Evaluating the Usability of pyuvm with cocotb for UART Verification

Mihir Ashvinbhai Donga, MINRES Technologies GmbH,
Regensburg, Germany (*mihir@minres.com*)
Eyck Jentzsch, MINRES Technologies GmbH, Munich, Germany (*eyck@minres.com*)
Prof. Dr. Juergen Mottok, OTH Regensburg,
Regensburg, Germany (*juergen.mottok@oth-regensburg.de*)

Abstract— *As SoC and IP designs grow increasingly complex, the demand for flexible and maintainable verification methodologies continues to rise. While SystemVerilog UVM remains the industry-standard verification framework offering constrained-random testing, functional coverage, and scalable environments, its steep learning curve and limited interoperability with modern software tools present notable challenges. This study explores an alternative approach using Python-based verification integrated with commercial simulators and compares it against a traditional UVM methodology.*
We implement two functionally equivalent verification environments for a UART IP core with an APB interface: a conventional SystemVerilog UVM testbench, and a Python-based solution combining pyuvm's structured verification components with cocotb's co-simulation capabilities. Both environments utilize industry-standard licensed simulators, ensuring a fair and practical comparison. Our evaluation focuses on four key metrics: simulation performance, functional coverage efficiency, testbench maintainability, and component reusability across projects.
Generally, results shows that Python-based verification can effectively complement traditional UVM flows, especially in unit-level verification scenarios, where rapid development, dynamic stimulus generation, and seamless integration with modern software stacks offer significant advantages. Although UVM remains favorable for large-scale SoC verification due to performance and maturity, the Python-based approach presents a compelling alternative for targeted use cases in commercial verification environments.

Keywords—Python-based verification; cocotb; pyuvm; SystemVerilog UVM; unit-level verification; functional coverage; RTL verification; UART; IP verification; simulation performance; verification methodology

I. INTRODUCTION

Modern hardware verification faces increasing complexity as designs become more sophisticated, demanding methodologies that balance thoroughness with efficiency [1][2]. SystemVerilog with UVM has long served as the industry standard for ASIC/FPGA verification due to its structured and reusable methodology. Our experimental results, however, highlight a practical challenge: a conventional UVM testbench utilizing a static constrained-random approach plateaued at approximately 87% functional coverage. While it is possible to achieve full coverage in UVM by manually authoring a comprehensive set of directed tests, this process requires significant engineering effort and a priori knowledge of the design's corner cases.

In contrast, our implementation using cocotb and pyuvm - a Python-based verification flow - achieves 100% functional coverage through dynamic and adaptive test generation. Specifically, the Python-based environment could analyze intermediate coverage results and iteratively adjust the stimulus to target uncovered scenarios, rather than relying solely on static constraint solving. This suggests that Python-based verification, once considered primarily a prototyping tool, is maturing into a practical alternative for rigorous verification tasks [3].

To evaluate these approaches, we used a UART IP core integrated with an APB interface as the Design Under Test (DUT). This configuration is commonly used in embedded systems, where the APB interface provides a low-complexity mechanism for accessing peripheral registers. The DUT is ideal for comparative analysis due to its well-defined behavior and verification requirements. We compare two distinct verification methodologies:

- A traditional UVM testbench using constrained-random stimulus generation
- A Python-based approach leveraging cocotb's co-simulation and pyuvm's UVM-like structure

Our study yields three key findings:

1. **Functional Coverage Completeness**: SV UVM coverage plateaus at 87% regardless of sample size (200 to 12,800), while cocotb + pyuvm consistently reaches 100% through adaptive dynamic stimulus.
2. **Verification Efficiency**: Python's runtime flexibility allows testbenches to make smart, real-time decisions and adapt tests without recompilation. This leads to faster convergence toward coverage goals. At the same

time, manual effort is reduced because writing, running, and debugging tests is much simpler and faster than with traditional UVM [4][3].

3. **Performance Tradeoffs**: Although UVM delivers faster raw simulation times (e.g., 3.07 ms vs. 6.1 ms for 12,800 samples), cocotb + pyuvm demonstrates superior coverage.

These findings show that Python-based verification is not only viable but, in some cases, superior for targeted verification challenges like Exhaustive Configuration Testing, Coverage Closure, Integration with Software/Drivers, etc., especially for:

- Unit-level IP verification requiring exhaustive configuration coverage
- Complex behavior validation needing runtime stimulus adaptation
- Agile or rapid-development cycles that benefit from Python's flexibility

This work offers empirical insight into when and how cocotb + pyuvm can not only complement but, in some cases, outperform traditional UVM-based flows, particularly in terms of coverage scalability and verification adaptability. Although SystemVerilog UVM remains indispensable for large-scale, performance-driven SoC verification, our results demonstrate that Python-based verification can achieve higher functional coverage with less complexity and faster test development. This makes cocotb + pyuvm a strong alternative for unit-level IP verification, rapid prototyping, and agile design cycles aligned with modern development practices [5].

