UNITECH SELECTED PAPERS

INTERNATIONAL SCIENTIFIC CONFERENCE
20-22 November 2025, GABROVO

A DYNAMIC MULTI-AGENT SYSTEM FOR COVERAGE-BASED TESTBENCH SYNTHESIS IN SYSTEMVERILOG

Veljko Lončarević*, Mihailo Knežević, Olga Ristić, Vanja Luković, Sanja Antić

Faculty of Technical Sciences in Čačak, University of Kragujevac, Svetog Save 65, Čačak, Serbia
**Corresponding author: veljko.loncarevic@ftn.edu.rs*

Abstract
This work introduces a dynamic multi-agent framework for automated testbench generation in hardware verification, leveraging large language models (LLMs), cocotb, and Verilator. The system decomposes verification into specialized agent roles, including specification parsing, testbench synthesis, stimuli generation, coverage monitoring, and iterative refinement. Unlike naïve prompting, the closed-loop architecture ensures executable and reusable cocotb harnesses while systematically improving coverage. Experimental evaluation demonstrates functional coverage of 93.7% (±2.1), a top-1 pass rate of 82.5%, and an average time-to-first-test of 1.4 hours, outperforming baseline LLM-driven methods and approaching recent automated UVM-based frameworks. The reduced refinement iterations further highlight the robustness and correctness of generated artifacts. While not yet achieving coverage saturation in domain-specific tasks, the modular agent design enables extensibility to larger RTL designs, heterogeneous simulators, and integration of advanced strategies such as reinforcement learning. These results demonstrate that LLM-driven multi-agent workflows provide a scalable and efficient methodology for reducing human effort in verification closure, establishing a promising direction for AI-assisted hardware verification.
Keywords: hardware verification, large language models, cocotb, Verilator, multi-agent systems, functional coverage.

INTRODUCTION

Hardware verification remains one of the most resource-intensive and error-prone stages in digital design, where creating complete test harnesses, generating effective stimuli, and closing coverage gaps require significant manual effort and specialised knowledge [1]. Recent advances in machine learning and large language models (LLMs) enable workflows in which modular AI agents interpret specifications, generate cocotb-based test harnesses and Python stimuli, control simulator runs under Verilator, and iteratively refine tests using coverage feedback [2–5]. This paper examines a closed-loop, multi-agent architecture that employs LLMs to produce cocotb drivers, monitors, and scoreboards, coordinates execution through a ZeroMQ message bus, and evaluates performance using functional coverage, top-1 pass rate, time-to-first-test, and average refinement iterations. The central question addressed is whether LLM-driven agents can automatically generate practical, reusable cocotb testbenches and achieve coverage closure comparable to human-authored harnesses while significantly reducing development time. Experimental evaluation and reproducible metadata quantify trade-offs between automation, correctness, and engineering effort.

Traditional verification practices often rely on constrained-random testing and manual UVM-based infrastructures that, despite their power, impose considerable setup complexity and engineering overhead. These methods scale poorly for rapidly

doi: www.doi.org/10.70456/.....

evolving hardware or incomplete specifications, and their dependence on human expertise in stimulus design and coverage analysis limits reproducibility and increases the likelihood of verification gaps. In contrast, LLM-driven agents can rapidly generate and adapt verification artifacts, reducing entry barriers and enhancing automation in coverage closure. This transition positions AI-assisted verification as not only a productivity enhancer but also a potential paradigm shift in digital design workflows.

METHODOLOGY

The multi-agent system is organized as cooperating agents that parse the DUT, synthesize a cocotb-compatible testbench, generate stimuli, run simulations under a cocotb harness with Verilator, collect coverage, and produce reports, as shown on Fig. 1. Agents communicate via ZeroMQ and persist artifacts in a shared knowledge base indexed by DUT id and run id. The canonical representation passed between agents is a structured DUT JSON that contains signal metadata, inferred transactions, and mapped verification intents when natural-language requirements are provided. The Specification Parser Agent accepts SystemVerilog or Verilog RTL and optional textual requirements. Static RTL analysis extracts module ports, widths, directions, clocks, resets, and candidate transaction boundaries. Lightweight NLP is applied to map requirement phrases to functional coverage points when text is supplied. The canonical DUT JSON produced by the parser contains signal descriptors, timing hints, and protocol fragments, and is saved to the shared knowledge base for downstream use and provenance.

