
Proceedings Paper:
Ye, Junhao, Pan, Dingrong, Liu, Hanyuan et al. (2026) Uvmarvel: an Automated LLM-Aided UVM Machine for Subsystem-Level RTL Verification. In: Design Automation Conference 2026.

UVMarvel: an Automated LLM-aided UVM Machine for Subsystem-level RTL Verification

Junhao Ye[1,2], Dingrong Pan[2], Hanyuan Liu[1,2], Yuchen Hu[1,2], Jie Zhou[1,2], Ke Xu[1,2], Xinwei Fang[3], Xi Wang[1,2], Nan Guan[4], Zhe Jiang[1,2†]
[1]Southeast University, China [2]National Center of Technology Innovation for EDA, China
[3]University of York, UK [4]City University of Hong Kong, Hong Kong
[†]Corresponding author: zhejiang.uk@gmail.com

Abstract

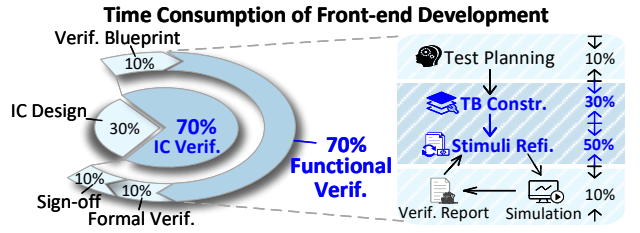
Verification presents a major bottleneck in Integrated Circuit (IC) development, consuming nearly 70% of total effort. While the Universal Verification Methodology (UVM) improves reuse through structured verification environments, constructing subsystem-level UVM testbenches and generating high-quality stimuli still require extensive manual coding, repeated EDA tool runs, and deep protocol and micro-architectural expertise. We present **UVMarvel**, an automated verification framework that leverages Large Language Models (LLMs) to build UVM testbenches for subsystem-level RTL. **UVMarvel** introduces an Intermediate Representation (IR) and a Bus Protocol Library to translate heterogeneous specifications into protocol-correct subsystem-level UVM testbenches, and employs a Signal Tracker and a Verilog Patching Library to guide LLM-based stimuli refinement. **UVMarvel** is the first framework capable of automatically constructing subsystem-level UVM testbenches across mainstream bus protocols, and it achieves an average code coverage of 95.65%, while reducing verification time from several human working days to a 4.5-hour automated execution.

1 Introduction

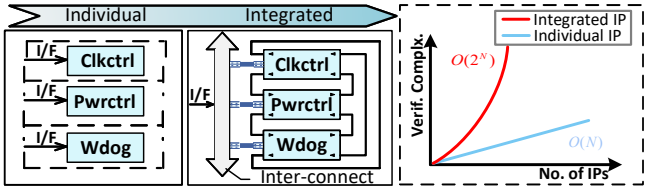
Agile hardware development pushes modern SoC projects toward rapid iteration and frequent design revisions, raising pressure on verification teams to match fast-changing designs. As shown in Fig.1a, verification has become the dominant bottleneck, consuming nearly 70% of the overall front-end cycle [17]. To manage this complexity, industrial flows predominantly rely on UVM. Despite its structure, UVM remains manual in practice, and assembling a full testbench for real designs requires deep engineering expertise.

To reduce this manual burden, companies incorporate template-driven scripting flows on top of UVM[14, 19, 33]. These automate repetitive and well-structured tasks, e.g., directory initialisation, code skeleton generation, or basic register and sequence templates and thus improve consistency. Yet, since these tools expand predefined templates without understanding micro-architectural intent, they cannot infer protocol timing, transaction semantics, driver/-monitor behaviour, or functional constraints. As designs evolve, engineers must manually fill these semantic gaps, and coverage closure continues to depend on expert effort rather than automation.

Recent advances in the LLMs create new opportunities for breaking this bottleneck. Unlike template systems, the LLMs can process natural-language specifications, RTL structure, protocol rules, and design-intent descriptions[5, 16, 24, 26, 29, 35, 41], allowing them to reason about behaviours that traditional scripts cannot capture. Prior works, e.g., MEIC[42] employs the dual fine-tuned LLMs with RTL toolchain, to automate error detection and correction in Verilog; UVLLM[21] integrates the LLMs with UVM methodology, to automate error detection and repair in Verilog code, ensuring comprehensive verification; AssertLLM[44] utilises three customized LLMs to generate SystemVerilog Assertions from specification documents for RTL verification, showing that the LLMs can assist with debugging and localized test generation. Among them, UVM² [46] is the first attempt at UVM automation verification, generating agents, sequences, and basic stimuli end-to-end. Yet, these advances remain confined to the IP level, where design interactions are local and protocol reasoning is relatively contained.



(a) Decomposition of IC front-end design and verification. Functional verification accounts for nearly 70% of verification effort, starting from test planning, with the most time spent on testbench construction and stimuli refinement.



(b) Comparison of verification complexity between individual and integrated IPs. Interactions and constraint couplings cause growth of complexity during integration.

Figure 1. Verification dominates IC front-end development. While module-level complexity scales linearly, subsystem-level verification grows exponentially due to inter-IP dependencies. (Verif.: Verification, TB: testbench, Refi.: Refinement, Constr.:Construction, I/F: Interface, Complx.: Complexity)

Challenges. Extending LLM-aided UVM verification beyond the IP level to an SoC subsystem creates a new set of challenges across the verification pipeline, from constructing the UVM testbench to generating stimuli. Even on a simple subsystem (a P-channel power controller), the framework repeatedly failed to assemble a UVM testbench once several interface signals were absent (Sec. 4.5). Unlike IP-level designs, subsystems present richer bus interfaces and more intricate control paths, documented in lengthy specifications with timing waveforms and block diagrams, which cause an exponential growth in complexity with the number of IP modules, as shown in Fig. 1b. This multi-modal documentation currently exceeds the interpretive capacity of the general-purpose LLMs, making automatic assembly of a correct subsystem-level UVM testbench unreliable.

