This is a repository copy of *UVLLM:An Automated Universal RTL Verification Framework using LLMs*.

White Rose Research Online URL for this paper:
https://eprints.whiterose.ac.uk/id/eprint/225386/

Version: Accepted Version

UVLLM: An Automated Universal RTL Verification Framework using LLMs

Yuchen Hu[1,2], Junhao Ye[1,2], Ke Xu[1,2], Jialin Sun[1,2], Shiyue Zhang[1,2], Xinyao Jiao[1,2], Dingrong Pan[1], Jie Zhou[1,2], Ning Wang[3], Weiwei Shan[1,2], Xinwei Fang[4], Xi Wang[1,2], Nan Guan[3], Zhe Jiang[1,2]

[1]National Center of Technology Innovation for EDA, China [2]School of Integrated Circuits, Southeast University, China
[3]Department of Computer Science, City University of Hong Kong, Hong Kong
[4]Department of Computer Science, University of York, UK

Abstract—Verifying hardware designs in embedded systems is crucial but often labor-intensive and time-consuming. While existing solutions have improved automation, they frequently rely on unrealistic assumptions. To address these challenges, we introduce a novel framework, UVLLM which combines Large Language Models (LLMs) with the Universal Verification Methodology (UVM) to relax these assumptions. UVLLM significantly enhances the automation of testing and repairing error-prone Register Transfer Level (RTL) codes, a critical aspect of verification development. Unlike existing methods, UVLLM ensures that all errors are triggered during verification, achieving a syntax error fix rate of 86.99% and a functional error fix rate of 71.92% on our proposed benchmark. These results demonstrate a substantial improvement in verification efficiency. Additionally, our study highlights the current limitations of LLM applications, particularly their reliance on extensive training data. We emphasize the transformative potential of LLMs in hardware design verification and suggest promising directions for future research in AI-driven hardware design methodologies. The Repo. of dataset and code: https://anonymous.4open.science/r/UVLLM/.

I. INTRODUCTION

Hardware design verification in embedded systems remains heavily dependent on human expertise, making it a tedious and error-prone process that often incurs significant costs [1], as Fig. 1(a) illustrates. A critical aspect of this process is debugging and repairing errors, an area where automated program repair (APR) can contribute. Originally developed for software [2]–[6], APR uses automated tools to fix errors with minimum human intervention and is now being adapted for Hardware Description Language (HDL) design verification due to its potential to reduce human errors and verification costs.

APR systems, as depicted in Fig. 1(b), receive design codes and test cases, and attempt to enact targeted modifications with predefined templates to ensure all tests are passed. Innovations such as Cirfix [7], Strider [8], and RTLrepair [9] demonstrate the potential of APR to reduce the labor and time required for hardware design verification. However, these APR methodologies predominantly rely on fixed templates and focus on addressing functional error, limiting their scope and effectiveness of the repairs.

Fortunately, recent advancements in LLMs such as generating hardware code from natural language specifications [10]–[13], and debugging hardware designs for both functional and syntax errors [14]–[18], have demonstrated promising results in bridging this gap. However, existing solutions are still insufficient for hardware design verification due to their reliance on unreliable assumptions and lack of consideration for the limitations inherent in LLMs. For instance, despite demonstrations of high fix rates for both syntax and functional errors [17], our analysis indicates that approximately 10% of the benchmarks manage to bypass the testbench without undergoing any repairs, and the reliability of some repairs remains questionable owing to insufficient coverage of test cases. These findings underscore the need for more robust solutions capable of effective deployment in real-world verification scenarios.

To address these shortcomings, we propose a comprehensive end-to-end hardware design verification framework, Universal Verification via Large Language Model (UVLLM). This framework integrates the established UVM with LLMs. Our approach enables the first automated hardware design verification framework that operates with

(a) Workflow of the Frontend Hardware Design.