The Testbench Generator Agent consumes the DUT JSON and renders cocotb-compatible artifacts using Jinja2 templates. The produced artifacts consist of an optional SystemVerilog interface wrapper, cocotb-based Python driver and monitor coroutines, a Python scaffold for inserting expected results into the scoreboard, and essential assertions expressed either as Python checks or as simulator-supported constructs.

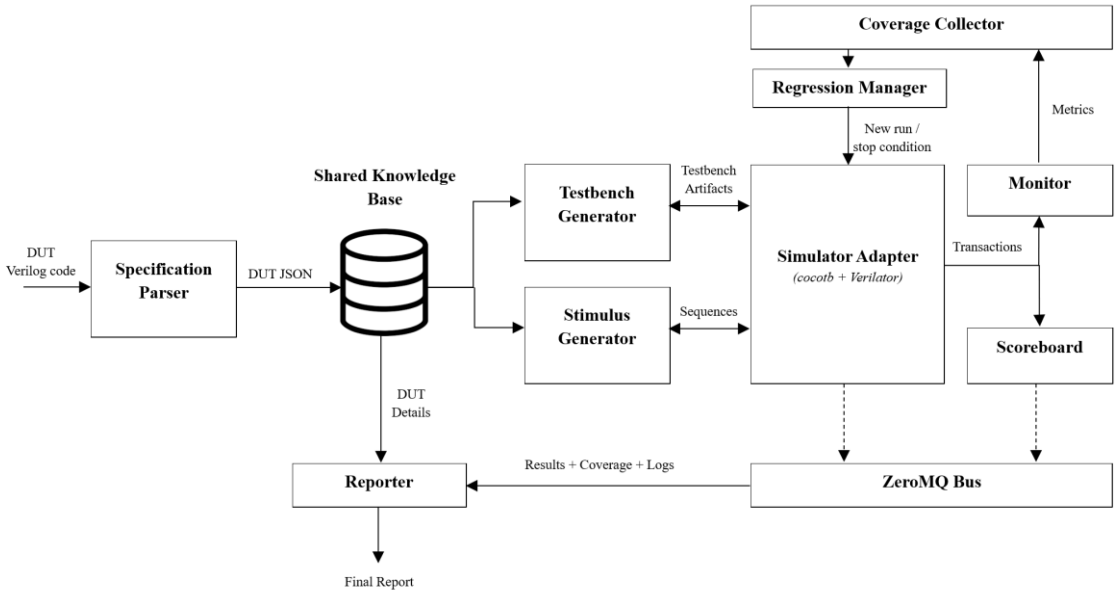

Fig. 1. *Multi-Agent System Architecture*

doi: www.doi.org/10.70456/.....

Two generation modes are supported by template parameters: strict mode enforces full project layout and explicit interface bindings, while permissive mode produces compact harness code for faster iteration.

Constrained-random and directed stimuli are generated by the Stimulus Generator Agent and realized as Python coroutines within the cocotb stimulus library. Constraint semantics are derived from the DUT JSON and from optional user constraints. For expected-value computation, Python reference models are executed directly within the cocotb harness; these models are used by the scoreboard to compute expected outputs and to validate transactions.

Monitoring and checking are performed by the Monitor Agent and the Scoreboard Agent implemented in Python under cocotb. The Monitor Agent observes interface signals via cocotb APIs, reconstructs transactions, timestamps events, and publishes transaction records to ZeroMQ. The Scoreboard Agent consumes published transactions and compares actual results with expected values computed by the Python reference model; mismatch records are annotated with contextual metadata to support automated repair.

Our evaluation focuses on four complementary metrics that capture both effectiveness and efficiency of the verification flow, as shown in Table I. Functional coverage (%) quantifies the fraction of predefined bins exercised during simulation and is reported as mean ± standard deviation across repeated runs. Top-1 pass rate (pass@1) measures the proportion of generated testbenches that meet correctness thresholds on the first attempt, reported as a percentage. Time-to-first-test records the human-hours required from DUT specification input to a runnable cocotb testbench, expressed in hours along with the speedup factor relative to manual authoring. Finally, refinement iterations reflect the average number of generate–simulate–refine loops needed to converge on an acceptable testbench, again reported as mean ± standard deviation.