Even when a UVM testbench is available, the generated stimuli still achieve low coverage, average below 40%. They fail to exercise corner-case behaviours, long dependency chains across multiple IPs,

DAC 2026, Long Beach, USA
2026. ACM ISBN 978-X-XXXX-XXXX-X/XX/XX
https://doi.org/10.1145/3770743.3811404

Junhao Ye[1,2], Dingrong Pan[2], Hanyuan Liu[1,2], Yuchen Hu[1,2], Jie Zhou[1,2], Ke Xu[1,2], Xinwei Fang[3], Xi Wang[1,2], Nan Guan[4], Zhe Jiang[1,2†]

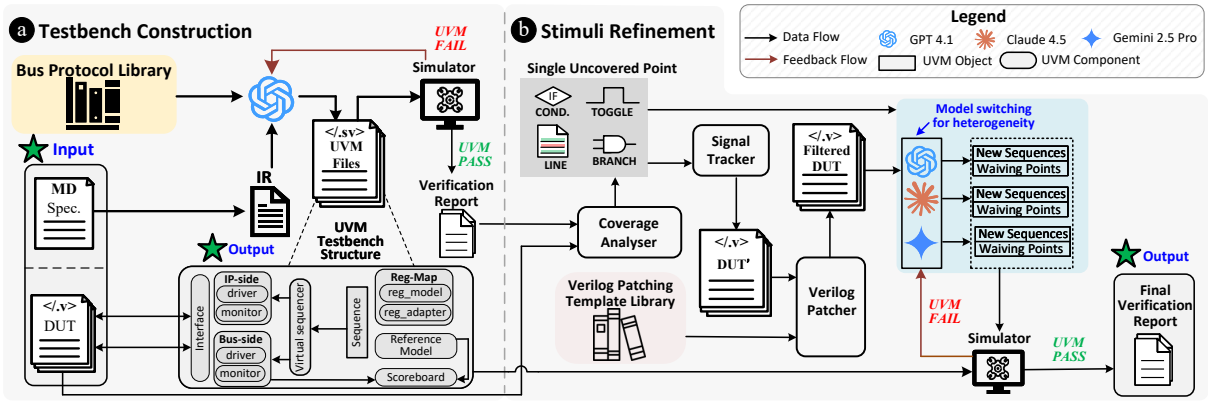


Figure 2. UVMarvel Framework. (a) **Testbench Construction**: the IR translated from design specifications, together with the Bus Protocol Library, guiding the LLMs to construct UVM testbench; and (b) **Stimuli Refinement**: uncovered coverage data are interpreted by the Coverage Analyser, filtered DUT is identified through Signal tracker and Verilog Patcher, and the LLMs generate new stimuli or waiving points to improve coverage.

rare event sequences and intricate state transitions. For example, the subsystem's behaviour often hinges on multi-component sequences (e.g., power request/acknowledgement handshakes) that must be orchestrated step by step, whereas the LLMs tend to emit simple, repetitive patterns that quickly saturate obvious cases while missing deeper corners. Without a clear view of inter-IP dependencies, the LLMs cannot reliably derive the nuanced sequences needed to cover all functional points, leading to early convergence of coverage.

Contributions. We present **UVMarvel**, the first automated, LLM-driven framework for subsystem-level UVM-based verification. To automatically construct subsystem-level UVM testbenches, **UVMarvel** integrates a verification-oriented Intermediate Representation (IR) with a scalable Bus Protocol Library, enabling the LLMs to interpret structural semantics and protocol behaviours across heterogeneous interfaces. For stimuli refinement, **UVMarvel** employs a Signal Tracker that extracts inter-IP dependency paths across modules, and a Verilog Patching Template Library that distils each path into minimal semantic blocks, enabling the LLMs to derive the nuanced multi-step sequences required for coverage improvement. As a unified solution that provides subsystem-level access to both design structure and execution behaviour for the LLMs, **UVMarvel** achieves industrial-grade coverage of 95.65%. The Bus Protocol Library and Verilog Patching Template Library are open-sourced at https://github.com/SEU-ACAL/reproduce-UVMarvel-DAC-26.

2 UVMarvel: An Overview

Aiming to accelerate the verification loop for subsystem-level RTL, UVMarvel integrates LLM assistance into a standard UVM workflow and organises the process into two stages, as shown in Fig. 2. The framework takes specifications and RTL as inputs and constructs a complete UVM testbench with final verification reports that include error logs, waiving candidates, and coverage results.

Stage (a) aims to construct testbenches. Information about the subsystem is often dispersed across specifications and RTL, making it difficult for the LLMs to understand the structure and interface behaviours. To provide a unified view, we convert these materials into an IR that summarises modules, connections and interface roles. Since the subsystem's behaviour further depends on protocol rules that the LLMs cannot easily extract, a Bus Protocol Library is supplied to give precise transaction guidance. With these supports, the LLMs generate the UVM testbenches, executed with the DUT to produce the first verification report.

Stage (b) aims to improve coverage. The problem is that initial stimuli typically fail to exercise deep, multi-IP interactions and long dependency chains, causing early coverage convergence. To address this, the coverage analyser decomposes the report into single uncovered points. Each point is examined by the Signal Tracker, which identifies the relevant signal path in the RTL. A patching step then patches the Verilog structure to produce a compact slice that preserves the essential dependency chain to generate a filtered DUT (a DUT that has only key path-related code). This filtered DUT, combined with the uncovered point, provides the context needed for the LLMs to generate additional stimuli or justified waiving candidates. The new stimuli are added, and the testbench is re-executed on the original DUT to improve coverage, while also producing the final verification report.

3 UVMarvel: The Framework Pipeline

3.1 Intermediate Representation

Verification in practice typically begins with a test plan (Fig. 1a), where engineers interpret the specification and decompose the design intent into concrete verification points. Existing LLM-based approaches, yet, devote little effort to this planning stage [3, 8, 43].

From the perspective of a UVM testbench, what we need is information that is much more concrete than what the specification exposes: (i) for the environment hierarchy (`uvm_env`, `uvm_agent`, `uvm_driver`, `uvm_monitor`), it must know how the DUT is instantiated and how it communicates with the outside system; (ii) for the `uvm_reg_block` and its adapter, it needs the programmable state (which registers exist, where they are mapped, how they are reset and accessed); and (iii) for sequence generation, it needs timing assumptions on the interfaces and the key functional scenarios and corner cases that should be exercised. In practice, human verification engineers gather exactly this information before they start writing UVM code or a test plan.

Guided by this observation and standard verification methodology documents [2, 4, 7, 36], we construct a five-part IR with components *Module Name*, *Interface Description*, *Register Configuration*, *Timing Characteristics* and *Functional Description*, as shown in Fig. 3. This IR distils the information needed for UVM environment, agent, register model and scenario construction into a verification-centred description that the LLMs can use directly.

