electronics

MDPI

Article

A Context-Aware Feedback Loop for AI-Assisted Verification IP Synthesis: Bridging the Gap from Natural Language to Regression-Ready

Chin-Wen Liao * and Cheng-Chia Wang

Department of Electrical and Mechanical Technology, National Changhua University of Education, Changhua City 50007, Taiwan; d1131013@mail.ncue.edu.tw
* Correspondence: tcwliao@cc.ncue.edu.tw

Abstract

The widening gap between System-on-Chip (SoC) design complexity and verification productivity has rendered traditional script-based automation insufficient. While Large Language Models (LLMs) offer promise for code synthesis, they typically fail in hardware verification contexts due to a lack of architectural consistency and an inability to reason about temporal signal semantics. This paper proposes a Context-Aware Verification Loop (CAVL) methodology, an iterative framework that integrates semantic project indexing with simulation-based feedback to achieve verification closure. Unlike static generation, CAVL employs a dynamic refinement cycle where compiler diagnostics, simulation logs, and functional coverage metrics serve as feedback signals to guide the AI agent. We validate this framework on a dual-mode I^2C Universal Verification Methodology (UVM) environment as a representative case study. The experimental results indicate, within this single-protocol context, the framework's capacity to (1) resolve complex signal-level contention issues through logic refactoring, (2) achieve complete functional coverage via directed test synthesis, and (3) maintain cross-file architectural consistency with reduced human intervention. This work presents an initial quantitative baseline for AI-driven Electronic Design Automation (EDA), suggesting that context-aware feedback loops offer a pathway toward restructuring the verification engineer's role from implementation to architectural intent specification.

Keywords: I^2C protocol; SystemVerilog; Verification IP; large language model; UVM

1. Introduction

1.1. Motivation and Problem Statement

SoC design complexity has outpaced verification productivity [1]. As features accumulate, the reachable state space and protocol interactions grow nonlinearly, making functional verification a primary schedule risk. Verification aims to demonstrate that an RTL implementation conforms to its specification over legal operating conditions, including protocol sequencing, corner cases, and system effects such as Clock Domain Crossings and power-state transitions [2,3]. Missed defects may require post-tape-out reworking; therefore, projects rely on simulation, constrained-random stimulus, assertions, formal methods, and emulation to improve defect detection and confidence.

A persistent bottleneck is the overhead of building and stabilizing the verification infrastructure itself [4]. Before meaningful stimulus can be applied, the testbench

Check for updates

Received: 10 March 2026
Revised: 19 April 2026
Accepted: 20 April 2026
Published: 21 April 2026

Copyright: © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

architecture—agents, drivers, monitors, scoreboards—regression flow, checking, and coverage must be established. For protocol-centric blocks, this infrastructure can approach the complexity of the design under test because it must generate compliant traffic and detect subtle violations [5]. Reducing time-to-first-passing-test without weakening checking rigor or coverage intent remains a key methodological objective [6,7].

Existing automated approaches, from Perl/Python (v3.10) template generators to early LLM-assisted generation tools [1,8], address individual aspects of this problem but do not close the loop between generation, simulation feedback, and coverage closure. This paper addresses that gap by proposing a structured iterative framework that couples LLM-based synthesis with simulation-oracle feedback.

1.2. Contributions of This Paper

Throughout this paper, *regression-ready* denotes a verification environment that (i) compiles and elaborates without errors under a fixed toolchain configuration, (ii) executes all test scenarios deterministically under fixed seeds, (iii) integrates into a unified `make regr` CI build target, and (iv) produces reproducible functional coverage metrics across independent simulation runs.

To address the challenges outlined above, this paper presents a structured methodology for context-aware AI-assisted verification, moving beyond simple code generation to a holistic, architectural co-design process. We utilize a modern IDE integrated with an LLM that maintains a persistent index of the entire codebase, enabling project-level reasoning. The specific contributions of this work are as follows:

- **Context-loop methodology:** We define a structured workflow where the AI agent acts as a co-designer, iteratively translating natural language requirements (e.g., "add 10-bit addressing") into a complete, cross-file implementation. This includes updating the transaction object, the driver logic, the monitor sampling, and the coverage model simultaneously, ensuring architectural consistency.
- **Automated protocol compliance and debugging:** We demonstrate the AI's capacity to analyze simulation logs and waveforms to identify complex root causes. Specifically, we present a case study where the system automatically identified a bus contention issue in a slave driver—where the verification component was incorrectly driving the clock—and refactored the code to use passive edge-detection tasks.
- **Dynamic coverage closure:** Unlike static template generators, our approach treats coverage as a dynamic target. We show how the system analyzed coverage reports (identifying holes in burst and NACK scenarios) and automatically generated targeted test cases to achieve 100% functional coverage without manual intervention.
- **Regression-ready infrastructure:** We contribute a reference implementation of a complete I^2C UVM environment that supports dual-mode (master/slave) operation, integrated into a unified regression suite (`make regr`). This demonstrates that AI assistance can deliver not just code snippets, but a functional infrastructure including Makefiles, file lists, and structured logging mechanisms.

We validate this methodology on a dual-mode I^2C Verification IP as a proof-of-concept case study. The results are specific to this configuration and are not claimed to generalize without further validation on additional protocols, simulators, and LLM families. We discuss this scope explicitly in Section 6.3.

The remainder of this paper is organized as follows. Section 2 reviews prior work on UVM-based verification, script-based automation, and the application of Large Language Models to hardware design, and positions CAVL against existing AI-assisted EDA approaches. Section 3 defines the CAVL framework, describes the experimental validation environment and protocol benchmark, details the four-phase workflow from specification

to coverage closure, and documents the tooling configuration required to reproduce the study. Section 4 presents the empirical outcomes: the synthesized transaction and coverage architectures, the role-aware driver arbitration, the automated resolution of physical-layer contention, the quantitative code-volume and coverage-convergence metrics, and the algorithmic formalization of the workflow. Section 5 interprets these results in terms of verification infrastructure latency, architectural coherence, heuristic debugging capability, and coverage-closure efficiency. Section 6 summarizes the methodological contributions, discusses implications for the verification engineering paradigm, enumerates the study's limitations, and outlines a future research roadmap toward autonomous EDA.

2. Related Work

2.1. UVM and Verification IP Development

To scale verification beyond ad hoc testbenches, industry converged on the Universal Verification Methodology (UVM), standardizing a reusable architecture on top of SystemVerilog [9,10]. UVM enables protocol-oriented Verification IP (VIP) through a factory mechanism for component substitution, a configuration database for parameter distribution, and Transaction-Level Modeling (TLM) for communication between components [11]. In principle, this supports reuse across block-, subsystem-, and SoC-level verification [12]. In practice, UVM carries substantial scaffolding overhead: even a minimal environment requires multiple coordinated classes (agent, driver, sequencer, monitor, scoreboard, environment) and correct build/connect-phase wiring. Small mistakes in port/export connections or factory registration can cause elaboration failures that are time-consuming to diagnose.