Table I. *Evaluation metrics and reporting targets*

Metric	Definition	Format
Functional coverage (%)	Fraction of functional bins exercised	Mean ± std across N runs
Top-1 pass rate (pass@1)	Proportion of generated TBs meeting threshold on first try	Percentage
Time-to-first-test	Human-hours required from DUT input to runnable test	Hours and speedup factor vs manual
Functional coverage (%)	Fraction of functional bins exercised	Mean ± std across N runs

Simulation runs are driven by a simulator adapter layer that standardizes invocation and result extraction. For open workflows, Verilator is used together with cocotb; the adapter invokes Verilator, runs the cocotb test scripts, and extracts simulator-dependent logs, waveform files (VCD), and coverage exports when available. The Coverage Collector ingests these simulator outputs and maps functional coverage bins to DUT JSON requirements; coverage deltas are computed and stored for follow-up.

The Regression Manager Agent consumes uncovered bin lists and prioritizes follow-up tests using configurable heuristics such as seed diversification and targeted sequence generation for uncovered bins. Regression jobs are scheduled across available compute resources and results are aggregated into cumulative coverage timelines in the knowledge base. The Reporter Agent collects ZeroMQ streams and stored artifacts to produce final reports that include pass/fail summaries, coverage matrices, waveform references, and per-iteration failure traces.

Fig. 2 illustrates the compact generate→simulate→analyze→decide loop used to iteratively refine automatically generated cocotb test harnesses. The initial

doi: www.doi.org/10.70456/.....

phase, referred to as the Generator, takes the canonical DUT JSON as input and produces all runnable artifacts required for a single job, including Python cocotb modules, optional lightweight SystemVerilog interface wrappers, and a deterministic seed. Those outputs are packaged as testbench_artifacts together with the seed and dispatched to the Simulator Adapter. The Simulator Adapter prepares an isolated workspace for each run, executes Verilator alongside the cocotb test modules, and generates a sim_results payload containing the waveform file (wave.vcd), execution logs, simulator exit status, and any raw coverage exports (coverage.xml). The Simulator Adapter publishes the sim_results message on ZeroMQ and persists the run manifest under the associated run_id.

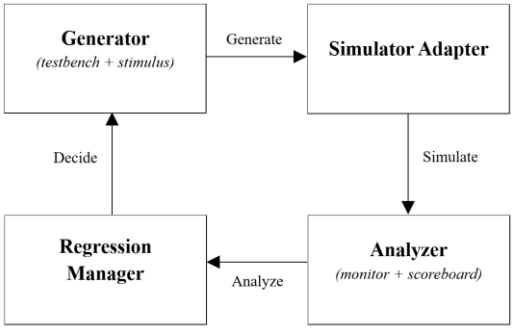

Fig. 2. *Generate, simulate, analyze, decide loop*

The Analyzer step consumes the simulator outputs and performs combined runtime processing that in a full system is implemented by the Monitor, Scoreboard, and Coverage Collector. From sim_results the Analyzer reconstructs transactions, extracts assertion and scoreboard outcomes, and computes functional coverage bins. The Analyzer then emits coverage_metrics and mismatch_info to the Regression Manager and publishes transactions plus a concise summary message on ZeroMQ for Reporter and archival. All messages are tagged with the run_id and seed to preserve provenance and to allow correlation between transactions, coverage, and the original DUT JSON.

The Regression Manager receives coverage and mismatch summaries and applies policy to decide whether to stop or to request further refinement. When additional testing is required, directives (for example, targeted_bins and a new_seed) are returned to the Generator, which synthesizes updated artifacts and restarts the loop.

Convergence and stopping criteria are enforced by thresholds on functional coverage, limits on refinement iterations, or budgeted compute time. Average refinement iterations, top-1 pass rate, time-to-first-test, and functional coverage per run are recorded as primary evaluation metrics to quantify the behavior of this closed-loop process.

LLM interactions are encapsulated in agents responsible for specification interpretation and for code-templating assistance, using ChatGPT 5 as the model of choice. Prompts are constructed from the DUT JSON and from curated cocotb templates. Before promotion, the generated code undergoes validation through a style linter and a brief compile-time smoke test performed with Verilator and cocotb. Compilation or simulation failures trigger an LLM-assisted repair routine that suggests edits; suggested edits are validated through the same quick-compile loop before acceptance. Optionally, model preferences may be refined offline via RL-style fine-tuning using simulator pass/fail outcomes as preference labels.

