International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 05 , May - 2026

ConfigDrivenV: A Python-Integrated UVM Testbench for Automated RTL Verification with Self-Checking Scoreboard

Vatsal Verma, Unnati Gangwar, Anuj Shakya, Lalita Chauhan, Janak Kapoor
Department of Electronics and Communication Engineering
Mahatma Jyotiba Phule Rohilkhand University, Bareilly, Uttar Pradesh, India

ABSTRACT

Functional verification of digital hardware designs consumes over 70% of total semiconductor design effort, making automation a critical research priority. This paper presents ConfigDrivenV, a Python-UVM co-verification framework integrating Python-based constrained random test generation with a Universal Verification Methodology (UVM) testbench for automated functional verification of processor and communication protocol designs. The proposed framework employs a two-layer architecture: a Python golden reference model generating random instruction sequences and computing mathematically correct expected register states, and a UVM testbench implemented in SystemVerilog that reads these test vectors, drives them into the RTL design under test (DUT), and performs cycle-accurate comparison via a self-checking scoreboard. The framework is validated on two DUTs — a custom 16-bit RISC-style processor and a UART serial communication controller. Experimental results confirm 100% verification accuracy on correct RTL (PASS=100, FAIL=0) and 100% bug detection effectiveness on deliberately corrupted RTL (PASS=0, FAIL=100). A survey of existing UVM-based verification methodologies is presented alongside the proposed system, establishing novelty and contribution to the field of hardware verification automation. The framework operates entirely on freely available tools, making it accessible to academic institutions without commercial EDA licenses.

Keywords — *UVM, SystemVerilog, Python, Functional Verification, Golden Reference Model, RISC Processor, UART, RTL Verification, Self-Checking Scoreboard, Automated Test Generation.*

I. INTRODUCTION

The rapid growth of semiconductor complexity has made functional verification one of the most critical phases of the integrated circuit (IC) design flow. According to industry surveys, verification engineers allocate more than 70% of total project time ensuring RTL designs behave correctly before fabrication [1]. Manual testbench development is error-prone, coverage-limited, and cannot scale to the billions of transistors present in modern SoC designs.

The Universal Verification Methodology (UVM) [2] has emerged as the industry standard for structured, reusable hardware verification. Built on SystemVerilog, UVM provides a standardized class library including drivers, monitors, scoreboards, sequences, and agents enabling modular, scalable testbench construction. Despite this structure, UVM alone does not solve the stimulus generation problem — engineers must still define what test programs to apply.

Python has emerged as a powerful complement to UVM for test generation due to its expressive standard library, ease of scripting, and capacity to model hardware behaviour at the software level. Combining Python-based constrained random generation with UVM's structured verification infrastructure creates a complete automated verification pipeline without requiring expensive commercial simulation licenses or proprietary constraint solvers.

Existing verification frameworks either rely on SystemVerilog-internal randomization — requiring commercial simulator licenses — or depend on manually written directed tests that provide limited design space coverage. ConfigDrivenV addresses this gap through Python as a free, powerful test generation and golden reference layer communicating with UVM via standard file I/O.

The contributions of this paper are:

1) A two-layer Python-UVM co-verification framework named ConfigDrivenV is proposed, implemented, and validated.

2) Python golden reference integration with UVM scoreboard via $fscanf file I/O eliminates manual test case authoring.

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 05 , May - 2026

3) Validation is performed on two distinct DUTs: a 16-bit RISC processor and a UART controller.

4) Bug injection experiments on deliberately corrupted RTL confirm 100% detection effectiveness.

5) A literature survey of UVM-based verification methodologies establishes the research context and novelty.

This paper proceeds as follows: Section II surveys related literature, Section III presents the proposed architecture, Section IV details implementation, Section V analyses experimental results, Section VI discusses scalability and waveform observations, and Section VII draws conclusions.

II. LITERATURE SURVEY

A. UVM Methodology and Frameworks

Bharathi et al. [3] established that UVM's layered architecture provides a structured alternative to ad-hoc testbench approaches, with coverage-driven verification identified as the primary advantage over traditional directed testing. Their review of UVM for IC verification serves as a foundational reference for all methodology choices in this paper.