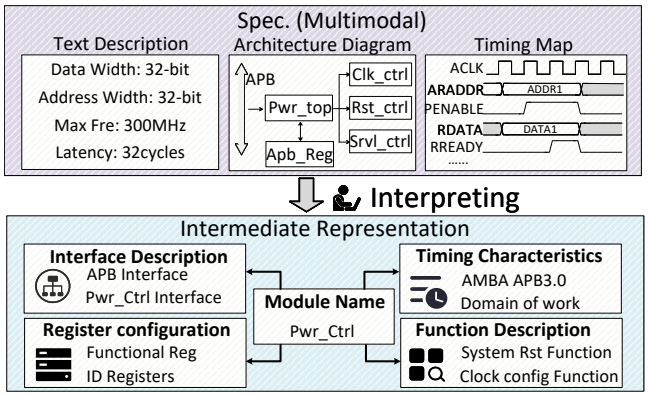


Figure 3. Multimodal specification vs. unified IR. IR condenses diagrams, waveforms and textual descriptions into verification-relevant fields.

3.2 Bus Protocol Library

When constructing a subsystem-level UVM testbench, bus-side components, e.g., drivers, monitors, agents and interfaces are indispensable: without them, the subsystem cannot interconnect its IP blocks or exercise its behaviour end-to-end. Industrial verification IP (VIP) packages protocol rules into reusable driver, monitor and checker [20, 30, 38], but these VIPs neither synthesise new, design-specific bus components from heterogeneous specifications nor expose protocol structure in a form that the LLMs can adapt. [1]

Mainstream bus protocols encode rich ordering rules, handshake relations and timing dependencies, and their official specifications are long and heterogeneous, often mixing timing diagrams, waveforms and descriptive text. Even with these documents available, current LLMs struggle to turn protocol manuals into correct UVM components, due to limited context, long-range dependencies and multi-modal artefacts [9]. At the same time, these protocols rely on a small set of regular behavioural patterns, and the LLMs are effective at reading and modifying structured code when guided by constraints. This motivates representing protocol behaviours as a concise set of human-readable UVM skeletons that capture the stable request/response and timing rules, and then asking the LLMs to specialise these skeletons to each DUT.

Based on this idea, we built a Bus Protocol Library to assist our framework in generating bus-side UVM components. For each supported protocol, the library provides UVM skeletons for the interface, driver, monitor and agent. During generation, the framework uses the protocol information in the IR to select the skeletons and prompts the LLMs to specialise them with the remaining IR details, such as signal names, groupings, widths and address ranges, as illustrated in Fig. 4. To keep protocol semantics consistent across designs, the LLMs are allowed to modify only designated regions of the skeletons (for example, configuration parameters), while the protocol control flow and handshake structure remain fixed. In this way, the protocol semantics come from the library, and the LLMs mainly adapt them to each DUT instance.

3.3 Coverage Analyser

After the UVM testbench has been constructed with the assistance of IR and the Bus Protocol Library, the verification flow proceeds to simulation. Each run produces a coverage report, typically as large

[1] In our preliminary SoC experiments(Sec.4.5), a general-purpose LLM asked to construct bus-side components directly repeatedly failed to assemble a correct subsystem-level UVM testbench once some interface signals were omitted.

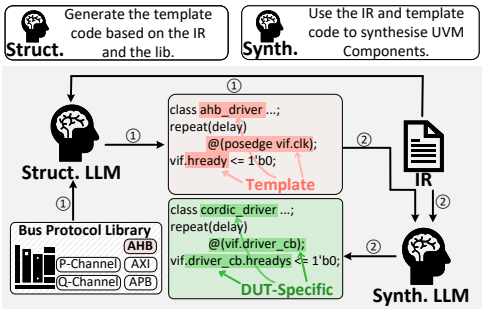


Figure 4. Generation of UVM bus components using the Bus Protocol Library, illustrated with AHB_Driver. It selects a protocol-specific UVM skeleton, then the LLM specialises it into DUT-specific code that is integrated as a key bus component.(Struct.: Structural, Synth.: Synthetic)

HTML pages that mix coverage data with UI markup. This format is convenient for human browsing but unsuitable for the LLMs, since most tokens are layout noise and the useful coverage items are scattered across many sections. To make coverage feedback usable, we introduce a Coverage Analyser that converts these reports into a compact, task-oriented summary.

The analyser takes the HTML coverage report as input and parses the coverage categories. For each category, it extracts all uncovered or partially covered items plus essential context such as hierarchical name and source file. It then emits these items as structured text grouped by coverage type and module, and feeds them to the LLMs. This strips away UI boilerplate and surfaces only uncovered targets, so the LLMs can focus on generating additional stimuli to close the remaining gaps.

3.4 Signal Tracker

After coverage feedback has been collected, UVMarvel analyzes why certain coverage points remain uncovered. In practice, the LLM-generated stimuli often leave many deep corner cases untested, resulting in low coverage. The missing cases are usually deep corners whose activation depends on long signals chains across several IPs and many cycles, such as handshakes. Hitting these points requires driving the right signals in a specific order and times. In subsystem-level RTL, the relevant assignments and conditions are scattered over many files and always blocks, so the LLMs cannot easily see how an uncovered signal is connected back to controllable inputs.

Existing work such as UVLLM [21], inspired by STRIDER [45], uses AST-based dependency trees to locate important signals, but these trees grow large on subsystem-level RTL and are not centred on any specific coverage point. We instead follow how an engineer debugs coverage: start from the uncovered signal, collect the statements that define or use it, and trace backwards to the top-level

Algorithm 1: Single-File Signal Tracing

Input: Target signals S_0, Verilog file F
Output: Relevant statement set $\mathcal{R}$
$Q \leftarrow S_0, V \leftarrow \emptyset, \mathcal{R} \leftarrow \emptyset$;
while $Q \neq \emptyset$ **do**
  $s \leftarrow \text{dequeue}(Q)$;
  if $s \in V$ **then** continue;
  $V \leftarrow V \cup \{s\}$;
  foreach *statement* t *in* F *referencing* s **do**
    $\mathcal{R} \leftarrow \mathcal{R} \cup \{t\}$;
    $X \leftarrow$ signals in $t \setminus \{s\}$;
    foreach $x \in X \setminus V$ **do** enqueue(Q, x);
  end
end
return $\mathcal{R}$