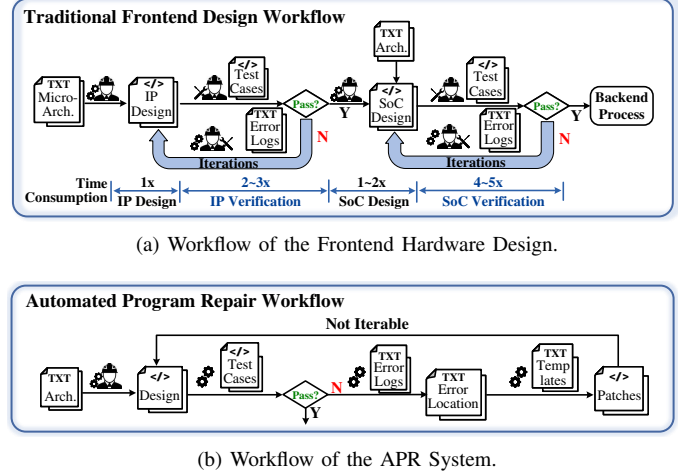

(b) Workflow of the APR System.

Fig. 1: The frontend process is divided into initial design and verification phases, with verification accounting for more than 70% of the duration [1]. Advanced APR techniques are integrated to automate and accelerate the repair stage during verification phase (</>: RTL codes).

practically assumptions, while also managing uncertainties associated with LLM behaviors. This systematic verification strategy, which encompasses testing and repairing, ensures a more robust solution than those currently highlighted in LLM-aided debugging research.

The main **contributions** of this paper are:

- **A comprehensive testing:** Utilizing the UVM, our approach enables flexible test modes and efficient coverage collection. Additionally, the reference models generated by LLMs provide a robust foundation for testing across diverse input scenarios.

- **An open-sourced tooling:** We have created an open-source toolset to enable the broad and early adoption of UVLLM, thereby easing its integration. These tools are publicly available at **https://anonymous.4open.science/r/UVLLM/**.

- **Extensive empirical validation:** We present an open-source error dataset derived from verified projects, containing 331 code instances with realistic errors across various modules, generated by our paradigm error generator. We will continue to organize and update this dataset periodically.

- **Demonstrated performance improvement:** UVLLM, incorporating GPT-4-turbo, significantly enhances verification automation, achieving a syntax error fix rate of 86.99% and a functional error fix rate of 71.92%, exceeding MEIC [17] in terms of repair rates and execution time under realistic testing scenarios. It delivers up to 48x speedup in debugging processes when compared with experienced engineers.

Organisation. The structure of this paper is as follows: Section II outlines the fundamental concepts of UVLLM, while Section III delves into the details of UVLLM and their underlying reasons. Section IV assesses our framework and compares it with existing methods. Section V provides conclusions and make discussions.
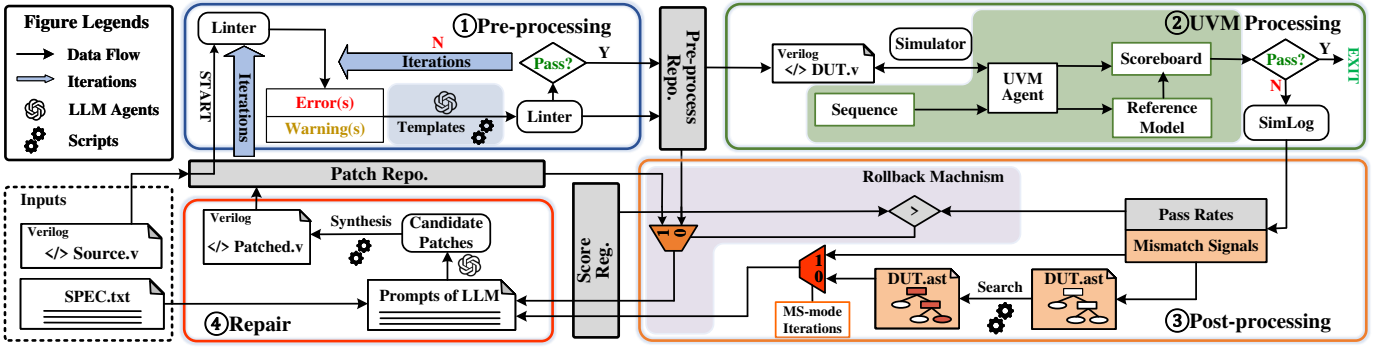
Fig. 2: The UVLLM Framework Overview: The process begins with the DUT and the Specification (Spec.), where the Spec. is used to generate a reference model. Initially, the DUT is pre-processed to eliminate syntax and focused timing-related errors (Step ①). Subsequently, the pre-processed code is tested under a UVM testbench (Step ②), and the log is then post-processed to extract relevant data (Step ③), which the debug agents use to generate candidate patches (Step ④). These codes and their pass rates are archived in the Repository (Repo.) and Register (Reg.) for future iterations.