II. VERIFICATION METHODOLOGIES OVERVIEW

A. *SystemVerilog UVM*

SystemVerilog UVM (Universal Verification Methodology), developed by Accellera, forms the foundation of modern ASIC and FPGA verification [1]. It is particularly well-suited for verifying complex SoCs due to its hierarchical architecture and comprehensive feature set, including reusable components such as drivers, monitors, sequencers, and support for constrained-random testing and functional coverage [2][6]. However, UVM's reliance on object-oriented SystemVerilog and its strict methodology introduces a steep learning curve [7]. Even simple testbenches often require extensive setup and boilerplate code. Furthermore, UVM typically depends on proprietary simulation tools due to the need for advanced debugging features, constrained-random support, and high-performance simulation engines. Commercial tools such as Mentor Questa, Cadence Xcelium, and Synopsys VCS provide these capabilities, whereas comparable open-source solutions remain limited. Despite these challenges, UVM remains a critical standard in production environments that demand robust coverage metrics, advanced debugging, and proven scalability.

B. *cocotb + pyuvm*

cocotb (Coroutine-based Co-simulation Testbench) and pyuvm represent a modern shift in hardware verification by leveraging Python's simplicity and expressive power [4]. These frameworks enable the development of complete verification environments using Python, offering a flexible alternative to traditional HDL-based methodologies such as SystemVerilog UVM [3][5].

cocotb allows testbenches to be written in Python using coroutines to drive and monitor simulations, removing the need for additional HDL-based stimulus generation. It supports both open-source and commercial simulators, including Verilator, Icarus Verilog, Mentor Questa, Cadence Xcelium, Synopsys VCS and ModelSim, making it widely accessible across different development environments. However, while cocotb excels in simplicity and rapid prototyping, it lacks the structured components such as drivers, monitors, agents, and scoreboards that are essential for building scalable and reusable testbenches in complex verification projects [3].

pyuvm extends cocotb's capabilities by introducing a UVM-inspired, object-oriented architecture within Python [5]. It provides key verification components such as `uvm_driver`, `uvm_monitor`, and `uvm_sequence`, enabling the construction of modular and reusable testbenches that align with the principles of UVM. pyuvm retains compatibility with cocotb's simulator interfaces, making it a natural progression for users seeking more structure without leaving the Python ecosystem [3][5].

The goals of pyuvm are to:

- Emulate UVM's hierarchical and modular design while preserving Python's simplicity and readability
- Enable structured, reusable verification environments without requiring SystemVerilog or proprietary tools

While pyuvm is still under active development and has limited adoption in commercial flows, it shows strong potential for academic research, prototyping, and unit-level IP verification especially where the overhead of full SystemVerilog UVM infrastructure is impractical [3]. Ultimately, pyuvm aims to bridge the gap between Python's flexibility and the disciplined structure of traditional verification methodologies [5].

III. UART VERIFICATION ENVIRONMENT SETUP

A. *Design Overview*

Figure 1 shows a top-level design that integrates an APB slave interface with a UART peripheral. It connects APB interface signals to the APB slave, which manages bus read/write operations and generates control signals for UART configuration and data transfer. The UART module handles serial communication, using signals from the APB slave to perform transmit and receive functions. Internal signals link the APB slave and UART blocks for data, address, and status exchange, enabling seamless communication between the APB interface and the UART device.