Junhao Ye[1,2], Dingrong Pan[2], Hanyuan Liu[1,2], Yuchen Hu[1,2], Jie Zhou[1,2], Ke Xu[1,2], Xinwei Fang[3], Xi Wang[1,2], Nan Guan[4], Zhe Jiang[1,2†]

Algorithm 2: Cross-File Recursive Tracing for Subsystems

Input: Target signals S_0, submodule files $\{F_1, \ldots, F_k\}$, top-level file F_{top}
Output: Global dependency set $\mathcal{G}$

$S \leftarrow S_0$, $i \leftarrow 0$, $\mathcal{G} \leftarrow \emptyset$;
repeat
  $i \leftarrow i + 1$, $E \leftarrow \emptyset$;
  foreach *submodule file* F_j **do**
    $V_j \leftarrow$ Algorithm 1(S, F_j);
    $\mathcal{G} \leftarrow \mathcal{G} \cup V_j$;
    extract I/O signals of F_j from V_j and add to E;
  end
  $S \leftarrow \text{unique}(E)$;
until $S = \emptyset$;
Run Algorithm 1 on F_{top} using $\mathcal{G}$ and keep all I/O ports of F_{top};
return $\mathcal{G}$

I/Os. The Signal Tracker automates this process: it takes as input a set of seed signals S_0 from uncovered coverage expressions and the subsystem-level Verilog files, and produces a compact cross-file dependency slice $\mathcal{G}$ together with the top-level I/O ports through which a testbench can drive these signals.

Single-file tracing. Algorithm 1 works on a Verilog file F_j. Given the current seed set S, it finds all statements that read from or assign to any signal in S, adds those statements to $\mathcal{R}$, and pushes any newly seen signals into the queue. The resulting set V_j is a small subset of RTL directly related to S within file F_j.

Subsystem-level designs usually span many `.v` files, and important chains often cross module boundaries. We therefore extend the tracing across files using a simple recursive expansion:

Cross-file expansion. Starting from S_0, we apply Algorithm 2 to each submodule file F_j and merge the fragments V_j into the global set $\mathcal{G}$. From each V_j we keep only the submodule input and output ports, collect them into a new seed set S, and repeat until no new interface signals are found. Finally, we run Algorithm 1 on the top-level file F_{top} using the accumulated dependencies in $\mathcal{G}$ and keep all top-level I/O ports as legal stimulus entry points.

At this stage, the tracker has identified the RTL statements and signals related to each uncovered coverage point and linked them to controllable top-level interfaces. These statements are still fragments, scattered across files and missing their original module and process context. In the next step, we reconstruct a small amount of Verilog structure around them so that they form a coherent view of the design that can be fed to the LLMs.

3.5 Verilog Patcher

As described in Section 3.4, the Signal Tracker identifies RTL statements and signals causally connected to each uncovered coverage point and links them to the relevant top-level interfaces. However, once lifted from their original files, these statements are scattered across different always blocks and modules and often lose the headers and surrounding control that make them valid Verilog. Although LLMs can tolerate some incompleteness and edit partially structured programs [23, 47], they still need basic syntactic units to understand control and data flow.

To restore this minimal structure, we introduce the Verilog Patching Template Library and a Patcher that uses it. The library provides a small set of patterns for canonical RTL constructs, including module shells, always blocks, case blocks, continuous assignments and instance-level connections, each preserving only the essential syntactic form. During patching, statements that originated from the same construct are grouped and attached to the corresponding template. For example, if the tracker collects branch assignments from a case block but the surrounding `case`/`endcase` are missing, the Patcher applies a case-block template to rebuild the header and closing keyword; if an assignment was originally inside an `always` block but is now isolated, the Patcher uses an always-block template to reintroduce the minimal process wrapper.

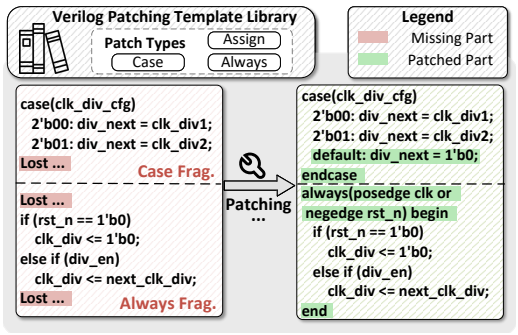


Figure 5. Verilog Patching Process. The library reconstructs incomplete code fragments into valid blocks based on their syntax types. In the example, it appends missing endcase and `default` keywords to terminated case statements and wraps isolated logic fragments within `always` blocks containing appropriate sensitivity lists.

The workflow is as follows. We take the key statements reported by the Signal Tracker, and the Verilog Patcher inspects the surrounding RTL to determine whether each fragment came from a case block, always block or module shell, then selects a matching template from the library. It fills in the missing structure to produce a patched version of the code, namely the Filtered DUT. As illustrated in Fig. 5, the left side shows an incomplete case and always fragments (in red), while the right side shows the patched version with restored `default`/`endcase` keywords and an enclosing `always` block with an appropriate sensitivity list. This Filtered DUT is then presented to the LLMs in the stimuli refinement stage.

3.6 LLM-Guided Sequence Execution

In a UVM testbench, the sequence is the execution vehicle of stimuli: it decides which transactions are applied to the DUT and in what order and timing. In our framework, the LLMs reason on the Filtered DUT and uncovered points, but all generated sequences are instantiated and run on the original subsystem-level UVM testbench, so coverage is always measured on the original DUT.

To avoid relying on a single model's judgment, we employ three different LLMs for coverage analysis and sequence generation, which helps reduce bias and explore a broader stimulus space. Given the Filtered DUT and uncovered items, the LLMs propose candidate sequences that are instantiated on the original DUT and simulated. During this loop, the models are also asked to flag coverage points that appear intrinsically unreachable; these are collected as waiver candidates and can be marked as waived in the final coverage report. If a candidate sequence causes compilation or simulation errors, the error messages are fed back to the LLMs to repair or discard the sequence and try again, closing the feedback loop of our framework.

4 Evaluation

Setup. We evaluate **UVMarvel** using LLM agents deployed through the ChatGPT API, with GPT-4.1 as the default model and Claude 4.5/Gemini 2.5 pro as comparative baselines. All generated testbenches are compiled and simulated using Synopsys VCS.