II. UVLLM: AN OVERVIEW

Designed to enhance the hardware design verification phase, UVLLM aids hardware developers in detecting and correcting common errors in RTL code. The UVLLM is applicable to various hardware environments, in this work, we illustrate the usage of UVLLM in Verilog. As depicted in Fig. 2, this framework combines traditional UVM with LLMs and assumes following inputs:

- **Design specifications** that outline the intended and expected behavior of the hardware component;
- **RTL codes** that contain the untested RTL code of the initial hardware design, i.e., Design Under Test (DUT).

Reflecting advancements in state-of-the-art (SOTA) research, it is noted that using LLMs for hardware design verification requires an iterative approach [17] and is more effective when LLMs are provided with detailed error information [19]–[21]. However, using LLMs still presents certain shortcomings, as current applications of LLMs struggle with reliability, applicability, and processing long code sequences. Additionally, the use of SOTA LLMs can be costly(e.g., the GPT-4-Turbo model, charges $0.01 per $1K$ input tokens and $0.03 per $1K$ output tokens [22]). To address these challenges, UVLLM introduces a cost-efficient, structured four-step process for verifying RTL code against design specifications, as illustrated in Fig. 2.

1) Pre-processing: Starting with raw RTL code as the input, this stage utilizes linters such as Verilator to pre-process the code, removing syntax errors and addressing timing-related functional errors using a combined LLM-script method. The output is syntax-correct DUT, setting a solid foundation for further functionality testing.

2) UVM Processing: Pre-processed DUT code is then tested against a pre-built UVM testbench to identify behavioral discrepancies. Outputs include detailed logs that either confirm alignment with the reference model or highlight deviations with specific signal values and test pass rates, facilitating targeted repairs.

3) Post-processing: Utilizing the UVM logs as input, this stage analyzes the logs to extract critical error data using a localization engine and the Abstract Syntax Tree (AST). The output isolates mismatch signals and specific error paths, preparing them for precise correction in the repair stage.

4) Repair: The final stage takes the design description, the DUT code and the detailed error information from the post-processing as input. Utilizing the information, the debug agents offer candidate patches to correct the errors. The repaired DUT code is then synthesized as the stage output for further iteration.

The termination conditions for the framework loop are: **1)** no errors are detected (**Success**), or **2)** the maximum number of iterations is reached (**Failure**). If any of the above conditions are met, the iteration stops. All history files are stored for reference.

Algorithm 1: Pre-processing DUT with Joint LLM-Script.

Input: DUT file F_D
Output: Pre-processed DUT file F_{Dprep}
1 **Function** PreproDUT(F_D) :
2   **repeat**
3   Log = Linter(F_{Dprep});
4      $Errs$ = Match($Log, Error$);
5      $Warns$ = Match($Log, Warning$);
6   **if** $Errs$ **then**
7   F_{Dprep} = GPT($F_{Dprep}, Errs$);
8   **else if** $Warns$ **then**
9        $WarnTemps$ = Search($Warns, WarnList$);
10   F_{Dprep} = Replace($F_{Dprep}, WarnTemps$);
11   **end**
12   **until** ($Errs == \varnothing$)&($Warns == \varnothing$);
13   **return** F_{Dprep}
14 **End Function**

Modularization. UVLLM uses modular design which allows it to adapt to a wide range of verification scenarios by enabling the use of different tools based on the needs. For example, one can replace an LLM with a more advanced model or use a different linter if required. This flexibility is made possible by standard interfaces between the pipelines, which simplify the integration of different tools via adjustments to the API or by keeping a consistent format.