Protocol-centric VIP development introduces additional complexity. A VIP for a bi-directional bus such as I^2C must support active signal driving (master role), passive observation (monitor role), and reactive response (slave role), often within the same simulation [13–15]. Prior work on UVM-based serial interface verification has documented the labor intensity of building such environments from scratch [16–18]. The CAVL methodology directly targets this bring-up cost by synthesizing the full UVM hierarchy, including build infrastructure, in a single guided session.

2.2. Limitations of Script-Based Automation

Verification teams have traditionally used Perl/Python generators that emit class shells, module instantiations, or connectivity maps from structured inputs such as CSV or YAML templates. These tools address initial scaffolding but stop at syntactic structure: they may produce a `driver.sv` with a `run_phase`, but the protocol-specific behavior—timing, bus semantics, corner-case handling—remains manual.

The core limitation is the absence of semantic reasoning. String substitution can instantiate an `i2c_agent` template, but it cannot infer protocol constraints such as open-drain pull-up behavior or the need to distinguish active master driving from passive slave sampling [19]. Script-generated code frequently requires substantial refactoring to become functional, shifting effort from "writing from scratch" to "repairing templates." Furthermore, classical generators are static: after code emission, there is no feedback loop that uses compiler errors, simulation logs, or coverage reports to refine the generated artifacts [20]. This motivates a context-aware approach in which generation is coupled with iterative analysis of build/runtime feedback.

2.3. LLMs for Code Generation and Hardware Design

Large Language Models have become practical tools in mainstream software development, synthesizing functions, refactoring modules, and proposing fixes from error messages [21–23]. Early work on applying LLMs specifically to hardware description

languages demonstrated the feasibility of generating Verilog from English-language descriptions [3]. Subsequent work scaled this to larger benchmarks and quantified the rate at which LLMs generate syntactically correct but functionally incorrect RTL [1,5].

Domain-adapted models such as ChipNeMo [7] and VeriGen [19] improve hardware-specific generation quality through fine-tuning on EDA corpora. Conversational hardware design workflows [8] have further shown that iterative dialogue with an LLM can refine RTL beyond what is achievable in a single prompt. Parallel advances in adjacent AI-for-engineering domains, including specialized dual-model classification pipelines that combine custom and fine-tuned backbones with tailored pre-processing [24], illustrate how domain-tailored model composition can improve task accuracy; such strategies, however, have yet to be systematically transferred to multi-file hardware verification synthesis. Overall, these contributions largely focus on RTL generation and do not address the distinct challenges of UVM verification environments, which require multi-file architectural consistency, temporal protocol semantics, and coverage closure.

Applying general-purpose LLMs to hardware verification is substantially more constrained than in software domains. UVM environments are tightly coupled: changes to a transaction structure require consistent updates across the transaction object, sequences, driver/monitor logic, scoreboard, and coverage. Without project-wide context and multi-file consistency, an LLM may generate code that compiles but is functionally inconsistent [25,26]. Hardware protocols also impose strict electrical and timing semantics—for example, I^2C open-drain behavior requires releasing lines to high impedance rather than actively driving logic '1' [27]. The EDA feedback loop is expensive: compile–elaborate–simulate iterations can take minutes, making unconstrained trial-and-error generation inefficient [28,29]. Effective AI assistance in verification therefore depends on context retention across the codebase, protocol-aware modeling choices, and architectural consistency that reduces iteration count.

The CAVL methodology described in this paper addresses these gaps by integrating a persistent project-level context index, a simulation oracle, and a structured four-phase workflow. Unlike prior work focused on one-shot RTL generation [1,5], CAVL operates as a closed-loop refinement system that targets not individual files but the complete, executable verification environment.

3. Materials and Methods

3.1. Proposed Framework: The Context-Aware Verification Loop (CAVL)

This study defines and validates the *Context-Aware Verification Loop* (CAVL), a methodological framework designed to bridge the semantic gap between natural language requirements and regression-ready hardware verification environments. Unlike traditional template-based generators or stateless LLM queries, CAVL operates as a closed-loop feedback system. The iterative refinement paradigm—where output quality improves through successive feedback cycles—is well established in AI-assisted design [8]. CAVL formalizes this principle by integrating a *Generative Agent* (the synthesis engine) with a *State Manager* (the project context index) and a *Verification Oracle* (the compiler and simulator feedback).

The framework postulates that verification efficiency is maximized when the synthesis engine maintains persistent awareness of three distinct contexts:

1. **Specification context:** The immutable requirements derived from the protocol standard and test plan.
2. **Architectural context:** The evolving state of the codebase, including package dependencies, class hierarchies, and file organization.
3. **Runtime context:** The dynamic feedback provided by simulation logs, error signatures, and functional coverage databases.

By coupling these contexts, the methodology transforms the verification process from a linear implementation task into an iterative refinement cycle, where the agent automatically converges toward a compliant Verification IP (VIP).

3.2. Experimental Validation Environment

To validate the CAVL methodology empirically, we established an experimental setup consistent with industrial sign-off standards. The verification environment was architected in *SystemVerilog* [30] (IEEE 1800–2017) leveraging the *Universal Verification Methodology* (UVM 1.2) library. This choice ensures that the generated artifacts are factory-registered UVM components compatible with standard regression suites.

The synthesis engine utilized Claude 3.5 Sonnet (Anthropic, Inc., San Francisco, CA, USA; API model identifier `claude-3-5-sonnet-20241022`, released October 2024) embedded within Cursor IDE (v0.45+). This integration provided the necessary *State Manager* functionality, enabling the agent to index and retrieve cross-file references (e.g., identifying how a change in `i2c_transaction` impacts `i2c_driver`). Validation was performed using *Synopsys VCS* (v2022.06-SP2-1), chosen for its strict Language Reference Manual (LRM) compliance and robust coverage infrastructure (`.vdb`), which served as the ground-truth oracle for the agent's self-correction loops.

3.3. Protocol Benchmark: Bi-Directional Open-Drain Semantics

The efficacy of the methodology was evaluated using the Inter-Integrated Circuit (I^2C) protocol as the primary benchmark. While transactionally simple, I^2C presents specific physical-layer challenges that frequently confound statistical code generators, making it an appropriate initial stress test for semantic reasoning:

- **Electrical semantics:** The open-drain bus topology requires tri-state driving logic (driving logic '0' or high-impedance) rather than push–pull behavior.
- **Role asymmetry:** The verification component must support dynamic role switching—acting as a time-authoritative master in one phase and a clock-synchronous slave in another.
- **Bit-level contention:** Simultaneous driving by the DUT and VIP results in 'X'-state propagation, requiring precise cycle-accurate arbitration.

The target capability set included Standard mode (100 kbit/s) and Fast mode (400 kbit/s), 7-bit and 10-bit addressing, and comprehensive error injection (NACK, arbitration loss). To facilitate a closed-loop experiment, the framework also synthesized behavioral reference RTL models (master and slave), enabling immediate end-to-end simulation without external IP dependencies.

I^2C was selected as a first case study because its physical-layer constraints are concrete and verifiable, and because a complete independent specification (NXP UM10204 [31]) is publicly available, against which the generated coverage model was reviewed. The choice deliberately excludes more complex protocols such as PCIe, AXI, or DDR; generalization to those protocols is identified as future work.