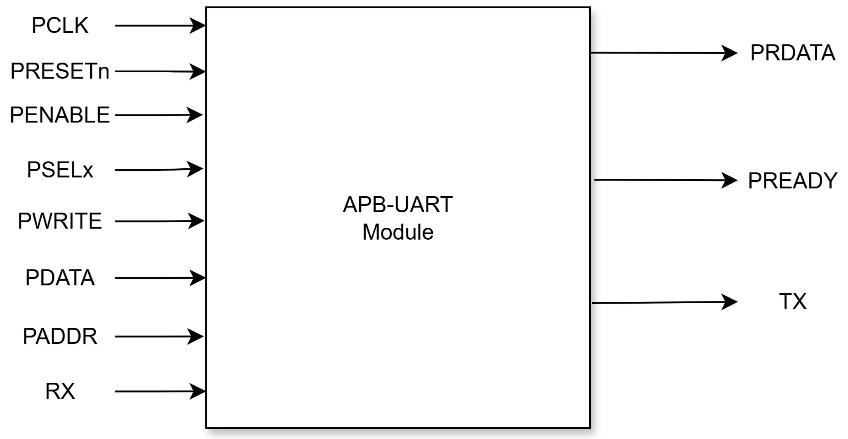

Figure 1: APB-UART Module

B. *SV-UVM Testbench Environment*

Figure 2 illustrates the traditional SystemVerilog UVM testbench architecture, resembling a detailed engineering blueprint. It clearly displays all verification components - including UART and APB agents with their monitors, drivers, and sequencers - connected in a precise, hierarchical structure [1]. This reflects SV-UVM's strength: a rigorously organized approach where every element has a defined place and connection before simulation begins [2][6]. The attention to component-level details makes it ideal for complex, large-scale verification projects [8].

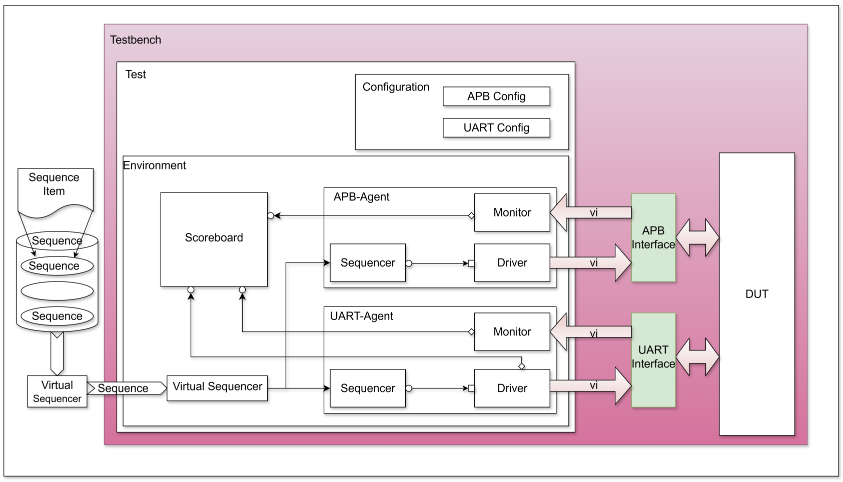

Figure 2: UVM-Based APB-UART Verification Environment

C. *cocotb + pyuvm Testbench Environment*

Figure 3 presents the cocotb/pyuvm approach with a same focus on simulation workflow and component hierarchy. It emphasizes Python's role in controlling the verification process through virtual sequences and scheduler interactions [3]. It better represents the dynamic, adaptable nature of Python-based verification. This streamlined view is particularly useful when quick iterations or runtime adjustments are needed during verification.

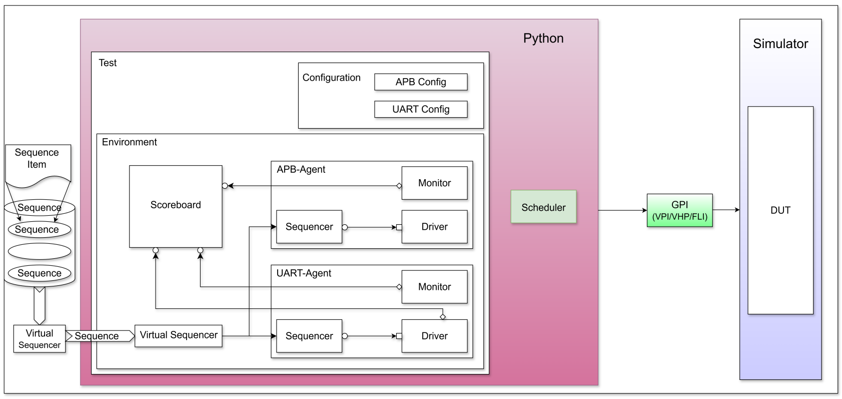

Figure 3: cocotb + pyuvm Based APB-UART Verification Environment

The key difference between Figure 2 and Figure 3 is not the internal testbench architecture, which remains essentially the same in both SystemVerilog UVM and cocotb/pyuvm, but rather how the testbench connects to the simulator and the DUT. In Figure 2, the testbench communicates with the DUT through SystemVerilog interfaces in a traditional HDL simulation environment. In contrast, Figure 3 shows how cocotb/pyuvm leverages the simulator's GPI (VPI/VHPI/FLI) interface and a Python scheduler to drive and monitor the DUT. This shift in integration enables Python-based verification to be more lightweight and accessible for rapid prototyping and debugging, while SystemVerilog UVM provides the maturity and robustness expected in industrial projects. Both approaches implement the same verification components, but they cater to different development and deployment needs.