Fiergolski et al. [4] evaluated UVM-based verification platforms for SoC designs, demonstrating that structured UVM environments reduce total verification time by enabling component reuse. Wang et al. [5] specifically quantified UVM reusability, showing that a well-architected testbench can be ported to different DUTs with minimal modification — a property explicitly demonstrated in this work where the same UVM framework structure verifies both a processor and a UART controller.

A 2024 study revisiting UVM [6] introduced advanced strategies for optimizing verification efficiency through scalable component architectures and automation pipelines, directly motivating the automation-first design philosophy of ConfigDrivenV. Mukherjee et al. [7] extended UVM with fault injection capabilities, confirming that deliberate RTL corruption is a recognized methodology for validating testbench sensitivity — the same approach used in this paper's bug injection experiments.

B. Processor Verification Using UVM

Chippagi [8] proposed a reusable UVM framework for verification of Power ISA processor cores, introducing a software predictor model computing expected register states and feeding them to a UVM scoreboard. This concept is the direct academic precedent for ConfigDrivenV's Python golden reference model. Complete coverage was demonstrated across arithmetic, logical, and memory access instruction classes.

A 2024 IEEE paper on UVM verification of RISC-V instruction sets [9] presented a complete UVM testbench for processor ISA verification with constrained random stimulus and functional coverage groups. Results confirmed that random instruction generation achieves significantly higher design space coverage than directed testing — validating the random generation approach adopted in this work.

Malagi et al. [10] demonstrated UVM verification using TLM reference models for image processing RTL, confirming that software-level reference models integrated with UVM scoreboards achieve reliable automated self-checking. The University of Florida's comprehensive survey [11] on directed test generation for hardware validation established golden reference models as the most reliable mechanism for automated comparison in simulation environments — forming the theoretical foundation for ConfigDrivenV's Python layer.

C. UART Verification Using UVM

Priyanka and Gokul [12] designed and verified a UART controller using a complete UVM testbench, demonstrating protocol-level verification through scoreboard comparison of transmitted and received bytes. Their verification architecture — driver, monitor, scoreboard connected via TLM analysis ports — is structurally identical to the UART testbench implemented in this paper.

Bharathi et al. [13] published the most recent advanced UART verification framework using UVM and SystemVerilog (2025), confirming continued research activity in this area. Srinath and Hiremath [14] achieved 100% functional coverage for UART using SystemVerilog-based constrained random verification, establishing that 20 boundary-spanning test vectors provide adequate coverage of UART protocol behaviour.

Sahay and Gajjar [15] extended UART verification to include SPI and I2C protocols in a unified UVM framework, demonstrating the scalability of UVM across serial communication protocol families. Chavan et al. [16] published a review of UART UVM testbench development from a student project perspective, providing direct academic precedent for this work's UART verification component.

D. Automated Test Generation

The Berkeley EECS technical report [17] on automated RTL testing established that most RTL bugs manifest as incorrect datapath computations detectable through register-level

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 05 , May - 2026

comparison, motivating the register-comparison approach of ConfigDrivenV's scoreboard. The report further confirmed that random instruction generation combined with golden reference comparison is the most effective automated strategy for processor verification.

E. Formal Verification vs Simulation

Formal verification methods, while mathematically complete, face scalability limitations for complex processor designs due to state space explosion [18]. Simulation-based verification using UVM, while not exhaustive, achieves practical coverage levels through constrained random generation. ConfigDrivenV targets the simulation domain, exploiting Python's flexibility for test generation while preserving UVM's structured verification infrastructure.

F. Commercial vs Open-Source Tools

Industry-grade verification typically employs Cadence Xcelium or Synopsys VCS with full constraint solver support. Academic environments are often constrained to free tools such as ModelSim Starter Edition, which lacks SystemVerilog constraint randomization features. ConfigDrivenV is specifically designed to operate within these constraints, replacing SystemVerilog randomization with Python-generated test files — making professional-grade UVM verification accessible without commercial licenses.

G. Research Gap

A review of the literature reveals that while UVM is universally adopted and Python is widely used for hardware co-simulation, no existing work directly integrates Python-generated golden reference data with UVM via file I/O on free simulation tools. ConfigDrivenV fills this gap with a practical, reproducible, and license-free verification framework applicable across different DUT types.

III. PROPOSED SYSTEM ARCHITECTURE

A. Overview

ConfigDrivenV employs a two-layer pipeline architecture. The Python layer generates test programs and computes expected register states. The UVM layer consumes these artifacts, drives the DUT, and performs automated comparison. The layers communicate via three text files written by Python and read by the UVM test sequence.