III. UVLLM: THE FRAMEWORK PIPELINE

As for the operation of the framework, we introduce the joint LLM-Script pre-processing stage (Section III-A), processing stage with tests (Section III-B), post-processing stage for error location (Section III-C), followed by the discussion on the microsystems integrated with the LLM agents (Section III-D). At last, we present our effort for human-like error generation (Section III-E) for evaluation.

A. Pre-processing via the Linter

To ensure compliance with best practices and avoid obvious functional defects, the code is pre-processed using Verilator for linting before UVM testbench evaluation. Through static code analysis and slicing, the method pinpoints potential error sources without executing the code. LLMs have proven effective in Verilog debugging, especially in repairing syntax errors and refining code with offered error sources. By resolving syntax errors and identifying semantic issues early on, the subsequent need for employing LLMs for debugging is minimized, reducing the costs for the use of LLMs.

Combined LLM-Script Pre-pocessing.

To further reduce costs, as detailed in Algorithm 1, we employ a cost-effective strategy that combines LLMs with supplementary scripting to minimize unnecessary LLM usage. In the pre-processing
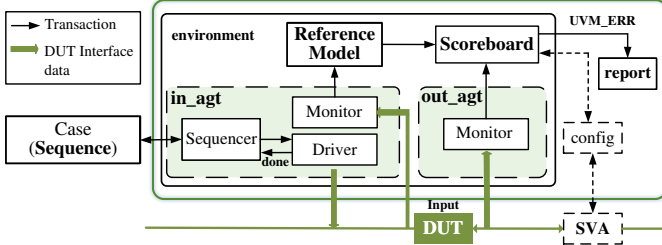
Fig. 3: Structure of the UVM [23], illustrating the key components and their interactions within a typical verification environment.

stage, LLMs are only utilized to help identify and correct syntax errors. These models draw on their extensive training across diverse codebases, efficiently recognizing and amending a broad spectrum of common coding errors and integrating essential error information.

Additionally, the pipeline incorporates scripts designed to address specific warnings that, while not syntax errors, could lead to potential runtime issues, especially some timing-related ones. For instance, in combinational logic circuits, blocking assignments are typical, and Verilator issues warnings for using non-blocking assignments in this case. Through predefined templates, it is possible to systematically identify and modify issues where a non-blocking assignment "$<=$" should be replaced with a blocking assignment "$=$", ensuring the code adheres to expected timing behaviors.

The pre-processing stage iterates until all syntax errors and focused timing-related warnings are resolved. Integrating LLM agents with scripts establishes a robust foundation for subsequent verification stages, ensuring that the DUT encounters only functional errors.

B. UVM Processing

Recent studies, such as MEIC [17], validate LLMs' effectiveness in hardware debugging. However, these studies neglect the importance of testbench construction and employ finite test cases, which restricts the test coverage and leads to overfitting on specific cases. This limited scope significantly undermines the general applicability of the results, as evidenced by a **10%** reduction in the actual fix rate reported by MEIC due to many error instances escaping detection. To overcome these limitations, the UVM framework, depicted in Fig. 3, is employed as the testbench for UVLLM to verify RTL codes, due to its robust support for flexible and complete testing modes.