D. *KEY IMPLEMENTATION DIFFERENCES*

pyuvm Based UART Agent

Listing 1: UART Agent

```python
class UARTAgent(uvm_agent):
    def build_phase(self):
        self.cfg = ConfigDB().get(None, "", "cfg", uart_config())
        self.monitor = UARTMonitor.create("monitor", self)
        if self.cfg.is_active:
            self.driver = UARTDriver.create("driver", self)
            self.sequencer = UARTSequencer.create("sequencer", self)

    def connect_phase(self):
        if self.cfg.is_active:
            self.driver.seq_item_port.connect(self.sequencer.seq_item_export)
```

As shown in Listing 1 demonstrates a modular agent implementation using pyuvm. It defines standard verification components driver, sequencer, and monitor and instantiates them during the build_phase based on a configuration object (uart_cfg) retrieved from the ConfigDB, a centralized database used to store and manage configuration settings across the testbench.. When the agent is active, the connect_phase links the sequencer and driver using the sequence item port. This structure mirrors the UVM methodology while leveraging Python's simplicity and readability for easier integration in lightweight or academic environments [3].

pyuvm UART Driver

Listing 2: UART Driver

```
class UARTDriver(uvm_driver):
    def build_phase(self):
        self.cfg = ConfigDB().get(None, "", "cfg", uart_config())
        self.dut = ConfigDB().get(None, "", "dut", cocotb.top)
        self.item_collected_port_drv = uvm_analysis_port("
            item_collected_port_drv", self)

    async def run_phase(self):
        while True:
            await RisingEdge(self.dut.PCLK)
            if not self.dut.PRESETn.value:
                continue
            req = await self.seq_item_port.get_next_item()
            self.cfg_settings()
            await self.drive_rx(req)
            self.item_collected_port_drv.write(req)
            self.seq_item_port.item_done()

    def cfg_settings(self):
        # Set line transmission cycles based on frame length
        pass

    async def drive_rx(self, req):
        # Simplified transmit logic for UART protocol
        for _ in range(self.LT):
            await RisingEdge(self.dut.PCLK)
            self.dut.RX.value = req.start_bit
            # (Send data bits, optional parity, and stop bits...)
```

Listing 2 illustrates a pyuvm implementation of a protocol-specific driver for UART verification. It extends the base uvm_driver class and manages stimulus generation by asynchronously fetching transactions from the sequencer in the run_phase. Configuration parameters are retrieved from ConfigDB during build_phase and guide signal driving behavior. The driver models UART transmission by toggling the RX line according to start bits, data payload, parity, and stop bits in the transaction. This Python-based coroutine approach enables fine-grained control of stimulus timing, leveraging cocotb triggers for cycle-accurate simulation in a more accessible, high-level language environment.

pyuvm UART Monitor

Listing 3: UART Monitor

```
class UARTMonitor(uvm_monitor):
    def build_phase(self):
        self.cfg = ConfigDB().get(None, "", "cfg", uart_config())
        self.dut = ConfigDB().get(self, "", "dut", cocotb.top)
        self.item_collected_port_mon = uvm_analysis_port("
            item_collected_port_mon", self)

    async def run_phase(self):
        while True:
            await RisingEdge(self.dut.PCLK)
            # Wait for start bit (Tx goes low)
            while int(self.dut.Tx.value) == 1:
                await RisingEdge(self.dut.PCLK)
            # Sample bits (simplified)
            reg = 0
            for _ in range(self.cfg.frame_len):
                # Wait one bit period (simplified)
                await Timer(int(1e9 / self.cfg.bRate), units='ns')
                reg = (reg >> 1) | (int(self.dut.Tx.value) << (self.cfg.
                    frame_len - 1))
            # Send captured data transaction
            txn = UARTTransaction()
```

```
                txn.transmitter_reg = reg
                self.item_collected_port_mon.write(txn)
```

Listing 3 observes serial transmission from the UART DUT and reconstructs received data frames [3]. In the run_phase, the monitor detects the start bit, samples data bits based on the configured baud rate, optionally accounts for parity, and verifies stop bits. It calculates the frame content and publishes each received transaction via an analysis port, enabling further checking or scoreboard comparison [3]. This monitor adheres to pyuvm's passive component model, focusing on protocol observability without interfering with DUT operation.