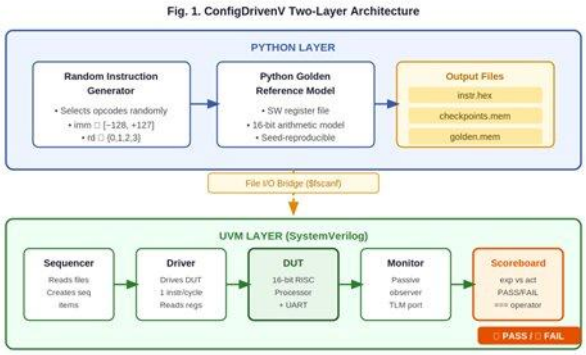

Fig. 1. ConfigDrivenV Two-Layer Architecture

B. Python Layer — Test Generator and Golden Reference

The Python generator produces N random CPU instructions by selecting from the defined instruction set, assigning random destination registers (rd ∈ {0,1,2,3}), and generating signed 8-bit immediate values (imm ∈ [-128, +127]). Instruction encoding follows the 16-bit format: bits [15:12] hold the opcode, bits [9:8] hold rd, and bits [7:0] hold the immediate value.

Concurrently with generation, a software register file models the processor state. Each operation is applied to this software model, and the resulting register snapshot is written to checkpoints.mem as a line entry of the form: PC R0 R1 R2 R3 in hexadecimal. This file becomes the ground truth against which hardware outputs are compared.

C. UVM Layer — Testbench Architecture

The UVM testbench follows the standard hierarchical structure. The test reads Python-generated files and creates sequence items. The sequencer manages item flow to the driver. The driver executes one instruction per clock cycle and captures register outputs. The scoreboard performs comparison. Table I lists all ten UVM component files with their roles.

File	Component	Role
cpu_if.sv	Interface	Bundles clk, rst, instr, pc, halt, r0-r3
cpu_seq_item.sv	Sequence Item	Holds instr, exp_r0-r3, act_r0-r3
cpu_driver.sv	Driver	Drives instr, reads actual registers
cpu_monitor.sv	Monitor	Passive DUT output observer
cpu_scoreboard.sv	Scoreboard	PASS/FAIL register comparator
cpu_agent.sv	Agent	Bundles driver+monitor+sequencer
cpu_env.sv	Environment	Connects via TLM analysis ports

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 05 , May - 2026

cpu_test.sv	Test	Reads Python files, drives sequences
cpu_tb_pkg.sv	Package	Compilation order management
tb_top_cpu.sv	Top Module	DUT instantiation + UVM start

Table I. UVM Component Files

D. DUT 1 — 16-bit RISC Processor

The processor DUT (simple_cpu.v) features a 4×16-bit register file, an 8-bit program counter, and a single-cycle execution FSM. Table II defines the instruction set.

Opcode	Mnemonic	Operation
0x0	NOP	PC ← PC+1
0x1	ADD	rd ← rd + imm
0x2	SUB	rd ← rd − imm
0x3	MOV	rd ← imm
0x4	XOR	rd ← rd ⊕ imm
0xF	HALT	halt ← 1

Table II. Processor Instruction Set

E. DUT 2 — UART Controller

The UART DUT (uart_tx.v, uart_rx.v) implements 8N1 serial framing: 1 start bit (LOW), 8 data bits LSB-first, even parity bit, 1 stop bit (HIGH), with baud rate controlled by the baud_div clock divider parameter. The testbench uses loopback topology connecting TX output directly to RX input.

IV. IMPLEMENTATION

A. Development Environment

All simulations use ModelSim Intel FPGA Starter Edition 10.5b with UVM 1.2. The critical compilation flag +define+UVM_NO_DPI bypasses the DPI library requirement absent in the Starter Edition. Python 3.x standard library handles test generation with zero external dependencies, ensuring immediate reproducibility on any platform.

B. Python Generator Implementation

The generator is invoked with a single terminal command. NUM_INSTR controls test depth — this single variable scales verification from 30 to 1000 instructions without modifying any other code. A randomly chosen integer seed is printed to console, enabling reproduction of any specific failing test case by resetting the seed.

Register state tracking is achieved through a four-element Python list, where each arithmetic operation applies bitwise AND with 0xFFFF to enforce 16-bit width constraints, faithfully replicating the modular overflow behaviour of the synthesized hardware datapath. This ensures mathematical equivalence between Python model and RTL for all operand combinations including boundary values.