3.4. Workflow Phases: From Specification to Closure

The CAVL workflow is structured into four sequential phases, each defined by specific input constraints and exit criteria. This structured approach prevents the "hallucination cascade" often observed in unconstrained LLM code generation.

Phase 1: Requirement Decomposition and Test Planning. The process initiates with the ingestion of the protocol specification. The agent synthesizes a structured `TESTPLAN.md` document, decomposing high-level requirements into atomic verification features (e.g., "Verify 10-bit address decoding," "Verify Clock Stretching timeout"). This document serves as the semantic anchor for the project; all subsequent code generation is constrained to satisfy

items listed in this plan. Simultaneously, the agent defines the repository architecture, enforcing a clean separation between the UVM agent (stimulus/monitoring), the environment (checking), and the Test Layer.

Phase 2: Structural Synthesis and Dependency Resolution. In this phase, the agent generates the UVM scaffolding. Unlike static templates, the synthesis is context-aware: the `i2c_agent`, `i2c_driver`, and `i2c_monitor` are generated concurrently to ensure port/export type compatibility. The methodology mandates the co-generation of the build infrastructure (Makefiles and file lists), ensuring that compilation directives (e.g., `+incdir+`, package ordering, and timescale definitions) are consistent with the generated SystemVerilog source, preemptively resolving non-functional bring-up errors.

Phase 3: Semantic Refinement via Simulation Feedback. This phase represents the core of the closed-loop methodology. The environment executes the initial sanity tests, and the resulting compiler logs and simulation outputs are fed back to the agent. This feedback loop enables iterative self-correction. For example, initial timing mismatches between the VIP driver and the reference RTL are detected via UVM error messages. The agent analyzes the error signature, correlates it with the driver logic, and synthesizes a corrective patch.

Phase 4: Optimization via Coverage-Guided Test Generation. The final phase targets verification closure. The agent integrates a functional coverage model (`i2c_coverage.sv`) and analyzes the coverage database after regression runs. Bins with zero hits act as prompts for the synthesis of directed random tests. For instance, an empty bin for "NACK on Data Phase" triggers the generation of `i2c_nack_test.sv`, specifically constrained to produce that condition. This cycle repeats until the aggregate coverage metric meets the sign-off threshold defined in Phase 1.

3.5. Algorithm for Dual-Mode Protocol Implementation

A critical methodological contribution is the derivation of a *Role-Aware Driver Algorithm* capable of handling the I^2C open-drain physics. To avoid bus contention—a common failure mode in automated code generation—the driver logic was explicitly separated into active (master) and passive (slave) threads.

The physical-layer interaction is modeled using a virtual interface control signal that maps logical intent to electrical state, ensuring that the VIP never actively drives a logic '1':

$$\texttt{SDA}_{physical} = \begin{cases} 0 & \text{if } \texttt{sda_drive} = 1 \text{ (Assert Low)} \\ Z & \text{if } \texttt{sda_drive} = 0 \text{ (Release High-Z)} \end{cases} \quad (1)$$

Equation (1) is enforced alongside a pull-up resistor model in the top-level testbench.

In slave mode, the methodology enforces a transition from time-based modeling to event-based modeling. The synthesized driver employs edge-synchronization primitives (e.g., `@(posedge vif.scl)`) to align its data output with the external master's clock. This algorithmic separation ensures that the VIP can dynamically switch roles within a single simulation without violating the single-driver-per-net constraint of the simulation kernel.

3.6. Heuristic Root Cause Analysis and Self-Correction

The capability of the framework to automatically diagnose failure modes was evaluated through fault injection scenarios. The methodology employs a *Heuristic Root Cause Analysis* (HRCA) strategy. When a simulation fails (e.g., a "Timeout" or "Data Mismatch"), the agent receives a composite prompt containing:

1. The relevant segment of the `UVM_log`.
2. The source code of the failing component.

3. The waveform state at the time of failure (represented as textual assertion failures and timestamped log events, as the agent does not process raw VCD files).

Using these inputs, the agent infers the causal chain. A representative validation case involved a Clock Domain Crossing (CDC) mismatch where the VIP's trigger signal was too narrow for the RTL's slow clock domain. The agent, observing the `UVM_FATAL: Timeout` and the clock definitions in `tb_top.sv`, correctly identified the timing violation and automatically refactored the trigger generation logic to extend the pulse width, thereby restoring simulation stability.

3.7. Data Collection and Metrics

To quantify the efficiency of the CAVL methodology, we tracked metrics across three dimensions:

- **Code volume:** Lines of Code (LOC) for generated UVM components and tests.
- **Verification completeness:** Percentage of functional coverage bins closed automatically vs. requiring manual intervention.
- **Workflow efficiency:** The number of prompt iterations required to reach a "clean compile" state and a "passing sanity" state.

It should be noted that a controlled comparison against a manual UVM development baseline was not conducted in this study. The reported metrics reflect the specific experimental run; no human engineer trial was performed under matched conditions. This constitutes a study limitation, discussed further in Section 6.3.

3.8. Experimental Reproducibility and Tooling Configuration

To enable the reproducibility of this study, we document the complete tooling configuration below.

Synthesis Engine. The LLM used was Claude 3.5 Sonnet (Anthropic, Inc.; API model identifier `claude-3-5-sonnet-20241022`, released October 2024). The model was accessed via Cursor IDE (v0.45+, https://cursor.sh), which provides project-level semantic indexing: the IDE maintains a persistent, cross-file codebase index that allows the model to retrieve relevant context from any file without manual copy/paste into the prompt window.

Interaction Mode. Two complementary interaction modes were used: (1) *Plan mode* during architectural phases (Phase 1 and Phase 2), in which the agent proposes a complete multi-file change plan before any edits are applied, enabling the engineer to review architectural coherence; and (2) *Agent mode* during refinement and test generation (Phase 3 and Phase 4), in which the agent applies changes directly in response to simulation log feedback.

Prompt Structure. Each composite prompt submitted during Phase 3 and Phase 4 consisted of three parts: (a) a high-level intent statement describing the desired behavior change or the failure to be resolved; (b) the relevant compiler or simulation log excerpt (typically 20–50 lines); and (c) an implicit reference to the affected source files via the codebase index. No raw VCD files were processed; waveform state was communicated through timestamped UVM log messages and assertion failure annotations.

Simulation Toolchain. Synopsys VCS v2022.06-SP2-1 was invoked with `-ntb_opts uvm-1.2`, `-timescale=1ns/1ps`, and full code coverage (`-cm line+tgl+cond+fsm +branch`). Coverage merging and reporting used `urg` against the accumulated `coverage.vdb` database.

Test Plan Reference. The `TESTPLAN.md` used as the Phase 1 output specified 16 test cases across four categories: Basic Transport (TP_001–TP_004), Transaction Types (TP_101–TP_104), Protocol Corner Cases (TP_201–TP_205), and Timing/Configuration (TP_301–TP_302). Speed modes covered Standard (100 kbit/s), Fast (400 kbit/s), and Fast+ (1 Mbit/s). Features confirmed out of scope for the current regression include multi-master arbitration and High-Speed mode (3.4 Mbit/s).