pyuvm UART Transactions and Sequences

Listing 4: UART Transaction

```
class UARTTransaction(uvm_sequence_item):
    def __init__(self):
        super().__init__()
        self.payload = vsc.rand_uint32_t()
        self.bad_parity = vsc.rand_bit_t()
        self.bad_parity_frame = vsc.rand_bit_t(7)
        self.payld_func = 0

    def calc_parity(self, frame_len, even_parity):
        self.payld_func = self.payload & 0xFFFFFFFF
        parity_result = 0
        for i in range(7):
            chunk = (self.payld_func >> (i * frame_len)) & ((1 << frame_len) -
                1)
            parity = self._calc_single_parity(chunk, even_parity)
            if self.bad_parity and ((self.bad_parity_frame >> i) & 1):
                parity = not parity
            parity_result |= (parity << i)
        return parity_result & 0x7F

    def _calc_single_parity(self, chunk, even_parity):
        bits_set = bin(chunk).count('1')
        return (bits_set % 2) != even_parity
```

This Listing 4 defines the UARTTransaction class as a customizable data container for UART stimulus and response, extending uvm_sequence_item [3]. Several sequence classes generate test scenarios targeting specific UART error conditions such as stop bit corruption, frame errors, and parity errors operation [3][5]. These sequences demonstrate pyuvm's support for asynchronous stimulus generation using Python's async/await syntax, enabling clear and flexible test case creation [3].

APBUART Sequencer

The apb_sequencer class extends the base uvm_sequencer component in pyuvm. It serves as the transaction generator and driver interface, sequencing APB transactions during verification.

APBUART Scoreboard

Listing 5: APBUART scoreboard

```
from pyuvm import *

class APBUARTScoreboard(uvm_scoreboard):
    def __init__(self, name, parent):
        super().__init__(name, parent)
        self.pkt_queue = []
        self.coverage = SomeCoverageClass()  # Replace with your coverage class
            instance

    def write(self, pkt):
        self.pkt_queue.append(pkt)
        # Sample coverage with relevant transaction fields
        self.coverage.sample(pkt.some_field)

    def report_phase(self):
        cov = self.coverage.get_coverage()
        self.logger.info(f"Coverage: {cov:.2f}%")
```

Listing 5 implements a uvm_scoreboard for verifying APB-UART integration. It subscribes to analysis ports from APB and UART monitors and drivers, storing received transactions in internal queues. The scoreboard compares APB configuration register accesses and data transactions against UART transactions to ensure correct operation. It validates configuration register values, transmit and receive data integrity, and checks for correct error signaling. Additionally, the scoreboard uses coverage components to track functional coverage of configuration, transmission, and reception, reporting the coverage results at the end of the simulation [5].

APBUART Environment

Listing 6: APBUART environment

```
class APBUARTEnv(uvm_env):
    def build_phase(self):
        super().build_phase()
        self.apb_agnt = APBAgent("apb_agnt", self)
        self.uart_agnt = UARTAgent("uart_agnt", self)
        self.scoreboard = APBUARTScoreboard("scoreboard", self)

    def connect_phase(self):
        super().connect_phase()
        self.apb_agnt.monitor.item_collected_port_mon.connect(
            self.scoreboard.item_collected_export_monapb)
        self.uart_agnt.monitor.item_collected_port_mon.connect(
            self.scoreboard.item_collected_export_monuart)
        self.uart_agnt.driver.item_collected_port_drv.connect(
            self.scoreboard.item_collected_export_drvuart)
```

Listing 6 instantiates and connects all key UVM components for the APB-UART testbench, including APB and UART agents, a combined scoreboard, and a virtual sequencer. In the build phase, it creates these components, while in the connect phase it links the agents' analysis ports to the scoreboard's analysis exports to facilitate transaction collection and checking. It also sets sequencers in the UVM configuration database for virtual sequencer access.