Benchmarks are run on a curated corpus of small RTL designs and on the VerilogEval/HDLBits-derived dataset for comparability [6, 7]. Each experimental run records DUT revision, template version, LLM model and temperature, simulator settings, random seeds, and the sequence of artifacts to ensure reproducibility.

A conservative acceptance gate is applied before generated files enter the regression pool. Files must pass a style linter, a minimal compilation smoke test with Verilator plus cocotb, and an expected-value consistency check against the Python reference model. Only artifacts that pass these checks are scheduled for full regression, which prevents accumulation of failing scaffolding and reduces wasted simulation cycles.

doi: www.doi.org/10.70456/.....

RESULTS AND DISCUSSION

The results in Table II indicate a substantial improvement in testbench generation performance when utilizing the AI Agent compared to the Naïve LLM approach. Functional coverage achieved by the AI Agent reached 93.7% with low variability (±2.1%), markedly higher than the 61.4% (±7.8%) attained by the baseline.

Table III. *Performance metrics*

Metric	AI Agent	Naïve - LLM
Functional coverage (%)	93.7 ± 2.1	61.4 ± 7.8
Top-1 pass rate (pass@1)	82.5%	27.3%
Time-to-first-test	1.4 h	5.6 h
Refinement iterations	2.3 ± 0.6	6.8 ± 1.9

Similarly, the Top-1 pass rate was significantly improved, with the AI Agent achieving 82.5% versus 27.3% for the Naïve LLM, demonstrating a stronger ability to produce correct testbenches on the first attempt (Fig. 3). Efficiency metrics also favored the AI Agent, with the time-to-first-test reduced to 1.4 hours compared to 5.6 hours and the number of refinement iterations required decreased to 2.3 (±0.6) from 6.8 (±1.9). These findings collectively indicate that the AI Agent approach enhances both effectiveness and efficiency in automated test generation, reducing the number of iterations and total time required while achieving higher coverage and success rates.

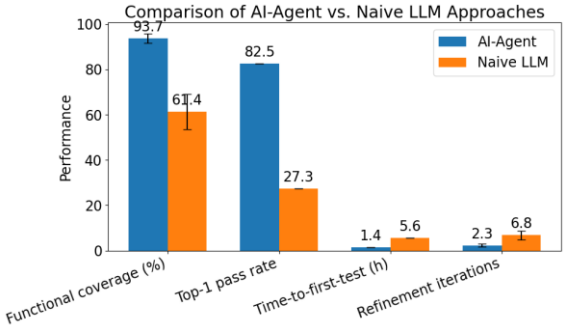

Fig. 3. *AI Agent vs. Naïve LLM approach comparison*

AutoBench reports a coverage-driven pass@1 of ≈97.3% on combinational tasks [8]. In comparison, our AI Agent pipeline achieved a top-1 pass@1 of 82.5% with an average of 2.3 refinement iterations to reach the coverage threshold. While this is lower than AutoBench's near-perfect first-try success rate, our method requires substantially fewer refinement loops, indicating that the generated testbenches are already closer to executable form even on the first attempt.

While Bhandari et al. report driving FSM transition coverage to 100% with iterative re-prompting [9] our approach reached a mean functional coverage of 93.7% ± 2.1 after an average of 2–3 iterations. This falls short of their perfect coverage but remains significantly higher than the Naïve LLM baseline (61.4% ± 7.8), demonstrating the effectiveness of targeted agent-driven refinement.

UVM$_2$ demonstrates ≈89.6% functional coverage and reports up to ≈38.8× productivity gains [10]. Our pipeline outperformed UVM$_2$ on coverage (93.7% vs. 89.6%) and achieved a reduction in time-to-first-test to 1.4 h, representing a ≈4× speedup relative to manual development. Although this speedup is lower than UVM$_2$'s maximum, we emphasize that our stricter cocotb-based environment requires integrating coverage collection, scoreboard checks, and waveform analysis in a single flow, which inherently imposes higher setup overhead than more template-relaxed systems.