4. Results

The experimental validation of the CAVL methodology yielded a regression-ready, dual-mode I^2C Verification IP (VIP). This section presents the empirical outcomes of the study, categorized by architectural synthesis, algorithmic correctness, and quantitative efficiency. We demonstrate that the framework successfully bridged the semantic gap between natural language specification and bit-level protocol compliance, automatically resolving complex physical-layer constraints that typically defy template-based generation.

4.1. Synthesis of Observable Transaction Architectures

The first validation criterion was the generation of a semantically rich Transaction-Level Modeling (TLM) layer. Unlike standard generators that produce bare structs, the CAVL agent synthesized an `i2c_transaction` object embedded with robust observability features. A critical architectural decision made by the agent was the implementation of a formatted serialization method (`convert2string`), transforming the transaction from a data container into a self-documenting verification artifact.

The generated code (Listing 1) demonstrates a high degree of context awareness: the agent inferred the need to decode protocol-specific enumerations (e.g., `I2C_ADDR_7BIT`) and format payload arrays into readable hex dumps, significantly reducing the cognitive load during log analysis.

Listing 1. Synthesized transaction visualization method for enhanced observability.

```
virtual function string convert2string();
string s;
s = "\n-----------------------------------------------------\n";
s = {s, $sformatf(" I2C TRANSACTION\n")};
// Context-aware decoding of addressing mode
s = {s, $sformatf(" Address : 0x%0x (%s)\n", addr,
(addr_mode==I2C_ADDR_7BIT) ? "7-bit" : "10-bit")};
s = {s, $sformatf(" Direction : %s\n", direction.name())};
s = {s, $sformatf(" Payload Size : %0d bytes\n", data.size())};

// Logic for efficient payload visualization
if (data.size() > 0) begin
s = {s, " Data Content :\n"};
foreach (data[i]) begin
if (i % 16 == 0) s = {s, $sformatf(" [%04x] ", i)};
s = {s, $sformatf("%02x ", data[i])};
if ((i+1) % 16 == 0) s = {s, "\n"};
end
end
s = {s, $sformatf(" Status : %s\n", status.name())};
s = {s, "-----------------------------------------------------"};
return s;
endfunction
```

4.2. Implementation of Role-Aware Driver Arbitration

A primary challenge in verifying bi-directional buses is the implementation of a driver capable of dynamic role switching. The CAVL workflow successfully synthesized a unified `i2c_driver` that implements a *Role-Aware Finite State Machine*. Instead of segregating logic into distinct classes, the agent architected a runtime decision branch within the `run_phase`, enabling the VIP to seamlessly transition between time-authoritative (master) and clock-synchronous (slave) behaviors within a single simulation timeslice.

Listing 2 illustrates this arbitration logic. The structural correctness of this implementation is significant because it allows for full-duplex verification scenarios (e.g., Master-Write followed immediately by Slave-Read) without testbench reconfiguration.

Listing 2. Dynamic arbitration logic for dual-mode operation.

```
task run_phase(uvm_phase phase);
// Physical Layer Initialization: Release Open-Drain Lines
vif.scl_drive <= 1'b1;
vif.sda_drive <= 1'b1;

forever begin
// Dynamic Role Evaluation at Transaction Boundary
if (cfg.is_master) begin
// --- MASTER MODE (Active Driving) ---
seq_item_port.get_next_item(req);
drive_transfer(req);
seq_item_port.item_done();
end else begin
// --- SLAVE MODE (Passive Reactivity) ---
// Sync to external clock domain
wait_for_request();
end
end
endtask
```

4.3. Derivation of Objective-Driven Coverage Models

The study verified the framework's ability to translate high-level requirements into quantitative coverage metrics. The synthesized `i2c_coverage` model was not a generic template but a precise mapping of the `TESTPLAN.md` specifications. The agent automatically identified critical cross-product spaces—specifically the interaction between transaction direction and response status (ACK/NACK)—which are essential for verifying error resilience.

Listing 3 displays the generated SystemVerilog covergroups. The automatic binning of address ranges (including the "General Call" at address 0) demonstrates that the CAVL agent correctly interpreted the I^2C specification's addressing rules without explicit manual prompt engineering. The complete coverage model comprises 156 bins across four primary covergroups, ensuring comprehensive state-space sampling.

Listing 3. Specification-derived functional coverage model.

```
covergroup i2c_protocol_cg;
option.per_instance = 1;

// Requirement-Traceable Address Coverage
cp_addr_7bit: coverpoint req.addr {
bins zero = {0}; // Detected General Call semantics
bins low_range = {[1:15]};
bins mid_range = {[16:111]};
bins high_range = {[112:126]};
bins max_val = {127};
}

// Cross-Coverage for Error Path Verification
cross_dir_status: cross cp_direction, cp_status;
endgroup
```

It should be noted that both the coverage model and the directed tests used to close it were generated by the same AI agent. The coverage model was reviewed by the authors

against the NXP I^2C specification (UM10204) to verify protocol completeness; however, no independent third-party coverage oracle was used. The reported 100% coverage therefore reflects completeness relative to the generated model, not relative to a formally verified coverage plan. This is a recognized limitation of the study, discussed further in Section 6.3.

4.4. Automated Semantic Correction of Physical-Layer Contention

One of the study's most significant findings was the framework's capacity for *Heuristic Fault Isolation*. During the iterative development phase, the initial slave-mode implementation exhibited bus contention, driving the clock line (`SCL`) in conflict with the reference master. The CAVL feedback loop—analyzing the 'X'-propagation reported in the simulation logs—diagnosed the root cause as an incorrect timing model.

The agent automatically refactored the driver logic, replacing delay-based assignments with event-based synchronization (Listing 4). This transition from active driving to passive edge detection (`wait(vif.scl == 1)`) effectively resolved the contention, demonstrating that the methodology can enforce physical-layer constraints through simulation feedback.

Listing 4. Refactored passive slave task for contention avoidance.

```
task slave_read_byte(output bit [7:0] data);
vif.sda_drive <= 1'b1; // Release SDA for input mode
vif.scl_drive <= 1'b1; // Enforce Passive SCL (No Driving)

for(int i=7; i>=0; i--) begin
// Event-Based Synchronization to External Master
wait(vif.scl == 1); // Sample Data Valid
data[i] = vif.sda;
wait(vif.scl == 0); // Setup Next Bit
end
endtask
```

Implementation note: The `wait(vif.scl == 1/0)` primitives in Listing 4 do not include a timeout guard. In the event of a bus error that holds SCL stuck, the simulation will hang indefinitely. Production VIP should augment this pattern with a `fork-join_any` watchdog timer to detect and report bus-hang conditions. This limitation is a known constraint of the current implementation and was not automatically resolved by the framework.