UVM Construct. To verify complex hardware system, the UVM offers a formal verification structure significantly advanced over simpler testbenches, incorporating components like Agents, Environments, Sequencers, Drivers, and Monitors. Each agent encapsulates a sequencer, driver, and monitor, enabling direct interaction with the DUT. The Sequencer organizes transactions generated from Sequence that simulate real-world operations, which the Driver then translates into pin-level actions on the DUT. Central to UVM's effectiveness is the Scoreboard, which compares actual results with expected outcomes to ensure the DUT performs correctly under various conditions. Additionally, UVM supports various test modes and coverage collection techniques that further enhance testing thoroughness. This method helps identify discrepancies and potential failures, enhancing the reliability and accuracy of the verification process.

Reference Model Generation. In UVM, reference models play a crucial role in verifying complex designs, such as those used in digital signal processing and cryptography, by providing high-level abstractions of the DUT. These models enhance simulation accuracy and efficiency, contributing to a more streamlined verification process. Traditionally, C/C++ is preferred for reference models in industry due to its seamless integration with SystemVerilog via Direct Programming Interfaces (DPI), which accelerates verification cycles [24]–[26]. In this context, the capabilities of LLMs are especially relevant. Given the abundance of open-source datasets, LLMs have shown

---

Algorithm 2: Post-processing with Localization Engine.

Input: DUT file F_D, UVM log L_{UVM}, reference waveform W_R, simulation waveform W_S, iterations $Iter$
Output: Error information $ErrInfo$

1 **Function** ErrChk(L_{UVM}, W_S) **:**
2   /* MT : $Mismatch\ Timestamp$ */
3   /* MS : $Mismatch\ Signals$ */
4   /* IV : $Input\ Values$ */
5   MT, MS = getMismatch(L_{UVM}, PAT_{MS});
6   **if** MS **then**
7   IV = getInputValue(W_S, MT);
8   **end**
9   **return** MT, MS, IV
10 **End Function**
11 **Function** ErrInfoFetch(F_D, L_{UVM}, W_R, W_S, $Iter$) **:**
12   /* SL : $Suspicious\ Code\ Lines$ */
13   MT, MS, IV = ErrChk(L_{UVM}, W_S);
14   **for** $ms \in MS$ **do**
15   DFG = getDFG(F_D, ms);
16      $SL = SL \bigcup$ traverse(DFG, IV);
17   **if** $detectSignal(FL, s)$ and $s \notin MS$ **then**
18        $MS = MS \bigcup \{s\}$;
19   **end**
20   **end**
21    $ErrInfo = (Iter < TH)\ ?\ MS : SL$;
22   **return** $ErrInfo$
23 **End Function**

---

remarkable proficiency in generating C/C++ code, making them well-suited to assist in crafting adaptable, high-quality reference models. These LLM-generated models can dynamically respond to the intricate demands of verification, continuously updating to support high-fidelity simulations and robust design validation.

Extensibility. The extensibility of the UVM is particularly evident when considering the integration of automated assertion generation. UVM's structured, modular framework for verification is optimally configured to incorporate advanced enhancements such as AI-driven assertions, which can systematically verify that the design behaves as expected across various protocols like APB (Advanced Peripheral Bus) and AHB (Advanced High-Performance Bus) [27].

C. Post-processing via Localization Engine

Current LLM-aided verification methods tend to use minimally processed logs as inputs, which are often low in information density, thereby diminishing the efficiency of LLMs in diagnosing and fixing errors. To address this, we adopted time-aware dynamic error localization [8] to extract more concrete and high-value information from these logs with methods. This method, tailored for HDL environments, surpasses the static localization method described in Section III-A by providing greater precision and temporal sensitivity.

The localization engine leverages dynamic analysis and temporal insights to detect discrepancies between expected and actual signal outputs as recorded in the UVM log. These discrepancies are crucial for performing dynamic slicing through data flow graphs (DFGs), as outlined in Algorithm 2. To optimize token usage, UVLLM adopts a segmented information extraction strategy. Initially, mismatch signals are input into the LLM's prompt as indicators of potential errors. If subsequent repair attempts fail, this indicates that relying solely on mismatched signals may be insufficient. To increase diagnostic precision, the system then incorporates actual execution paths—identified as suspicious—into the analysis alongside the error signals. This approach focuses on actual execution paths in operation, leading to more precise identification of errors.