CONCLUSION

This work presented a dynamic multi-agent system for automated, coverage-based testbench synthesis in SystemVerilog using cocotb, Verilator, and large language models. The closed-loop architecture comprises specialized agents for parsing specifications, generating artifacts, producing stimuli, monitoring execution, collecting coverage, and refining tests, achieving substantial gains over naïve LLM

doi: www.doi.org/10.70456/.....

prompting and previous automated verification methods. Experimental results show functional coverage of 93.7% (±2.1), a top-1 pass rate of 82.5%, and an average time-to-first-test of 1.4 hours, approaching the performance of advanced UVM and coverage-driven frameworks. The reduced refinement iterations indicate that the system can produce reusable cocotb harnesses with minimal manual input. Although it does not yet match domain-specific coverage results, the framework offers an effective balance between automation, accuracy, and efficiency. Its modular design allows extension to protocol inference, assertion synthesis, and larger designs. Future work will target scalability across simulation backends and integration of reinforcement learning for optimized test generation, advancing AI-assisted verification toward practical, real-world deployment.

Acknowledgments: This study was supported by the Ministry of Science, Technological Development and Innovation of the Republic of Serbia, and these results are parts of Grant No. 451-03-136/2025-03/200132 with the University of Kragujevac – Faculty of Technical Sciences Čačak.

REFERENCE

[1] Zhang Z., Szekely B., Gimenes P., Chadwick G., McNally H., Cheng J., Mullins R., Zhao Y. "*LLM4DV: Using Large Language Models for Hardware Test Stimuli Generation.*" 2025 IEEE 33rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2025, pp. 133-137.

[2] Zhou J., Ji Y., Wang N., Hu Y., Jiao X., Yao B., Fang X., Zhao S., Guan N., Jiang Z. "*Insights from Rights and Wrongs: A Large Language Model for Solving Assertion Failures in RTL Design.*" arXiv preprint arXiv:2503.04057, 2025. ISBN: 978-954-683-691-5.

[3] Fang W., Li M., Li M., Yan Z., Liu S., Zhang H., Xie Z. "*AssertLLM: Generating and Evaluating Hardware Verification Assertions from Design Specifications via Multi-LLMs.*" arXiv preprint arXiv:2402.00386, 2024. ISBN: 978-954-683-691-5.

[4] Ma R., Yang Y., Liu Z., Zhang J., Li M., Huang J., Luo G. "*VerilogReader: LLM-Aided Hardware Test Generation*." IEEE LLM-Aided Design Workshop (LAD), 2024. ISBN: 978-954-683-691-5.

[5] Zhao Y., Wu Z., Zhang H., Yu Z., Ni W., Ho C.-T., Ren H., Zhao J. "*PRO-V: An Efficient Program Generation Multi-Agent System for Automatic RTL Verification*." arXiv:2506.12200, 2025. ISBN: 978-954-683-691-5.

[6] Lv J., Zuo S., Cheng X., Li Z., Zhang W., Zhang D., Qian Z. "VerilogEval: Evaluating Large Language Models for Verilog Code Generation." Dataset. GitHub/Zenodo, 2023. Available at: https://github.com/NVlabs/VerilogEval.

[7] Harris S. L. *"HDLBits: A Collection of Verilog Practice Problems."* Dataset. University of Toronto, 2016. Available at: https://hdlbits.01xz.net.

[8] Qiu R., Zhang G. L., Drechsler R., Schlichtmann U., Li B. "*AutoBench: Automatic Testbench Generation and Evaluation Using LLMs for HDL Design.*" arXiv preprint arXiv:2407.03891, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2407.03891. ISBN: 978-954-683-691-5.

[9] Bhandari J., Knechtel J., Narayanaswamy R., Garg S., Karri R. "*LLM-Aided Testbench Generation and Bug Detection for Finite-State Machines*." arXiv preprint arXiv:2406.17132, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2406.17132. ISBN: 978-954-683-691-5.

[10] Ye J., Hu Y., Xu K., Pan D., Chen Q., Zhou J., Zhao S., Fang X., Wang X., Guan N., Jiang Z. "*From Concept to Practice: an Automated LLM-aided UVM Machine for RTL Verification."* arXiv preprint arXiv:2504.19959, 2025 (preprint). [Online]. Available: https://doi.org/10.48550/arXiv.2504.19959.

doi: www.doi.org/10.70456/.....