4.5. Quantitative Analysis: Code Efficiency and Coverage Convergence

The efficiency of the CAVL methodology was quantified by analyzing the generated code volume, the progression of functional coverage, and the iteration cost. Table 1 presents the architectural footprint of the environment. The ratio of test-specific code (350+ LOC) to infrastructure code (∼500 LOC) indicates a verification architecture where effort is focused on scenario creation rather than boilerplate management.

Regarding workflow efficiency, the agent required an average of 3.5 prompt iterations to resolve initial compilation errors and 2.1 iterations to fix runtime failures during the sanity phase. Table 2 presents a summary of the estimated workflow efficiency across all four phases. These figures were reconstructed retrospectively from session logs; they were not formally instrumented during the experiment and should be interpreted as approximate. No controlled comparison against manual UVM development was performed, and the claimed efficiency gains are therefore indicative rather than conclusive.

Table 3 illustrates the effectiveness of the coverage-driven closure loop. The initial regression achieved only 41% coverage, identifying significant gaps in error handling and extended addressing. Through the automated generation of directed tests (`i2c_nack_test`, `i2c_burst_test`), the framework converged to 100% functional coverage, validating the closed-loop optimization strategy.

Table 1. Code generation metrics: Dual-mode I^2C UVM environment. The asterisk (*) in `src/tests/*.sv` denotes an aggregate over all SystemVerilog test files under that directory; bold indicates the aggregated total.

Component Layer	File Artifact	Synthesized LOC
Testbench Top	`tb_top.sv`	121
UVM Agent Core	`i2c_agent.sv`	59
Driver (Dual-Mode FSM)	`i2c_driver.sv`	297
Monitor (Passive)	`i2c_monitor.sv`	85
Transaction Object	`i2c_transaction.sv`	72
Coverage Model	`i2c_coverage.sv`	95
Scenario Layer	`src/tests/*.sv`	350+
Total Environment		**1079**

Table 2. Estimated CAVL workflow efficiency by phase (retrospectively reconstructed). "N/A" indicates the metric is not applicable to Phase 1 (requirement decomposition), which involves no compilation.

Metric	Phase 1	Phase 2	Phase 3	Phase 4
Prompts exchanged (est.)	3	8	∼21	∼15
Elapsed session time (h, est.)	0.5	1.5	2.5	1.5
Avg. iters. to clean compile	N/A	3.5	3.5	2.1
Coverage iterations (Phase 4)	—	—	—	4
Manual LOC edits (est.)	<5	<10	<20	<10

Note: All values are approximate; prompt counts and edit counts were not formally instrumented. Total estimated session time: ∼6 h.

Table 3. Convergence of functional coverage via directed test synthesis.

Protocol Feature	Associated Coverpoint	Initial Coverage	Final Coverage
Addressing Mode	`cp_addr_mode`	50%	100%
Address Ranges	`cp_addr_7bit`	40%	100%
Bus Direction	`cp_direction`	100%	100%
Response Status	`cp_status`	33%	100%
Burst Throughput	`cp_data_size`	20%	100%
Complex Sequencing	`cp_repeated_start`	0%	100%
Aggregate Metric	**Protocol Compliance**	**41%**	**100%**

4.6. Simulation Performance and Regression Stability

The stability of the synthesized environment was evaluated under regression conditions. Table 4 summarizes the execution metrics for the final test suite. The environment demonstrated deterministic behavior across all modes, with execution times scaling linearly with packet count. Notably, the `i2c_sanity_test` successfully exercised dynamic mode switching (master → slave) without simulator lockups, confirming the robustness of the arbitration logic described in Section 4.7.

Table 4. Regression suite performance metrics †.

Test Scenario	Traffic Volume	Operational Mode	Wall Time (s)	Result
`i2c_sanity_test`	1000 Pkts	Dual (M ↔ S)	1.2	PASSED
`i2c_slave_test`	100 Pkts	Slave Only	0.4	PASSED
`i2c_burst_test`	50 Pkts	Master Only	0.3	PASSED
`i2c_nack_test`	20 Pkts	Master Only	0.2	PASSED
`i2c_10bit_test`	50 Pkts	Master Only	0.3	PASSED
`i2c_speed_test`	100 Pkts	Master (Fast Mode)	0.4	PASSED

† Wall times represent simulation execution only; VCS compilation and elaboration phases are excluded and typically require 30–90 seconds per build invocation.

4.7. Algorithmic Formalization of the CAVL Workflow

To decouple the methodology from specific implementation tools, we extracted the core control logic governing the environment's construction and execution. Algorithm 1 formalizes the *Context-Aware Generation Workflow*, modeling it as an optimization problem where the objective function is the minimization of compilation errors and coverage holes. This formalization highlights the iterative nature of the process, distinguishing it from linear "fire-and-forget" code generation.

Algorithm 2 subsequently abstracts the *Dual-Mode Arbitration Logic*. This algorithm encodes the fundamental asymmetry of the I^2C protocol: the master branch operates as a time-authoritative scheduler, while the slave branch operates as a reactive state machine. By enforcing this algorithmic separation, the VIP guarantees contention-free operation on open-drain buses, a property ensured by the conditional logic evaluated at each transaction boundary.

Algorithm 1 Context-aware UVM generation and closure workflow.

Input: Natural Language Protocol Specification S_{spec}
Output: Regression-Ready Verification Environment E_{final}
Initialize Project Context C_{proj} (File System, Testplan)
$E_{curr} \leftarrow$ GENERATESKELETON(S_{spec}, C_{proj})
while Compilation Errors exist in E_{curr} **do**
    $L_{err} \leftarrow$ PARSELOG(VCS Output)
    ANALYZEERROR(L_{err}, C_{proj}) ▷ Identify Dependency/Syntax Issues
    $E_{curr} \leftarrow$ REFACTOR(E_{curr}, L_{err}) ▷ Apply Context-Aware Fix
end while
for each Test Case T_i defined in $C_{proj}.Testplan$ **do**
    $E_{curr} \leftarrow$ SYNTHESIZETEST(T_i)
    $R_{sim} \leftarrow$ RUNSIMULATION(E_{curr})
    if $R_{sim} == FAIL$ **then**
        $D_{root} \leftarrow$ ROOTCAUSEANALYSIS(R_{sim}, E_{curr})
        $E_{curr} \leftarrow$ APPLYPATCH(D_{root}) ▷ e.g., Fix Trigger Pulse Width
        goto line 11
    end if
end for
$DB_{cov} \leftarrow$ EXTRACTCOVERAGE(E_{curr})
while $DB_{cov} < 100\%$ **do**
    $H_{bin} \leftarrow$ IDENTIFYHOLES(DB_{cov}) ▷ e.g., NACK bin empty
    $T_{directed} \leftarrow$ GENERATEDIRECTEDTEST(H_{bin})
    ADDTOREGRESSION($T_{directed}$)
    $DB_{cov} \leftarrow$ RUNREGRESSION
end while
Return E_{curr}

Algorithm 2 Dual-mode driver arbitration and protocol execution logic.