APBUART config_test

Listing 7: APBUART configuration test

```
class apbuart_config_test(apbuart_base_test):
    def build_phase(self):
        super().build_phase()
        self.apbuart_config_sq = apbuart_config_seq.create("apbuart_config_seq"
            )

    async def run_phase(self):
        self.raise_objection()
        for i in range(self.cfg.loop_time):
            self.set_config_params(9600, 8, 3, 1, 1)  # last arg=1 means
                randomize
            self.logger.info(f"Iteration {i+1}: UART Config:\n{self.cfg}")
            self.set_apbconfig_params(2, 1)  # last arg=1 means randomize APB
                config
            self.logger.info(f"Iteration {i+1}: APB Config:\n{self.apb_cfg}")
            await self.apbuart_config_sq.start(self.env_sq.v_sqr)
        self.drop_objection()
        await Timer(20, "ns")
```

This Listing 7 inherits from `apbuart_base_test`, where the APBUART environment (`APBUARTEnv`) is instantiated. In the build_phase, it creates the configuration sequence. During the run_phase, it runs multiple iterations where UART and APB configurations are randomized and applied. Each iteration starts the configuration sequence on the virtual sequencer, verifying the APB-UART interface behavior under varying configurations. The test manages UVM objections to control simulation flow and includes logging for configuration details.

cocotb Top-Level Testbench

Listing 8: Top-level cocotb testbench for APBUART

```
@cocotb.test()
async def tbench_top(dut):
    # Start 50 MHz clock
    # Apply reset (active low)
    ConfigDB().set(None, "*", "dut", dut)
    await uvm_root().run_test(apbuart_config_test)
```

This Listing 8 initializes the DUT clock and reset signals using cocotb coroutines, setting a 50 MHz clock and an active-low reset pulse. It leverages ConfigDB to store references to DUT signals and buses, enabling access to these signals from pyuvm components such as drivers, monitors, and sequencers without explicit interface wrappers. The UVM test starts by invoking uvm_root().run_test() asynchronously, integrating cocotb's coroutine scheduler with the UVM test execution.

IV. EXPERIMENT EVALUATION

A. *Technical Comparison: SV-UVM vs. cocotb + pyuvm*

Category	**SV-UVM**	**Cocotb+PyUVM**	**Technical Implications**
Language Foundation	SystemVerilog (IEEE 1800) + UVM Class Library (IEEE 1800.2)	Python 3.6+ + Cocotb (co-simulation) + PyUVM (UVM-like Python library)	SV-UVM is strongly typed; PyUVM relies on dynamic typing.
Simulation Interface	Direct integration with HDL simulators via PLI/VHPI	Socket-based communication with simulators (VPI/FLI)	SV-UVM has lower latency; PyUVM adds Python interpreter overhead (2–5× slower).
Importation	`import uvm_pkg::*`	`from pyuvm import *`	All UVM classes/functions become available without referencing the package.
Concurrency Model	UVM phases (build_phase, run_phase) with fork-join	Python coroutines (async/await) with event loop	PyUVM allows true parallelism; SV-UVM requires careful thread management.
Debugging	Waveform-based (Verdi/DVE) + UVM reporting (`uvm_info`, `uvm_error`)	Python debuggers (pdb) + logging + limited waveform access	SV-UVM enables signal-level debug; PyUVM excels at testbench logic inspection.
Randomization	Native `rand`/constraint with SV solver	PyVCS constraint solver (Python-based CSP solver)	PyVCS provides SV-like constraints with Python syntax (e.g., `@constraint` decorators).
Configuration	`uvm_config_db` with static type checking	`ConfigDB()` dictionary with dynamic types	SV-UVM catches config errors at compile time; PyUVM may need runtime checks.
Coverage	Built-in covergroup, coverpoint, cross	PyVCS coverage collectors	PyVCS enables SV-style coverage with Python flexibility (e.g., dynamic bins).
TLM Ports	`uvm_{*}_port`, `uvm_{*}_export` with strict compliance	Python method calls + decorators	SV-UVM enforces protocol compliance; PyUVM is flexible but prone to errors.
Clock Domain Handling	Native clocking blocks (`@(posedge clk)`)	Cocotb triggers (`await RisingEdge(dut.clk)`)	SV-UVM has precise timing control; PyUVM depends on simulator synchronization.