Rollback Mechanism. During the development of UVLLM, a *Rollback mechanism* was implemented to address the issue of inaccuracies in LLM outputs, often termed "hallucination" [28]–[32]. This feature
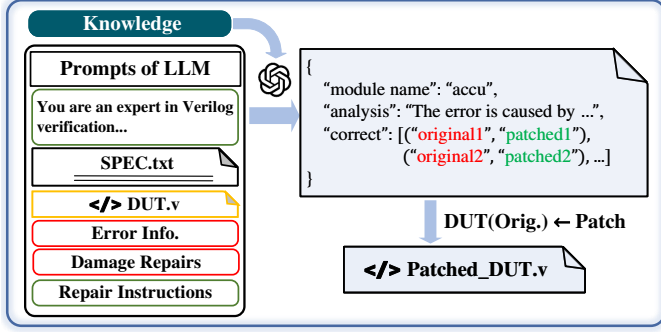
Fig. 4: Input and Output Formats for LLM Agents: The input format is structured for various agents with minor prompt modifications. The output is provided in JSON format, featuring original-patch pairs.

TABLE I: Part of common Verilog errors in real-world designs.

Types	Error	Symptoms
Declare	Type Misuse	output **reg** [8:0] result; output [8:0] result;
Declare	Bitwidth Misuse	reg **[8:0]** count; reg **[7:0]** count;
Assignment	Operator Misuse	always @(*) result = a + b; always @(*) result = a **-** b;
Assignment	Variable Name Misuse	assign r1 = **r1_temp**; assign r1 = **r2_temp**;
Assignment	Value Misuse	if (rstn) data = **32'b0**; if (rstn) data = **32'b1**;
Condition	Wrong Judgment Value	for(i = 0; **i < 7**; i ++) begin ... end for(i = 0; **i < 15**; i ++) begin ... end
Condition	Wrong Sensitivity	always@(posedge clk **or negedge rstn**) ... always@(posedge clk) ...
Port	Port Mismatch	mod mod1(.a(a), .b(b), .in_bd(**{bdg, 1'b1}**)); mod mod1(.a(a), .b(b), .in_bd(**1'b1**));

is crucial for preventing the accumulation of errors across iterations due to reliance on flawed candidate repairs. Despite previous studies utilizing LLMs as reward models to assess repair quality [17], [33], there is a lack of robust quantitative metrics to effectively measure the correctness of these modifications. This gap can lead to inefficiencies in the rollback process, potentially triggering false positive rollbacks.

Within the UVM framework, the quality of Verilog code iteration is evaluated using a scoreboard that assigns the test pass rate. We assume that higher scores correlate with fewer errors and better functionality. The Rollback mechanism functions by preserving a history of code versions and their scores. If a new iteration scores lower than a previous one, indicating a decline in code quality, the mechanism reverts to the highest-scoring version, as illustrated in Fig. 2. The alterations that led to the decrement in score are thus recorded as "damage repairs", which is utilized in Fig. 4.

D. Repair Agent

The LLM agent functions as an adept RTL repair expert, leveraging three key inputs: the design specification, which outlines intended functionality and port definitions; RTL code; and error information. To enhance the repair process and facilitate iterative improvements, the system incorporates "damage repairs" as an additional input, crucial for preventing the recurrence of unsuccessful corrections when the rollback mechanism is activated. In a multi-agent setup, specific modifications to the prompts for each agent address different aspects of the debugging process, enabling a more nuanced and thorough error analysis in the RTL code. The primary prompts that guide the debugging activities of these agents are illustrated in Fig. 4, emphasizing a tailored approach to error resolution.