Initialize Open-Drain Interface ($SDA \leftarrow Z, SCL \leftarrow Z$)
loop
  if `cfg.is_master` == True **then** ▷ **Active Mode (Time-Authoritative)**
    $Req \leftarrow$ GETNEXTITEM
    SENDSTART
    SENDBYTE($Req.Address|Req.RW$)
    $Ack \leftarrow$ SAMPLEACK
    if $Ack == ACK$ **then**
      TRANSFERPAYLOAD($Req.Data$)
    end if
    SENDSTOP
  else ▷ **Reactive Mode (Clock-Synchronous)**
    Wait for Start Condition ($SDA \downarrow$ while $SCL = 1$)
    $Addr \leftarrow$ PASSIVEREADBYTE ▷ Sync to External SCL edges
    SENDACK
    if $Command == Write$ **then**
      $Data \leftarrow$ PASSIVEREADBYTE
      SENDACK
    else
      PASSIVEWRITEBYTE($Data$)
      PASSIVESAMPLEACK
    end if
    Wait for Stop Condition
  end if
end loop

5. Discussion

5.1. Reduction in Verification Infrastructure Latency

A defining metric for verification productivity is Time-to-First-Test (TTFT)—the critical path interval between the freezing of a specification and the successful execution of the first valid transaction against the DUT. In conventional UVM workflows, this phase is disproportionately dominated by non-recurring engineering (NRE) tasks: the assembly of class hierarchies, the wiring of TLM ports, and the configuration of build scripts. While essential for compilation, these integration activities contribute zero direct value to the primary objective of defect detection. Manual scaffolding is also prone to "integration friction"—subtle defects such as package import ordering, timescale mismatches, or factory registration errors—that can stall a project for days before a single line of functional stimulus is executed.

Our experimental results indicate that the CAVL framework compresses this latency by treating the verification environment as a coupled system rather than a set of disjointed files. By co-synthesizing the UVM hierarchy alongside the simulation infrastructure (Makefiles, file lists), the methodology enforces consistency by design. Moreover, the automated generation of lightweight behavioral reference models decouples verification development from RTL availability, enabling a shift-left workflow where the testbench architecture and stimulus logic are stabilized against a generated model before the actual DUT is integrated. These observations are specific to the I^2C case study and may not hold uniformly for more complex protocols or larger design teams.

5.2. Architectural Coherence and Technical Debt Management

A pervasive concern with automated code generation is the risk of producing unstructured code that satisfies immediate functional requirements but proves unmaintainable. In manual flows, this often manifests through the "Copy–Modify" antipattern, where

engineers clone legacy environments to bootstrap new projects, inadvertently propagating deprecated methodologies and latent bugs.

Contrary to these risks, the synthesized artifacts in this study exhibited adherence to UVM 1.2 separation of concerns. The generated architecture maintained a rigid boundary between the *Agent Layer* (encapsulating protocol mechanics and signal-level timing) and the *Test Layer* (orchestrating high-level scenarios). This separation ensures that the introduction of complex new scenarios entails only additive changes to the test library, without requiring invasive refactoring of the core driver state machines. Furthermore, the unified management of SystemVerilog packages (`i2c_pkg.sv`) effectively resolved symbol visibility conflicts, a common source of fragility in large-scale regression suites. Whether this architectural quality would persist at greater environment scale or with a different protocol remains an open question.

5.3. Heuristic Diagnosis and Semantic Refactoring of Physical Layers

The successful automated resolution of the slave-mode bus contention issue indicates that the CAVL framework operates beyond syntactic auto-completion. It functions as a heuristic reasoning engine capable of debugging physical-layer constraints. The observed failure—simultaneous driving of the `SCL` line by both the DUT and the VIP—was a semantic violation of the I^2C protocol's open-drain topology.

Resolving this required a fundamental shift in control model: transitioning the driver from an active, open-loop timing model (delay-based) to a passive, closed-loop reactivity model (edge-synchronized). In a traditional debugging loop, isolating this root cause requires an engineer to manually correlate a high-level simulation hang with microsecond-level waveform contention. The CAVL framework accelerated this by pattern-matching the error signature (`X`-state propagation and timeouts) against the driver's semantic model. Crucially, the agent did not merely patch timing values; it performed a semantic refactoring, structurally altering the task implementation to use event-waiting primitives. This ability to derive structural corrections from behavioral anomalies distinguishes the methodology from simple coding assistants.

5.4. Systematic Convergence of Functional Coverage

Traditional constrained-random verification relies on the law of large numbers to close corner-case coverage holes. This stochastic process is inherently inefficient; thousands of redundant cycles are often expended to hit a single low-probability bin (e.g., an arbitration loss during a specific address phase). As the state space of the protocol grows, this "Monte Carlo" approach yields diminishing returns, leading to the well-known coverage plateau.

The methodology presented in this work replaces this probabilistic search with a targeted, inverse-problem approach. By treating coverage holes as explicit prompts for synthesis, the framework closes the verification loop systematically. The automated generation of `i2c_nack_test` and `i2c_burst_test` demonstrated that the agent could infer the necessary stimulus constraints (e.g., forcing a non-existent address to trigger a NACK) directly from the absence of data in the coverage database. The implication for industrial application is that "smart," directed tests may reduce the total compute resources required for regression in the final percentile of coverage closure. Whether this advantage holds for significantly more complex state spaces remains to be evaluated.

6. Conclusions

6.1. Summary of Methodological Contributions

This study establishes the initial efficacy, in a single I^2C case study, of the *Context-Aware Verification Loop* (CAVL), a structured methodology for the automated synthesis of

regression-ready hardware verification environments. By reframing verification environment construction as a closed-loop control problem, we demonstrated that LLMs—when integrated with a project-aware state index and a simulation feedback oracle—can transcend simple code completion to perform architectural co-design within the scope studied.

The experimental validation on a dual-mode I^2C Verification IP yielded three distinct contributions. First, we formalized a *Scaffold Synthesis Algorithm* that eliminates NRE costs associated with UVM bring-up by co-generating the component hierarchy, transaction abstractions, and build infrastructure. Second, the study demonstrated *Heuristic Physical-Layer Debugging*: the framework's ability to automatically diagnose bus contention and perform semantic refactoring proves that generative agents can enforce physical protocol constraints through iterative simulation feedback. Third, we validated a *Deterministic Coverage Closure* strategy. By inverting the verification problem—using coverage holes as prompts for directed test synthesis—the methodology replaced stochastic search with a targeted optimization loop, achieving 100% functional coverage convergence without human intervention on the generated coverage model.

6.2. Implications for the Verification Engineering Paradigm

The results suggest a potential restructuring of the verification engineer's role, shifting the cognitive load from syntactic implementation to semantic specification. In the traditional UVM workflow, significant engineering bandwidth is consumed by boilerplate friction—the mechanical effort of wiring factory registrations, analysis ports, and configuration databases. The proposed methodology automates these structural tasks, potentially elevating the abstraction level of the discipline. This trajectory parallels algorithmic modernization efforts observed in other engineering disciplines, where migration from legacy manually maintained pipelines to algorithm-centric, platform-aware implementations has refocused engineers on specification fidelity and validation rather than low-level plumbing [32].