4.1 Evaluation Metrics

Success Rate of Generation (SRG). This metric measures how often a generated UVM testbench is both syntactically valid and

Table 1. Benchmark designs used for evaluation, covering diverse hardware modules across multiple on-chip interfaces, including APB-based watchdog and power-control units, an AHB CORDIC accelerator, Q-Channel and P-Channel low-power controllers, and an AXI-based interface remap block.

Design Name	Protocol	Description	Module Counts	Line Counts
Watchdog	APB	detects failures via a programmable counter, triggers an interrupt, and resets if ignored, with APB-configurable locked registers.	3	1500+
Pwrctrl	APB	provides APB register control for SCP power, clock, and reset signal management.	9	2000+
Cordic	AHB	provides hardware acceleration for trigonometric and square root calculations.	3	1100+
IdleControl	Q-Channel	manages AXI and DMA interfaces, entering idle/stop states and accepting or rejecting low-power requests based on interface activity.	3	800+
LPctrl	P-Channel	enables safe CDC and low-power data exchange by using a low-power channel.	8	1500+
Busremap	AXI	converts master-slave signals, providing timing isolation, synchronisation, and transaction tracing for secure, efficient data exchange.	10	3000+

Table 2. Code coverage and functional coverage for each design.

Design	Code Coverage (%)	Functional Coverage (%)
Watchdog	98.84	100
Pwrctrl	93.66	90.64
Cordic	100	100
IdleControl	94.90	96.12
LPctrl	90.83	89.33
Busremap	95.66	98.27

functionally correct. For each design, we let the LLMs generate N_{total} testbenches. A generation is counted as successful if (i) the testbench completes the full VCS compilation and simulation flow without errors, and (ii) under the same regression stimuli, all checkers pass and the observed outputs match those of a reference constructed by experienced verification engineers. The success rate is

$$\text{SRG} = \frac{N_{\text{correct}}}{N_{\text{total}}} \times 100\%. \quad (1)$$

Coverage. After a testbench successfully runs on VCS, we report code coverage and functional coverage collected by the simulator.

Code coverage includes:

- **Score**: aggregate code coverage score for a design.
- **Line**: each executable line is hit at least once.
- **Branch**: both outcomes of each branch are taken.
- **Condition**: each Boolean expression is exercised.
- **Toggle**: each signal bit switches between 0 and 1.

Functional coverage is measured by user-defined covergroups and coverpoints that track whether specified scenarios, transactions and state combinations have been exercised. [2]

4.2 Benchmark

Unlike IP-level benchmarks such as RTLLM [28], Verilog-Eval [25] and UVM² [46], operating on isolated RTL blocks, our evaluation is based on industrial subsystem-level designs, reflecting genuine verification challenges. As summarised in Table 1, the benchmarks span heterogeneous IPs connected through APB, AHB, AXI, P-Channel, and Q-Channel interfaces.

4.3 Overall Framework Effectiveness

Execution time. UVMarvel covers both UVM testbench construction and stimuli refinement. Starting from a human-provided IR and running until code coverage reaches 90%, our framework takes 4.5 hours on the benchmark subsystem, which is 20.17× faster than the manual verification flow, as shown in Fig. 6. In other words, subsystem-level verification tasks that previously required several working days of human-driven UVM development and regression are now completed within an hour-scale automated run.

SRG of testbench generation. UVMarvel achieves an overall SRG of 93.33% across all subsystem components: only AXI-based subsystems fail, mainly due to incorrect coordination between AXI channels (e.g., inconsistent AW/W/B handshakes). In contrast, existing testbench-generation work (e.g., MEIC[42] and UVM²[46]) achieves 0% SRG at it, because these methods are limited to IP-level RTL and cannot handle register configuration, bus transactions, and other issues that arise only in subsystem-level RTL.

Coverage. UVMarvel reaches 95.65% average code coverage. We compare against two automated stimuli-generation baselines: MEIC[42], relying on random stimuli, and UVM²[46], activating LLM capabilities through structured prompts and templates. As both lack bus-protocol knowledge, we provide them with a pre-constructed subsystem-level UVM testbench and Bus Protocol Library used by UVMarvel so that the comparison isolates the stimuli generation capability. As shown in Fig. 7, UVMarvel achieves higher code coverage on all metrics in our benchmark.

4.4 Impact of IR on Stage ⓐ

We evaluate the impact of the IR by comparing two settings: giving the LLMs the original specification and giving them the IR. As shown in Fig. 8, using the IR improves coverage by nearly 30% on average, indicating that the IR is more than a simple paraphrase of the specification. On closer inspection, we observe that in the spec-only setting, the LLMs often instantiate incomplete environments, for example, omitting bus monitors or failing to connect register models to the corresponding bus agents, so that large parts of the

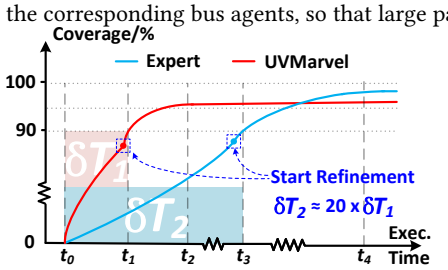


Figure 6. End-to-end verification time: UVMarvel vs. experts (across all benchmarks). We exclude the test planning/IR authoring phase (0-t_0) from time evaluation to ensure precision—although this slightly lowers the acceleration ratio- to aligns UVMarvel's automated verification objectives with manually defined coverage goals, guaranteeing fair and objective experimental results. UVMarvel achieves 90% coverage nearly twenty times faster than manual verification and converges to an industrial-grade final coverage level comparable to the expert flow. (t_0: end of IR/test planning, t_1: 90% coverage by **UVMarvel**, t_2: coverage closure by **UVMarvel**, t_3: 90% coverage by experts, t_4: coverage closure by experts, δT_1: time for **UVMarvel** to reach 90% coverage, δT_2: time for experts to reach 90% coverage)

[2]We focus on code coverage in our comparisons because it is defined directly on the RTL structure and is therefore comparable across designs and methods. Functional coverage instead depends on engineer-defined covergroups and reflects project-specific intent; in practice, it is only examined after code coverage exceeds about 90%, and even then remains subjective. To keep our evaluation objective, all comparative and ablation studies use code coverage as the primary metric, and unless otherwise stated **coverage** refers to code coverage; we still report functional coverage for completeness in Table 2, using covergroups written by experienced engineers.