Table 1: Technical Comparison: SV-UVM vs. Cocotb+PyUVM

B. Coverage Metrices

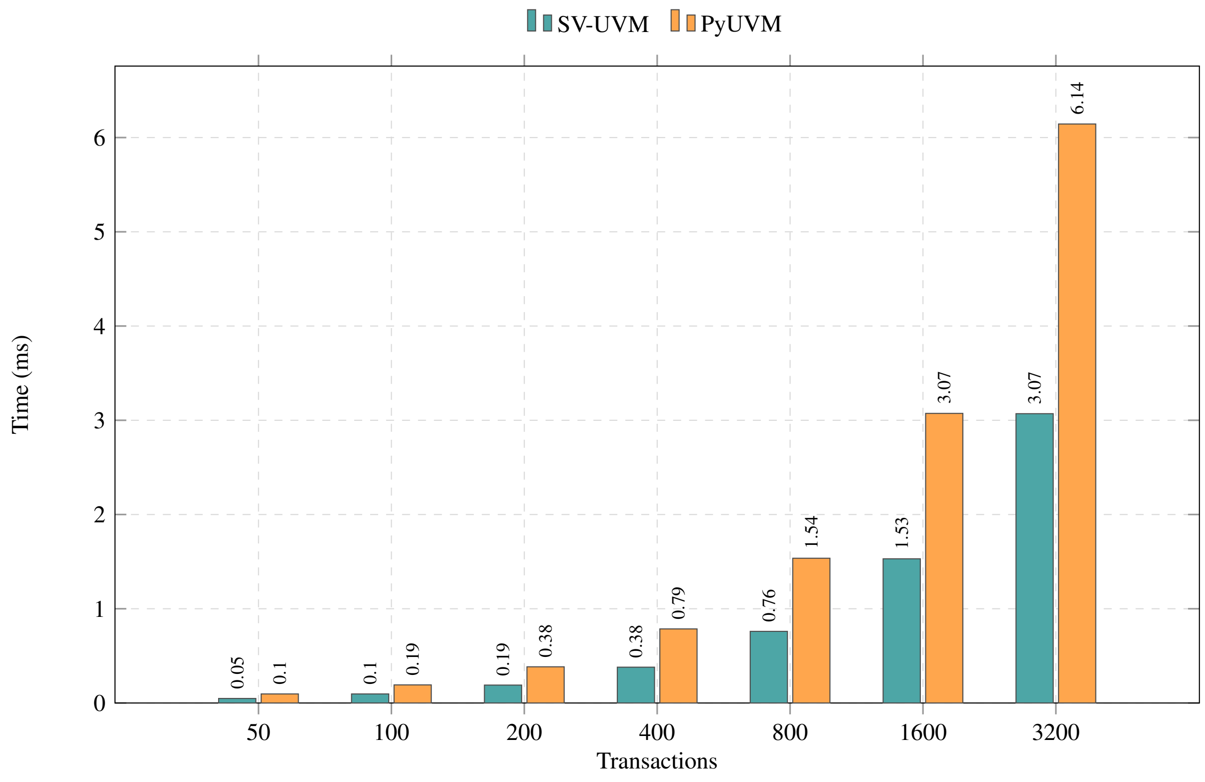

Figure 4: Simulation time v/s number of transactions.