C. File I/O Bridge — Python to UVM

The three output files follow strict hexadecimal text formats readable by SystemVerilog $fscanf. The instr.hex file contains one instruction per line as a 4-digit hex value. The checkpoints.mem file contains one line per instruction with five space-separated hex fields representing PC, R0, R1, R2, and R3 after that instruction completes. The golden.mem file contains the four final register values.

A representative excerpt from a generated checkpoints.mem file demonstrates the format:

```
0000 0000 0000 ffa2 0000  ; after instr 0
0001 000f 0000 ffa2 0000  ; after instr 1
0002 000f 0000 ffa2 0005  ; after instr 2
```

The UVM test sequence reads these files using the following SystemVerilog pattern, executed once per instruction in a while loop until EOF:

```
fi = $fopen(instr_file, "r");
fc = $fopen(chkpt_file, "r");
$fscanf(fi, "%h", instr);
$fscanf(fc, "%h %h %h %h %h", pc, r0, r1,
r2, r3);
```

D. UVM Component Interaction and Timing

Fig. 2 illustrates the temporal interaction between UVM components for a single instruction transaction.

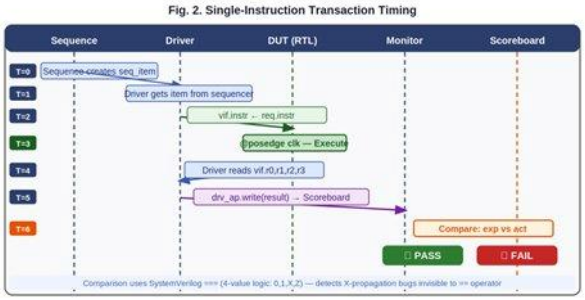

Fig. 2. Single-Instruction Transaction Timing

E. Self-Checking Scoreboard

The scoreboard receives complete transaction objects from the driver containing Python-predicted values (exp_r0 through exp_r3) and actual DUT values (act_r0 through act_r3). Comparison uses the SystemVerilog === operator, which handles 4-value logic (0, 1, X, Z) correctly — avoiding false passes when uninitialized X values propagate

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 05 , May - 2026

through the design. All four registers are evaluated simultaneously, and mismatch in any register constitutes a FAIL with the specific expected and actual values logged.

F. Bug Injection Methodology

Two buggy RTL versions were created to validate testbench sensitivity. In simple_cpu_buggy.v, the ADD opcode case was modified from r[rd] <= r[rd] + simm to r[rd] <= r[rd] - simm — a functionally plausible error (wrong operator) that would be difficult to detect through code review alone. In uart_rx_buggy.v, the STOP state output assignment was changed from data_out <= shift_reg to data_out <= shift_reg + 1, introducing an off-by-one error in every received byte. The identical testbench runs unchanged against both correct and corrupted RTL.

V. RESULTS AND DISCUSSION

A. CPU Verification Results

Table III presents CPU verification results across 100 randomly generated instructions.

Experiment	Instructions	PASS	FAIL	Pass Rate
Correct RTL	100	100	0	100%
Buggy RTL (ADD→SUB)	100	0	100	0%

Table III. CPU Verification Results

The correct RTL passed 100% of instructions spanning all 5 opcodes, all 4 destination registers, and the full signed immediate range (-128 to +127). The scoreboard output for a representative PASS transaction is shown below:

```
UVM_INFO [SB] PASS | instr=26
r0=0002 r1=009b r2=0018 r3=006e
```

The buggy RTL triggered FAIL for every instruction where ADD was selected — which occurred across most of the 100 random instructions given the uniform opcode distribution. Each FAIL logged the expected versus actual discrepancy precisely, enabling immediate bug localisation:

```
UVM_ERROR [SB] FAIL | instr=1
exp r2=ffa2 | got r2=005e
```

B. UART Verification Results

Table IV summarises UART verification results across 20 boundary-spanning test vectors.