Junhao Ye[1,2], Dingrong Pan[2], Hanyuan Liu[1,2], Yuchen Hu[1,2], Jie Zhou[1,2], Ke Xu[1,2], Xinwei Fang[3], Xi Wang[1,2], Nan Guan[4], Zhe Jiang[1,2†]

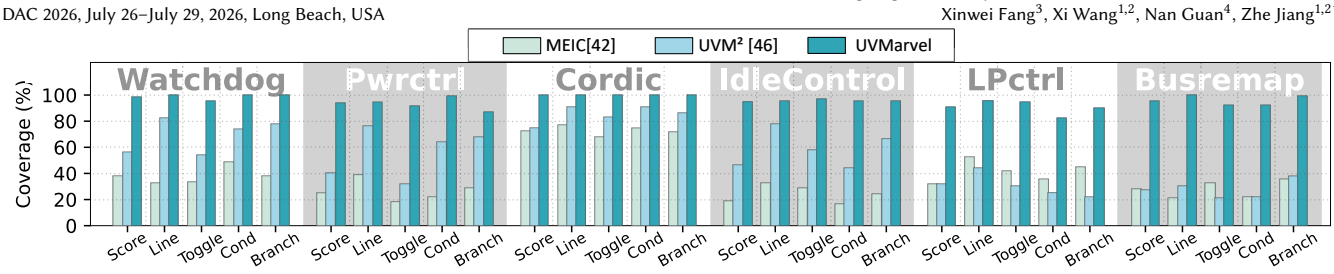


Figure 7. Code coverage of MEIC [42], UVM² [46], and UVMarvel across six benchmark tests, evaluated using five coverage components.

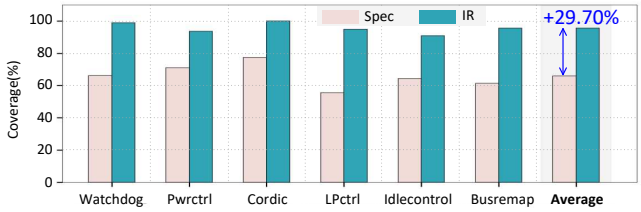


Figure 8. Code coverage comparison between IR and SPEC inputs across six benchmark tests, showing higher coverage achieved by IR inputs.

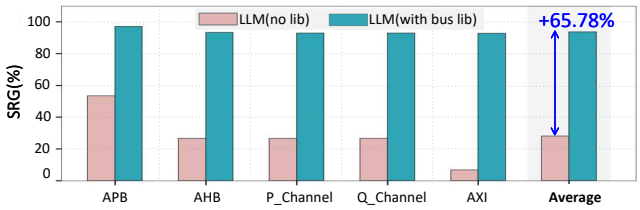


Figure 9. Comparison of SRG performance with and without the Bus Protocol Library, validated over five different bus protocols.

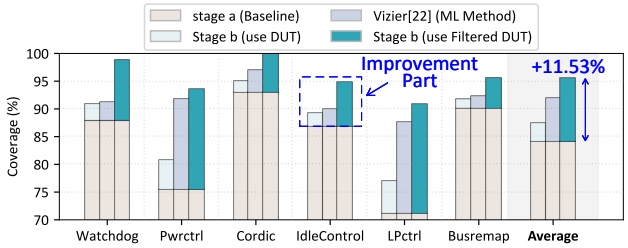


Figure 10. Coverage improvement achieved by the three methods across six benchmark tests, with the filtered DUT getting the highest performance.

DUT are never driven. By presenting design intent in a predictable, composable format, the IR helps the model infer handshake directions, transaction ordering and interface semantics, leading to more complete and effective subsystem-level UVM testbenches.

4.5 Impact of Bus Protocol Library on Stage ⓐ

We further evaluate the role of the Bus Protocol Library by comparing two settings: (1) prompting the LLMs without the library, and (2) supplying the library as guidance. In both cases, all non-bus UVM components are fixed and provided as context; the LLMs are only asked to generate the bus-side components.

As shown in Fig. 9, providing the Bus Protocol Library improves the SRG by 65.78%, demonstrating that the LLMs struggle to infer protocol behaviours directly from specifications. The challenge is particularly acute for AXI, where handshake patterns and decoupled channels introduce nontrivial ordering, response, and backpressure constraints. Without guidance, the success rate for AXI falls below 7%; with the protocol library, it exceeds 90%.

4.6 Impact of Signal Tracker & Verilog Patcher on Stage ⓑ

We evaluate the effect of the Signal Tracker and Verilog Patcher by measuring score coverage improvement during refinement. Prior work has shown that machine learning can assist in exploring design and parameter spaces [1, 6, 10–13, 15, 18, 22, 27, 31, 32, 34, 37, 39, 40], so we also compare against a representative ML-based optimisation approach. Concretely, we adopt Google's open-source Vizier framework [22] as an ML-based refinement method.

Using the coverage of Stage ⓐ as the baseline, we compare three refinement strategies: feeding the LLMs the full DUT, refining stimuli with Vizier, and feeding the LLMs the Filtered DUT. As shown in Fig. 10, the Filtered DUT achieves the largest average gain (11.53%),

Table 3. Code coverage achieved by different LLMs.

Design Name	GPT4.1	Claude4.5	Gemini2.5pro
Watchdog	98.70	98.70	98.84
Pwrctrl	93.40	93.66	93.66
Cordic	100.00	100.00	100.00
IdleControl	94.90	94.80	94.60
LPctrl	90.70	90.70	90.83
Busremap	95.40	95.66	95.60

while Vizier [22] and the raw DUT yield only 6.66% and 3.35%, respectively. These results lead to two observations. (1) When applied directly to the full subsystem-level RTL, the LLMs make limited progress in improving coverage; even though Vizier [22] requires many optimisation iterations and computation, it copes better with the large search space than unguided LLM refinement. (2) Once we provide compact and dependency-preserving context through the Filtered DUT, the LLMs surpass both the raw-DUT and ML baselines in coverage gain, suggesting that the LLMs have potential for verification when combined with appropriate structural guidance.

4.7 Code coverage achieved by different LLMs

To reduce the risk of relying on a single model that might miss corner cases, Stage ⓑ employs three different LLMs when generating new sequences. As shown in Table 3, the three models achieve very similar coverage under the same IR, protocol scaffolding and Filtered DUT. This convergence across models suggests that UVMarvel is robust to the choice of the LLMs; its coverage improvement remains stable even when the underlying model changes.