This transition implies that the value of the human engineer will increasingly reside in intent specification and architectural oversight. The engineer becomes the verifier-of-the-verifier, responsible for defining rigorous test plans, specifying corner cases, and auditing the correctness of the AI-generated checkers. This shift may lower the barrier to deploying industry-standard verification architectures for teams with limited UVM expertise, while allowing senior architects to focus on complex system-level behaviors rather than signal-level connectivity.

6.3. Limitations and Scope

While the results of this case study are encouraging, several critical limitations must be acknowledged before the methodology can be considered for broader industrial adoption.

Single-protocol scope: This study was validated on I^2C exclusively. I^2C was chosen because its physical-layer semantics are concrete and because a complete independent specification exists (NXP UM10204). No evaluation was performed on more complex protocols such as AXI, PCIe, or DDR, and generalization of the CAVL methodology to those protocols cannot be assumed without additional experimentation.

Single LLM and single simulator: The results were obtained with one model (Claude 3.5 Sonnet) and one simulator (Synopsys VCS). Performance may differ with other LLM families or with simulators that produce differently formatted error messages.

Absence of a controlled baseline: No manual UVM development trial was performed under matched conditions. The reported LOC counts, iteration averages, and elapsed times are observations from a single guided session, not the product of a controlled experiment.

Claims regarding efficiency gains over manual development are therefore not quantitatively supported and should be interpreted as indicative.

Coverage model independence: Both the functional coverage model and the directed tests that close it were generated by the same AI agent. The coverage model was reviewed by the authors against the NXP I^2C specification; however, no independent third-party coverage oracle was used. The reported 100% coverage reflects completeness relative to the AI-generated model and does not constitute an independently verified completeness claim.

Context saturation and state drift: As project complexity scales, the finite context window of the synthesis engine can lead to "state drift," where the agent proposes changes that contradict established architectural invariants. Mitigating this requires rigorous context management, including modular scoping of prompts and the re-assertion of design constraints at phase boundaries.

Stochasticity in synthesis: The stochastic nature of LLMs means that equivalent specifications can yield divergent implementations across runs. The workflow must be anchored by rigid style guides and structural templates to ensure the AI functions as a conformant developer rather than an unconstrained generator.

These limitations collectively define the current research scope and motivate the future work described below. We recommend a zero-trust posture toward AI-generated verification artifacts: all generated code should be treated as untrusted until validated by independent compilation, simulation, and author review of the coverage model's protocol completeness.

6.4. Future Roadmap: Toward Autonomous EDA

The CAVL framework represents a foundational step toward fully autonomous EDA. Analogous trajectories toward AI-driven autonomous inspection and predictive-maintenance loops are already emerging in adjacent cyber–physical engineering domains [33,34], which suggests that a similar transition is plausible for verification provided that context management and independent validation remain first-class concerns. Building on these results, we identify three high-potential research vectors:

1. **Formal-simulation co-synthesis:** Future workflows should integrate formal verification. A hybrid neuro-symbolic approach could use the LLM to extract intent models (temporal properties) from specifications, which are then mathematically proven by a symbolic model checker, providing correctness guarantees that simulation alone cannot offer.
2. **Multi-modal waveform reasoning:** The current debugging loop relies on textual logs. Integrating agents with direct perception of waveform databases (VCD/FSDB) would enable visual debugging, allowing the system to correlate signal edges, measure pulse widths, and detect glitches directly, significantly reducing the Mean Time to Resolution (MTTR) for complex timing violations.
3. **Self-healing regression pipelines:** Extending the coverage-closure loop, we envision CI systems capable of self-repair: automatically isolating the failing seed, triaging the root cause (DUT bug vs. testbench error), and proposing a candidate patch, creating a resilient verification ecosystem that adapts to design changes in real time.

Context-aware AI is not merely a productivity tool but a candidate transformative technology for hardware verification. By embedding generative intelligence within a disciplined, feedback-driven engineering workflow, the industry may break the linear relationship between design complexity and verification effort. The present work establishes the methodology and identifies the open research questions that must be addressed to realize that potential at industrial scale.

Author Contributions: Methodology, C.-W.L.; writing—original draft preparation, C.-W.L.; writing—review and editing, C.-C.W.; supervision, C.-W.L.; project administration, C.-C.W.; resources, C.-C.W. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The original contributions presented in this study are included in the article. Further inquiries can be directed to the corresponding author.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

CAVL	Context-Aware Verification Loop
VIP	Verification IP
UVM	Universal Verification Methodology
LLM	Large Language Model
RTL	Register-Transfer Level
DUT	Design Under Test
EDA	Electronic Design Automation
TLM	Transaction-Level Modeling
TTFT	Time-to-First-Test
NRE	Non-Recurring Engineering
I^2C	Inter-Integrated Circuit
FSM	Finite State Machine
CDC	Clock Domain Crossing
CI	Continuous Integration
MTTR	Mean Time To Resolution
SVA	SystemVerilog Assertion
IDE	Integrated Development Environment
VCS	Verilog Compiler Simulator
DVE	Discovery Visual Environment
HRCA	Heuristic Root Cause Analysis

References

1. Thakur, S.; Ahmad, B.; Pearce, H.; Tan, B.; Garg, B.; Karri, R.; Vasan, S. VerilogEval: Evaluating large language models for Verilog code generation. In Proceedings of the 2023 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), San Francisco, CA, USA, 29 October–2 November 2023; IEEE: Piscataway, NJ, USA, 2023.
2. Gaurav, K.; Kumar, V.; Singh, B.K. Dependability analysis of a system using state-space modeling techniques: A systematic review. *IEEE Trans. Reliab.* **2023**, *72*, 1340–1354. [CrossRef]
3. Pearce, H.; Tan, B.; Ahmad, B.; Bhatt, R.; Karri, R. Dave: Deriving automatically Verilog from English. In Proceedings of the 2020 ACM/IEEE Workshop on Machine Learning for CAD (MLCAD), Virtual, 16–20 November 2020; ACM: New York, NY, USA, 2020.
4. Jiang, Y.; Sun, S.; Zheng, X. Regression testing optimization for ROS-based automated systems: A comprehensive review of techniques. *arXiv* **2025**, arXiv:2506.16101.
5. Lu, Y.; Liu, S.; Zhang, Q.; Xie, Z. RTLLM: An open-source benchmark for design RTL generation with large language models. In Proceedings of the 29th Asia and South Pacific Design Automation Conference (ASP-DAC), Incheon, Republic of Korea, 22–25 January 2024; IEEE: Piscataway, NJ, USA, 2024; pp. 722–727.
6. Korol, V. Integrating AI tools into the continuous testing process. *Am. J. Eng. Technol.* **2025**, *7*, 222–229. [CrossRef]
7. Liu, M.; Ene, T.; Kirby, R.; Cheng, C.; Pinckney, N.; Liang, R.; Alben, J.; Anand, H.; Banerjee, S.; Bayraktaroglu, I.; et al. ChipNeMo: Domain-adapted LLMs for chip design. *arXiv* **2023**, arXiv:2311.00176.
8. Blocklove, J.; Garg, S.; Karri, R.; Pearce, H. Chip-Chat: Challenges and opportunities in conversational hardware design. In Proceedings of the 2023 ACM/IEEE 5th Workshop on Machine Learning for CAD (MLCAD), Edinburgh, UK, 10–13 September 2023; ACM: New York, NY, USA, 2023.