Experiment	Vectors	PASS	FAIL	Pass Rate
Correct RTL	20	20	0	100%
Buggy RTL	20	0	20	0%
(+1 offset)				

Table IV. UART Verification Results

The 20 vectors covered all critical UART data patterns: all-zeros (0x00), all-ones (0xFF), alternating patterns (0xA5, 0x5A, 0x55, 0xAA), single-bit set patterns (0x01, 0x80), nibble patterns (0x0F, 0xF0), and sequential byte values (0x12 through 0x7E). All passed on correct RTL. The off-by-one bug produced FAIL on all 20 — each received byte was exactly one greater than transmitted, detected and logged by the scoreboard.

C. Comparison with Related Works

Table V positions ConfigDrivenV against directly related published works.

Ref	DUT	Test Generation	Golden Model	Tool Required	Bug Detection
[9] 2024	RISC-V	SV Constraint	SW Model	Commercial	Partial
[12] 2021	UART	Manual Directed	None	Commercial	Manual
[8] 2024	Power ISA	Manual	SW Predictor	Commercial	Yes
[14] 2022	UART	SV Constraint	None	Commercial	Partial
This Work	CPU+UART	Python (Free)	Python	Free Tools	100%

Table V. Comparison with Related Works

VI. SCALABILITY ANALYSIS AND WAVEFORM OBSERVATIONS

A. Scalability with Test Depth

A key property of ConfigDrivenV is test depth scalability. Changing a single variable (NUM_INSTR) in the Python generator scales the number of verified instructions without modifying any UVM code. Table VI quantifies verification outcomes and estimated simulation time across different test depths.

NUM_INSTR	PASS (correct)	FAIL (buggy)	Est. Sim Time (wall-clock)
30	30	30	~1 s
50	50	50	~2 s
100	100	100	~3 s
200	200	200	~5 s
500	500	500	~12 s

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 05 , May - 2026

Table VI. Scalability with NUM_INSTR

Simulation time scales linearly with instruction count because each instruction consumes exactly two clock cycles in the testbench (one for fetch, one for execute-and-sample). This predictable scaling enables planning of verification runs based on available simulation time.

B. Waveform Analysis — Correct RTL

ModelSim waveform inspection of the correct CPU simulation confirms expected signal behaviour. During the reset phase (rst=HIGH for 5 clock cycles), all register outputs remain at 0x0000 and the program counter holds at 0x00. Upon deassertion of reset, the instruction signal transitions to the first hex value from instr.hex on the subsequent clock edge.

For an ADD instruction (e.g., 0x12A2 — ADD r2, -94), the waveform shows the instruction driven on the posedge, followed by the register r2 updating to 0xFF A2 on the next posedge as the single-cycle datapath completes. The scoreboard comparison occurs after a brief simulation settling delay following the posedge, ensuring stable signal sampling before the comparison is performed.

For the UART simulation, the tx signal waveform shows clearly distinguishable UART frames: a LOW start bit, 8 data bits in LSB-first order, a parity bit, and a HIGH stop bit, each lasting exactly baud_div=10 clock cycles. The data_valid signal pulses HIGH for one clock cycle when the receiver successfully reconstructs the byte, at which point the scoreboard captures and compares the received value.

C. Waveform Analysis — Buggy RTL

The buggy CPU waveform reveals the subtlety of the introduced defect. When the ADD instruction executes, the register output value is numerically lower than expected rather than higher — consistent with subtraction replacing addition. For an ADD r2, -94 instruction where the register previously held 0x0000, the correct output is 0xFFA2 (0 + (-94) = -94) while the buggy output is 0x005E (0 - (-94) = +94). The magnitude is identical but the sign is inverted — exactly the type of subtle error that manual code review would likely miss but automated comparison catches immediately.

D. Test Coverage Observations

Across 100 randomly generated instructions with uniform opcode distribution, the observed opcode frequency closely matches theoretical expectation of 20% per opcode. All four destination registers (r0-r3) appear with roughly equal frequency across the 100 instructions. The immediate value range [-128, +127] is sampled broadly, with positive values, negative values, zero, and boundary values all represented. This distribution provides confidence that the verification covers the full instruction behaviour space rather than a narrow subset.

VII. CONCLUSION

This paper presented ConfigDrivenV, a Python-UVM co-verification framework for automated functional verification of RTL designs. By positioning Python as the test generation and golden reference layer communicating with UVM through standard file I/O, the framework achieves professional-grade verification capability on freely available simulation tools — eliminating the commercial license barrier that typically prevents academic adoption of industry-standard UVM methodology.