5 Conclusion

We have presented UVMarvel , the first UVM-based verification framework for subsystem-level RTL. By combining the IR with the Bus Protocol Library, UVMarvel automatically constructs subsystem-level UVM testbenches, and by using the Signal Tracker together with the Verilog Patching Library, it further boosts score coverage to 95.65% with a 4.5-hour automated execution.

6 Acknowledgement

We appreciate the reviewers for their helpful feedback. This work is supported by the National Key Research and Development Program (Grant No.2024YFB4405600), the Basic Research Program of Jiangsu (Grants No. BK20243042), and the Fundamental Research Funds for the Central Universities (No. 2242025K20013).

References

[1] Mohamed A Abd El Ghany and Khaled A Ismail. 2021. Speed up functional coverage closure of cordic designs using machine learning models. In *2021 International Conference on Microelectronics (ICM)*. IEEE, 91–95.

[2] Accellera Systems Initiative 2015. *Universal Verification Methodology (UVM) 1.2 Reference Manual*. Accellera Systems Initiative.

[3] Berk Berabi et al. 2024. LLM4HW: From Natural Language to Verilog Generation. In *Proceedings of the 61st ACM/IEEE Design Automation Conference (DAC)*. ACM, 1–6.

[4] Janick Bergeron. 2000. *Writing Testbenches: Functional Verification of HDL Models*. Springer.

[5] Jitendra Bhandari, Johann Knechtel, Ramesh Narayanaswamy, Siddharth Garg, and Ramesh Karri. 2024. Llm-aided testbench generation and bug detection for finite-state machines. *arXiv preprint arXiv:2406.17132* (2024).

[6] Harsh Bhargav, Vineesh Vs, Binod Kumar, and Virendra Singh. 2021. Enhancing testbench quality via genetic algorithm. In *2021 IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*. IEEE, 652–656.

[7] Cadence Design Systems 2019. *Metric-Driven Verification Methodology User Guide*. Cadence Design Systems.

[8] Guanlan Chen et al. 2024. LLM4DV: Large Language Models for Design and Verification. In *2024 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–9.

[9] Hong Chen, Xin Wang, Yuwei Zhou, Bin Huang, Yipeng Zhang, Wei Feng, Houlun Chen, Zeyang Zhang, Siao Tang, and Wenwu Zhu. 2024. Multi-modal generative ai: Multi-modal llm, diffusion and beyond. *arXiv preprint arXiv:2409.14993* (2024).

[10] Jingyi Chen, Lei Yan, Shikai Wang, and Wenxuan Zheng. 2024. Deep reinforcement learning-based automatic test case generation for hardware verification. *Journal of Artificial Intelligence General science (JAIGS) ISSN: 3006-4023* 6, 1 (2024), 409–429.

[11] Hyojin Choi, In Huh, Seungju Kim, Jeonghoon Ko, Changwook Jeong, Hyeonsik Son, Kiwon Kwon, Joonwan Chai, Younsik Park, Jaehoon Jeong, et al. 2021. Application of deep reinforcement learning to dynamic verification of dram designs. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 523–528.

[12] Gabriel Mihail Danciu and Alexandru Dinu. 2022. Coverage fulfillment automation in hardware functional verification using genetic algorithms. *Applied Sciences* 12, 3 (2022), 1559.

[13] Siddhanth Dhodhi, Debarshi Chatterjee, Eric Hill, and Saad Godil. 2021. Deep stalling using a coverage driven genetic algorithm framework. In *2021 IEEE 39th VLSI Test Symposium (VTS)*. IEEE Computer Society, 1–4.

[14] Jaideep Varier EV, V Prabakar, and Karthigha Balamurugan. 2019. Design of generic verification procedure for IIC protocol in UVM. In *2019 3rd International conference on Electronics, Communication and Aerospace Technology (ICECA)*. IEEE, 1146–1150.

[15] Martin Fajcik, Pavel Smrz, and Marcela Zachariasova. 2017. Automation of processor verification using recurrent neural networks. In *2017 18th International Workshop on Microprocessor and SOC Test and Verification (MTV)*. IEEE, 15–20.

[16] W Fang, M Li, M Li, Z Yan, S Liu, H Zhang, and Z Xie. [n. d.]. AssertLLM: Generating and Evaluating Hardware Verification Assertions from Design Specifications via Multi-LLMs. arXiv 2024. *arXiv preprint arXiv:2402.00386* ([n. d.]).

[17] Harry Foster. 2020. Wilson research group functional verification study: IC/ASIC functional verification trend report. *Wilson Research Group and Mentor, A Siemens Business, White Paper* (2020).

[18] Deepak Narayan Gadde, Thomas Nalapat, Aman Kumar, Djones Lettnin, Wolfgang Kunz, and Sebastian Simon. 2024. Efficient stimuli generation using reinforcement learning in design verification. In *2024 20th International Conference on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD)*. IEEE, 1–4.

[19] Nikolaos Georgoulopoulos and Alkiviadis Hatzopoulos. 2019. UVM-based verification of a digital PLL using systemverilog. In *2019 29th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*. IEEE, 23–28.

[20] Stepan Harutyunyan, Taron Kaplanyan, Artak Kirakosyan, and Haykaram Khachatryan. 2020. Configurable verification IP for UART. In *2020 IEEE 40th international conference on electronics and nanotechnology (ELNANO)*. IEEE, 234–237.

[21] Yuchen Hu, Junhao Ye, Ke Xu, Jialin Sun, Shiyue Zhang, Xinyao Jiao, Dingrong Pan, Jie Zhou, Ning Wang, Weiwei Shan, et al. 2024. Uvllm: An automated universal rtl verification framework using llms. *arXiv preprint arXiv:2411.16238* (2024).

[22] Qijing Huang, Hamid Shojaei, Fred Zyda, Azade Nazi, Shobha Vasudevan, Sat Chatterjee, and Richard Ho. 2022. Test parameter tuning with blackbox optimization: A simple yet effective way to improve coverage. In *Proceedings of the design and verification conference and exhibition US (DVCon)*.

[23] Kensen Li, Uri Alon, Alessio Parisi, and Richard Sutton. 2024. Large Language Models Are Zero-Shot Program Synthesizers. *Transactions on Machine Learning Research* (2024).

