is generated in the design, diverting the flow of input to the specific path. Path coverage is regarded to be more comprehensive than branch coverage since it can detect flaws in the order of operations.

4.7. FSM coverage

As it works on the design's behavior, it is the most complex sort of code coverage. In a finite state machine, this evaluates how often states are visited, transited, and how many sequences are covered.

A coverpoint is a fundamental coverage unit in SystemVerilog. It has numerous bins, each of which may hold multiple values. Every coverpoint has a variable or signal connected with it. The coverpoint variable value is compared to each designated bin during the sampling event. If a match is found, the number of hits in the given bin is increased.
Covergroups, which are special class-like structures, arrange coverpoints. A single covergroup can have several instances, each of which can gather coverage on its own. A covergroup necessitates sampling, which is a logic event (e.g. a positive clock edge). By invoking the *sample()* function in the testbench, sampling can be called implicitly.
When a function with a coverage is called with cocotb-coverage, sampling is performed each time. A cocotb coroutine that monitors the sample signal must be constructed in order to give the exact same functionality.
An example of cocotb sampling is shown in the sample code below.

```
@cover_group_1
def sampling_function(...):
    #Function to sample the coverage of cover_group_1
    #Do something sampling_function(...) Implicit call of sampling
can be anywhere in the code
@cocotb.coroutine
def edge_sensitive_sampling():
    # process to observe the logical event that samples the
coverage while True:
    yield RisingEdge(en) sampling_function(...) #Implicit sampling
cocotb.fork(edge_sensitive_sampling) #Fork the process observing
the sampling event
```

5. Results and Discussions

The python based verification methodology discussed so far form a foundation to implement a complete verification environment for a DUT. The results obtained after verifying an IP core is briefed in this section.

5.1. Coverage Report

The Gcov report generated are run for an IP core and the table below is obtained from verification environment in the initial run of C test.

IP Name	Statements Executed	Statements missed	Code Coverage
IP1	1081	331	69.380%

The Gcov report shown in table below is obtained after optimization of verification environment by removing dead code.

IP Name	Statements Executed	Statements missed	Code Coverage
IP1	1081	135	87.511%

5.2. Comparison between UVM and cocotb

By providing an abstract modular method, cocotb based verification enables use case extraction and, via abstraction, makes reuse, sharing, and amplification of use cases simple. The IP cores of an SoC chain verified using standard UVM methodology was compared with that of the cocotb framework. Results showed significant improvement in the simulation time.

6. Conclusion

The python based verification methodology enables use case extraction and, via abstraction, makes reuse, sharing, and amplification of use cases simple. It is a solution which allows for vertical (IP to SoC) reuse and horizontal (verification engine portability) reuse. C tests are often used in the verification of electronic SoCs and subsystems nowadays. These tests are in addition to UVM's IP-level verification and are often written manually or using simple code generators and they trail significantly behind the UVM automation that has become prevalent in hardware functional verification. Manually developing C tests does not adequately handle the effort of test generation and maintenance, test reuse across subsystems and systems, and utilising these tests for future system derivatives. Furthermore, the complete flow of defining objectives, automating stimulus production, executing tests to satisfy the goals, and gathering the findings in a succinct and intuitive dashboard presents hurdles for effective system validation.

Cocotb does not necessitate the use of any additional RTL code. In the simulator, the top level is instantiated as the DUT. Python is used to provide stimulation to the DUT's inputs and monitor the outputs. Given that it does not necessitate knowledge of HDLs, it can be of great help to those who are unfamiliar with it.

The IP cores verified using standard UVM methodology versus cocotb framework shows significant improvement in the simulation time as the python based framework requires only the c-model to generate the vectors, in contrast to the UVM methodology which requires both RTL as well as c-model to verify the design.

REFERENCES

[1] S. Jiang, P. Pan, Y. Ou and C. Batten, "PyMTL3: A Python Framework for Open-Source Hardware Modeling, Generation, Simulation, and Verification," in IEEE Micro, vol. 40, no. 4, pp. 58-66, 1 July-Aug. 2020, doi: 10.1109/MM.2020.2997638.

[2] S. Jiang, Y. Ou, P. Pan, K. Cheng, Y. Zhang and C. Batten, "PyH2: Using PyMTL3 to Create Productive and Open-Source Hardware Testing Methodologies," in IEEE Design & Test, vol. 38, no. 2, pp. 53-61, April 2021, doi: 10.1109/MDAT.2020.3024144.

[3] C. Spear, "Systemverilog for verification, second edition: A guide to learning the testbench language features," 2008.

[4] F. Shahzad, "Pymote 2.0: Development of an Interactive Python Framework for Wireless Network Simulations," in IEEE Internet of Things Journal, vol. 3, no. 6, pp. 1182-1188, Dec. 2016, doi: 10.1109/JIOT.2016.2570220.

[5] D. Gowda, R. Segu and K. A. Gupta, "Development of a verification environment to capture the functional coverage of PUCCH features in 5G User Equipment Simulator," 2020 Third International Conference on Advances in Electronics, Computers and Communications (ICAECC), 2020, pp. 1-5, doi: 10.1109/ICAECC50550.2020.9339510.

[6] S. Ali et al., "Towards Pattern-Based Change Verification Framework for Cloud-Enabled Healthcare Component-Based," in IEEE Access, vol. 8, pp. 148007-148020, 2020, doi: 10.1109/ACCESS.2020.3014671.

[7] M. W. Anwar, M. Rashid, F. Azam, A. Naeem, M. Kashif and W. H. Butt, "A Unified Model-Based Framework for the Simplified Execution of Static and Dynamic Assertion-Based Verification," in IEEE Access, vol. 8, pp. 104407-104431, 2020, doi: 10.1109/ACCESS.2020.2999544.

[8] L. Masing, F. Lesniak and J. Becker, "A Hybrid Prototyping Framework in a Virtual Platform Centered Design and Verification Flow," in IEEE Embedded Systems Letters, vol. 13, no. 1, pp. 1-4, March 2021, doi: 10.1109/LES.2020.2995084.

[9] Cocotb's documentation, https://docs.cocotb.org/en/stable/

[10] Gcov—a Test Coverage Program, https://gcc.gnu.org/onlinedocs/gcc/Gcov.html