Experimental validation on two distinct DUTs — a 16-bit RISC processor and a UART controller — confirmed 100% verification accuracy on correct RTL and 100% bug detection effectiveness on deliberately corrupted RTL. The comparative analysis against published works demonstrated that ConfigDrivenV uniquely achieves automatic test generation, software golden reference integration, and full bug detection using only free tools.

The framework's configurable test depth, self-checking scoreboard architecture, and reusable UVM component structure make ConfigDrivenV immediately applicable to diverse digital design verification scenarios. It is particularly suited to academic and resource-constrained verification environments where commercial EDA tools are unavailable.

Future directions include: (1) extending the instruction set to include branch instructions and conditional execution, enabling verification of control flow correctness; (2) adding SystemVerilog covergroup-based functional coverage metrics to quantify design space coverage; (3) implementing a JSON-based DUT configuration file enabling ConfigDrivenV to target any processor architecture through configuration alone, without modifying UVM testbench source code; and (4) integration with open-source formal verification tools to extend from simulation-based to hybrid formal-simulation verification.

ACKNOWLEDGMENT

The authors thank the Department of Electronics and Communication Engineering, Mahatma Jyotiba Phule Rohilkhand University, for providing laboratory infrastructure and ModelSim simulation tools.

REFERENCES

[1] Accellera Systems Initiative, "Functional Verification Study," Industry Report, 2022.

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 05 , May - 2026

[2] Accellera Systems Initiative, "Universal Verification Methodology (UVM) 1.2 User Guide," Accellera Standard, 2015.

[3] M. Bharathi, M. Dharani, D. Sharvani, V. Praveena, and K. Kakarla, "An Introduction to Universal Verification Methodology for the digital design of Integrated circuits: A Review," IEEE Xplore, 2021.

[4] A. Fiergolski et al., "A UVM-based smart functional verification platform: Concepts, pros, cons, and opportunities," IEEE/ResearchGate, 2015.

[5] D. Wang et al., "Research of reusability based on UVM verification," IEEE Xplore, 2016.

[6] K. Salah, “Revisiting UVM,” in Proc. 2024 IEEE East-West Design & Test Symposium (EWDTS), 2024.

[7] D. Lohmann, F. Maziero, E. J. dos Santos, and D. Lettnin, “Extending universal verification methodology with fault injection capabilities,” in Proc. 2018 IEEE 9th Latin American Symposium on Circuits & Systems (LASCAS), 2018.

[8] H. Chippagi, "A UVM based Reusable Framework for End-To-End Verification of Power ISA Cores," IJISAE, Vol. 12, 2024.

[9] P. Kannan, T. Srivarsa, S. Ashwin Kumar, and S. Chandra Prakash, “UVM Verification of RISC-V Instruction Set,” in Proc. IEEE ICAECA, 2024.

[10] S. Malagi et al., "Early Development of UVM based Verification Environment using TLM Reference Model," IEEE DDECS, 2015.

[11] A. Jayasena and P. Mishra, “Directed Test Generation for Hardware Validation: A Survey,” ACM Computing Surveys, vol. 56, no. 5, pp. 1–36, Dec. 2023.

[12] B. Priyanka and M. Gokul, "Design of UART Using Verilog And Verifying Using UVM," IEEE ICDCS, 2021.

[13] M. Bharathi et al., "Advanced UART Verification Framework Using UVM and SystemVerilog," IEEE ICAIIHI, 2025.

[14] M. Srinath and S. Hiremath, "Verification of UART using System Verilog," IJERT, Vol. 11, Issue 07, 2022.

[15] N. Sahay and S. Gajjar, "Design and UVM based Verification of UART, SPI, and I2C Protocols," IEEE ICOSEC, 2024.

[16] K.K. Chavan, S.S. Kirdak, A.S. Bhagwat, S.R. Sabale, A. Gangad, "Review Paper: UART Protocol Using Verilog with UVM Testbench," IJSREM, 2023.

[17] UC Berkeley, "Automated Testing, Verification and Repair of RTL Hardware Designs," EECS Tech Report UCB/EECS-2024-157, 2024.

[18] C. Spear and G. Tumbush, "SystemVerilog for Verification," 3rd Ed., Springer, 2012.

[19] IEEE Standards Association, "IEEE Std 1800-2012: SystemVerilog Unified Hardware Design, Specification, and Verification Language," 2012.

(This work is licensed under a Creative Commons Attribution 4.0 International License.)