[24] Mengming Li, Wenji Fang, Qijun Zhang, and Zhiyao Xie. 2025. Specllm: Exploring generation and review of vlsi design specification with large language model. In *2025 International Symposium of Electronics Design Automation (ISEDA)*. IEEE, 749–755.

[25] Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. 2023. Verilogeval: Evaluating large language models for verilog code generation. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 1–8.

[26] Shang Liu, Wenji Fang, Yao Lu, Jing Wang, Qijun Zhang, Hongce Zhang, and Zhiyao Xie. 2024. Rtlcoder: Fully open-source and efficient llm-assisted rtl code generation technique. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2024).

[27] Yuntao Lu, Chen Bai, Yuxuan Zhao, Ziyue Zheng, Yangdi Lyu, Mingyu Liu, and Bei Yu. 2025. DeepVerifier: Learning to Update Test Sequences for Coverage-Guided Verification. *ACM Transactions on Design Automation of Electronic Systems* (2025).

[28] Yao Lu, Shang Liu, Qijun Zhang, and Zhiyao Xie. 2023. RTLLM: An open-source benchmark for design rtl generation with large language model. *arXiv preprint arXiv:2308.05345* (2023).

[29] Karthik Maddala, Bhabesh Mali, and Chandan Karfa. 2024. Laag-rv: Llm assisted assertion generation for rtl design verification. In *2024 IEEE 8th International Test Conference India (ITC India)*. IEEE, 1–6.

[30] Vazgen Melikyan, Stepan Harutyunyan, Artak Kirakosyan, and Taron Kaplanyan. 2021. Uvm verification ip for axi. In *2021 IEEE East-West Design & Test Symposium (EWDTS)*. IEEE, 1–4.

[31] Nurun Nahar Mondol, Arash Vafei, Kimia Zamiri Azar, Farimah Farahmandi, and Mark Tehranipoor. 2024. RL-TPG: automated pre-silicon security verification through reinforcement learning-based test pattern generation. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1–6.

[32] Eric Ohana. 2023. Closing functional coverage with deep reinforcement learning: A compression encoder example. *San Jose, USA* (2023).

[33] TM Pavithran and Ramesh Bhakthavatchalu. 2017. UVM based testbench architecture for logic sub-system verification. In *2017 International Conference on Technological Advancements in Power and Energy (TAP Energy)*. IEEE, 1–5.

[34] Amer Samarah, Ali Habibi, Sofiene Tahar, and Nawwaf Kharma. 2006. Automated coverage directed test generation using a cell-based genetic algorithm. In *2006 IEEE International High Level Design Validation and Test Workshop*. IEEE, 19–26.

[35] Yu-An Shih, Annie Lin, Aarti Gupta, and Sharad Malik. 2025. FLAG: Formal and LLM-assisted SVA Generation for Formal Specifications of On-Chip Communication Protocols. *arXiv preprint arXiv:2504.17226* (2025).

[36] Chris Spear and Greg Tumbush. 2012. *SystemVerilog for Verification: A Guide to Learning the Testbench Methodology*. Springer.

[37] SL Tweehuysen, GLA Adriaans, and M Gomony. 2023. Stimuli generation for ic design verification using reinforcement learning with an actor-critic model. In *2023 IEEE European Test Symposium (ETS)*. IEEE, 1–4.

[38] Simone Vagaggini, Marco Trafeli, Roberto Ciardi, Daniele Davalle, Lucana Santos, Pietro Nannipieri, and Luca Fanucci. 2022. SpaceWire Codec VIP: An innovative architecture of UVM-based Verification Environment: SpaceWire Test and Verification, Short Paper. In *2022 International SpaceWire & SpaceFibre Conference (ISC)*. IEEE, 1–4.

[39] Shobha Vasudevan, Wenjie Joe Jiang, David Bieber, Rishabh Singh, C Richard Ho, Charles Sutton, et al. 2021. Learning semantic representations to verify hardware designs. *Advances in Neural Information Processing Systems* 34 (2021), 23491–23504.

[40] Shikai Wang, Jingyi Chen, Lei Yan, and Zuwei Shui. 2025. Automated test case generation for chip verification using deep reinforcement learning. *Journal of Knowledge Learning and Science Technology ISSN: 2959-6386 (online)* 4, 1 (2025), 1–12.

[41] Yonghao Wang, Jiaxin Zhou, Hongqin Lyu, Zhiteng Chao, Tiancheng Wang, and Huawei Li. 2025. DeepAssert: An LLM-Aided Verification Framework with Fine-Grained Assertion Generation for Modules with Extracted Module Specifications. *arXiv preprint arXiv:2509.14668* (2025).

[42] Ke Xu, Jialin Sun, Yuchen Hu, Xinwei Fang, Weiwei Shan, Xi Wang, and Zhe Jiang. 2025. MEIC: Re-thinking RTL Debug Automation using LLMs. In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design* (Newark Liberty International Airport Marriott, New York, NY, USA) *(ICCAD '24)*. Association for Computing Machinery, New York, NY, USA, Article 100, 9 pages. doi:10.1145/3676536.3676801

[43] Zhenyuan Xu et al. 2024. SpecLLM: Exploring Generation and Understanding of Hardware Specifications with Large Language Models. *arXiv preprint arXiv:2402.17733* (2024).

[44] Zhiyuan Yan, Wenji Fang, Mengming Li, Min Li, Shang Liu, Zhiyao Xie, and Hongce Zhang. 2025. Assertllm: Generating hardware verification assertions from design specifications via multi-llms. In *Proceedings of the 30th Asia and South Pacific Design Automation Conference*. 614–621.

[45] Deheng Yang, Jiayu He, Xiaoguang Mao, Tun Li, Yan Lei, Xin Yi, and Jiang Wu. 2023. STRIDER: Signal value transition-guided defect repair for HDL programming assignments. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 43, 5 (2023), 1594–1607.

[46] Junhao Ye, Yuchen Hu, Ke Xu, Dingrong Pan, Qichun Chen, Jie Zhou, Shuai Zhao, Xinwei Fang, Xi Wang, Nan Guan, et al. 2025. From Concept to Practice: an Automated LLM-aided UVM Machine for RTL Verification. *arXiv preprint arXiv:2504.19959* (2025).

[47] Peng Yin, Marc Brockschmidt, and Miltiadis Allamanis. 2023. CodeTransForm: Evaluating and Improving Code Understanding Capabilities of Large Language Models. *arXiv preprint arXiv:2310.03001* (2023).