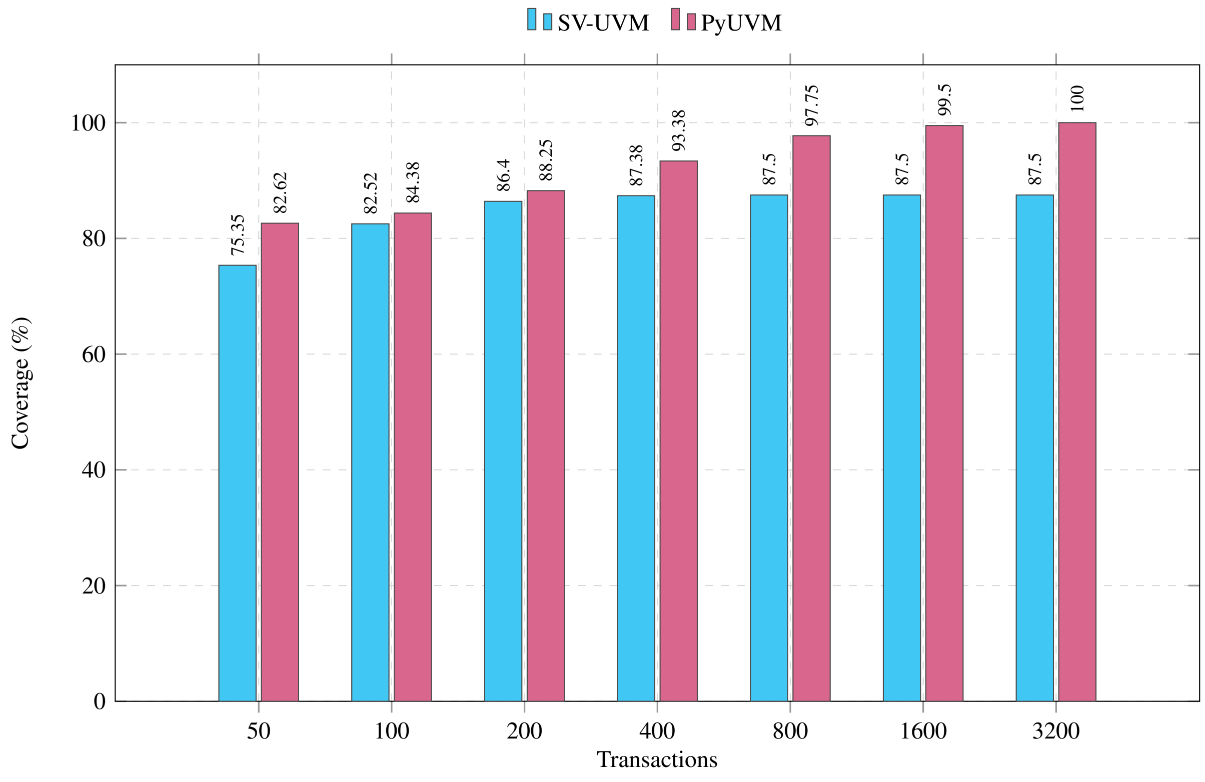

Figure 5: Functional coverage v/s number of transactions.

Figure 4 & Figure 5 presents a comparison of configuration coverage, simulation time, and transaction count between two verification methodologies: SystemVerilog UVM and cocotb with pyuvm. The data reflects results from a configuration-driven test targeting UART features, including baud rate, frame length, parity bit, and stop bit [7].

The SV UVM method shows consistently faster simulation times as the number of transactions increases, but the coverage plateaus at 87.5%, even with larger sample sizes. On the other hand, cocotb + pyuvm achieves higher configuration coverage, reaching 100%, though with increased simulation durations [2].

At lower transaction counts (50–400), both methods perform similarly in terms of coverage growth, but cocotb + pyuvm quickly surpasses SV UVM as the sample size increases. This comparison highlights a tradeoff between the speed efficiency of SV UVM and the coverage completeness of cocotb + pyuvm, allowing users to choose based on verification goals whether prioritizing faster execution or thorough configuration space exploration.

V. CONCLUSION

This study shows that Python-based verification using cocotb and pyuvm is a viable alternative to SystemVerilog UVM for unit-level IP verification. Using a UART IP core as a case study, we compared both approaches in terms of coverage, development effort, and integration. While UVM offers faster simulation and mature tool support, cocotb + pyuvm achieved 100% coverage with simpler, more flexible testbenches. Python's ease of use, quick development, and integration with modern software tools make it especially useful for academic, open-source, and agile environments. These results provide practical guidance on when Python-based verification can complement or replace traditional UVM.

References

[1] U. B V and K. Muchalambi, "Design and verification of ddr5 subsystem using uvm methodology," pp. 1–6, 2024. DOI: 10.1109/CSITSS64042.2024.10817033.

[2] C. Liu, X. Xu, Z. Chen, and B. Wang, "A universal-verification-methodology-based testbench for the coverage-driven functional verification of an instruction cache controller," *Electronics*, vol. 12, no. 18, p. 3821, 2023. DOI: 10.3390/electronics12183821.

[3] Abdelbaky, Dessouky, and Salem, "Python-based dram memory controller testbench: Pyuvm an early report," *Journal of Advanced Research in Applied Sciences and Engineering Technology*, vol. 52, no. 2, 176–188, Sep. 2024. DOI: 10.37934/araset.52.2.176188.

[4] D. N. Gadde, S. Kumari, and A. Kumar, "Effective design verification – constrained random with python and cocotb," May 2024, Published in DVCon Europe 2023. DOI: 10.48550/arXiv.2407.10312.

[5] G. Wang, N. Tan, Y. Cheng, and P. Zhang, "A comparative study of different verification platforms of ahb-spi," pp. 847–851, 2024. DOI: 10.1109/ICICM63644.2024.10814204.

[6] S. Marconi, E. Conti, P. Placidi, A. Scorzoni, J. Christiansen, and T. Hemperek, "A systemverilog-uvm methodology for the design, simulation and verification of complex readout chips in high energy physics applications," pp. 35–41, 2017. DOI: 10.1007/978-3-319-47913-2_5.

[7] M. Dharani, M. Bharathi, B. Rajeswari, A. M. Yadav, D. Niranjan, and A. C. D. Reddy, "Design and verification of an adder-subtractor using uvm methodology," pp. 26–30, 2023. DOI: 10.1109/CSNT57126.2023.10134642.

[8] Ankitha and H. V. R. Aradhya, "A python-based design verification methodology," vol. 23, no. 06, pp. 901–911, 2021. DOI: 10.51201/JUSST/21/05358.