9. Oye, E. Designing a reusable UVM verification framework for multi-protocol high-speed interfaces in SoCs. *Electronics* **2025**. Available online: https://www.researchgate.net/profile/Micheal-Wilson-2/publication/392305440_Designing_a_Reusable_UVM_Verification_Framework_for_Multi-Protocol_High-Speed_Interfaces_in_SoCs/links/683c9c4a6a754f72b58fbfd5/Designing-a-Reusable-UVM-Verification-Framework-for-Multi-Protocol-High-Speed-Interfaces-in-SoCs.pdf (accessed on 19 April 2026).
10. Rao, T. Modularity, scalability, reusability, configurability, and interoperability of ASIC/FPGA verification IP. *Electronics* **2022**. Available online: https://www.diva-portal.org/smash/record.jsf?pid=diva2:1650927 (accessed on 19 April 2026).
11. Albu, A.M.; Pollo, G.; Burrello, A.; Pagliari, D.J.; Tesconi, C.; Panaro, L.; Soldi, D.; Autieri, F.; Vinco, S. Integrating SystemC TLM into FMI 3.0 co-simulations with an open-source approach. *arXiv* **2025**, arXiv:2508.20223.
12. Accellera Systems Initiative. *Universal Verification Methodology (UVM) 1.2 User Guide*; Accellera Systems Initiative: Napa, CA, USA, 2015.
13. Liao, C.W.; Yu, H.C.; Liao, Y.C. Verification of SPI protocol using universal verification methodology for modern IoT and wearable devices. *Electronics* **2025**, *14*, 837. [CrossRef]
14. Lu, C.Y.; Su, W.Z.; Deng, C.H.; Liao, Y.C. Design and validation of SystemVerilog I^2C VIP with integrated assertions and error injection strategies. *Electronics* **2025**, *14*, 3574. [CrossRef]
15. Deshpande, U.; Goh, K.O.M.; Araujo, S.; Chate, R.; Bhosale, A. Functional Modelling and Validation of the Arm PrimeCell DMA Controller Using SystemC for SoC Prototyping. SSRN. 2025, 5355255 . Available online: https://www.researchgate.net/publication/394709392_Functional_Modelling_and_Validation_of_the_Arm_Primecell_Dma_Controller_Using_Systemc_for_SOC_Prototyping (accessed on 19 April 2026).
16. Kumar, P.S.; Rajalakshmi, R.; Kumar, N.H.; Gupta, B.P.; Nikhilesh, C.H.; Pavan, J.S.P. Robust serial driver verification through UVM framework. In Proceedings of the 2024 Control Instrumentation System Conference (CISCON), Manipal, India, 2–3 August 2024; IEEE: Piscataway, NJ, USA, 2024; pp. 1–6.
17. Ali, A.M.; Shalaby, A.; Saif, S.; Taher, M. A UVM-based verification approach for MIPI DSI low-level protocol layer. In Proceedings of the 2022 International Conference on Microelectronics (ICM), Casablanca, Morocco, 4–7 December 2022; IEEE: Piscataway, NJ, USA, 2022; pp. 74–77.
18. Bodingbauer, J. System-Level Verification and Testing of a Safety-Critical SoC Using HW/SW Co-Simulation. Ph.D. Thesis, Vienna University of Technology, Vienna, Austria, 2022.
19. Zhao, S.; Fang, G.; Lu, F.; Guo, G.; Hu, J.; Liu, Y. VeriGen: A large language model for Verilog code generation. *ACM Trans. Des. Autom. Electron. Syst.* **2024**, *29*, 1–22. [CrossRef]
20. Indamutsa, A. Scalable and Extensible Cloud-Based Low-Code Model Repository. Ph.D. Thesis, Università Degli Studi Dell'Aquila, L'Aquila, Italy, 2023.
21. Masamba, N.P. Intelligent Testing for Simulation-Based Verification of Complex Industrial Hardware. Ph.D. Thesis, University of Bristol, Bristol, UK, 2024.
22. Lima, B.; Pinto, R. Current challenges and future perspectives in testing IoT systems: A comprehensive review. *IEEE Sens. Rev.* **2025**, 3, 22–47. [CrossRef]
23. Hao, Y. Analog Integrated Circuit Design Empowered by Artificial Intelligence Techniques: From Building Blocks to Small Systems. Ph.D. Thesis, University of Glasgow, Glasgow, UK, 2025.
24. Prema, G.; Rithanya, P.; Shiva Priya, A. Dual-model approach for knee osteoarthritis classification using custom CNN and fine-tuned VGG16 with histogram equalization. *Smart Wearable Technol.* **2026** . [CrossRef]
25. Ray, P.P. A survey on model context protocol: Architecture, state-of-the-art, challenges and future directions. *Authorea Prepr.* **2025** . [CrossRef]
26. Sharma, G.; Kulkarni, V.; King, M.; Huang, K. Towards unifying quantitative security benchmarking for multi-agent systems. *arXiv* **2025**, arXiv:2507.21146.
27. Attia, S.; Betz, V. Toward software-like debugging for FPGAs via checkpointing and transaction-based co-simulation. *ACM Trans. Reconfigurable Technol. Syst.* **2023**, *16*, 1–24. [CrossRef]
28. Chang, J.; Jaiantilal, A.; Kuchroo, S.; Tung, V.; Chen, Y.-H.; Kumar, S. AutoChip: Automating HDL generation using LLM feedback. *arXiv* **2023**, arXiv:2311.04887. [CrossRef]
29. Dinu, A.; Danciu, G.M.; Ogrutan, P.L. Cost-efficient approaches for fulfillment of functional coverage during verification of digital designs. *Micromachines* **2022**, *13*, 691. [CrossRef] [PubMed]
30. *IEEE Std 1800-2017*; IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language. IEEE: New York, NY, USA, 2018.
31. *UM10204*; I^2C-bus Specification and User Manual, Rev. 7.0. NXP Semiconductors: Eindhoven, The Netherlands, 2021.
32. Gopal, S.; Bhat, P.; Jumbe, S.; Subramanian, M.; Reddy, V.; Roy, P. From wires to wearables (3): ST-segment fidelity across modalities: Sydäntek's algorithmic transition from legacy leads to wearables. *Smart Wearable Technol.* **2026** . [CrossRef]

33. Islam, M.M.; Emon, J.I.; Ng, K.Y.; Asadpour, A.; Aziz, M.R.A.; Baptista, M.L.; Kim, J.M. Artificial intelligence in smart manufacturing: Emerging opportunities and prospects. In *Artificial Intelligence for Smart Manufacturing and Industry X.0*; Springer Nature: Cham, Switzerland, 2025; pp. 9–36.
34. Karabegovic, I. AI-enhanced mobile and collaborative robotics for autonomous inspection and predictive maintenance in smart manufacturing. *Smart Wearable Technol.* **2025**. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.