Formalizing agent's outputs. It's commonly noticed that LLM-generated responses tend to include detailed explanations during debugging, which can clutter the main objective of code debugging. In light of our observations, it becomes evident that LLMs exhibit enhanced debugging capabilities with superior reasoning process. To prevent the inclusion of irrelevant details and erase the hallucination during the iterative cycle, a method for distilling the responses generated by the LLM is adopted. Utilizing the Structured Outputs method enables adherence to JSON Schema [34], thereby ensuring that responses are consistently formatted according to predefined structures. By requiring the response to be in JSON format and to contain an element labeled "correct" which consists of pair of wrong codes and right codes, the code sections accentuated in Fig. 4 are refined and carried into the subsequent iteration.

E. Benchmark Generation

The incidence of errors in module code is closely related to specific attributes such as code length and functional complexity [35]–[37]. To evaluate the efficacy of the verification methodology, a well

constructed evaluation dataset was developed through a systematic selection of samples from validated open-source datasets, representing a diverse range of codebases. These samples were then deliberately infused with typical errors to simulate real-world coding mistakes.

In design bases combining both commercial and open-source IPs, a comparative analysis was performed on two consecutive versions of the code: one immediately before and another after code repository commits. This analysis focused on identifying discrepancies and documenting changes made during the design process, effectively highlighting the differences between pre and post-commit versions. These error-modification pairs, detailed in Table I, were crucial for developing prompts for LLM and for terms used in pattern matching, showcasing common human-made errors such as misuse of assignments and the mismatch port in instantiation. This approach showcases common human-made errors, such as misuse of assignments and mismatches in port instantiation, and provides a crucial benchmark for evaluating the verification effectiveness.

IV. EVALUATION

This section presents our experimental setup, evaluation metrics research questions, and discussions to the results.

Setup. In our experiment, we employed LLM agents via the OpenAI API, with GPT-4-turbo as the default model. An evaluation benchmark was then constructed using the extensively verified RTLLM dataset [38], which encompasses a diverse array of real-world errors. The initial code's ability to pass the compiler is indicated by a syntax error or functional error. We utilized a range of simulation tools including VCS [39], Iverilog [40], Modelsim [41], Yosys [42], and the linting tool Verilator [43] to ensure comprehensive verification and analysis. We set the threshold of iterations to 5, as the improvement is hardly observed after that. All experiments were conducted on an AMD EPYC 7763 2.45GHz CPU. For each instance, we asked LLMs for 5 times to reduce the randomness of the response.

A. Evaluation Metrics

Recent work [44], [45], tended to use pass@k metrics to assess functional correctness. For each problem in the problem set, k code samples are generated at a time, and the problem is considered solved if any sample passes the simulation test.

Hit Rate (HR). Specifically, our framework quantifies effectiveness using Hit Rate (HR) [46]. For erroneous code θ_i and its corrected version θ_i^*, we evaluate a set of test cases $\{(x_i^1, y_i^1), \ldots, (x_i^m, y_i^m)\}$. The corrected code θ_i^* must produce the correct output y_i^j for each input x_i^j, ensuring all cases pass. That is, $\bigwedge_{j=1}^{m} a_{\theta_i^*}(x_i^j) = y_i^j$. The overall rate for n corrected versions is calculated as:

$$\textbf{HR} = \sum_{i=1}^{n} \frac{\bigwedge_{j=1}^{m} \left[a_{\theta_i^*}(x_i^j) = y_i^j \right]}{n} \times 100\% \qquad (1)$$
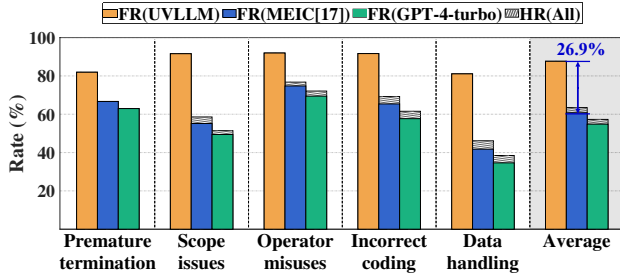
Fig. 5: HR vs. FR in Syntax-Error Verification with Different Methods [17]. The differences between HR and FR are shaded.
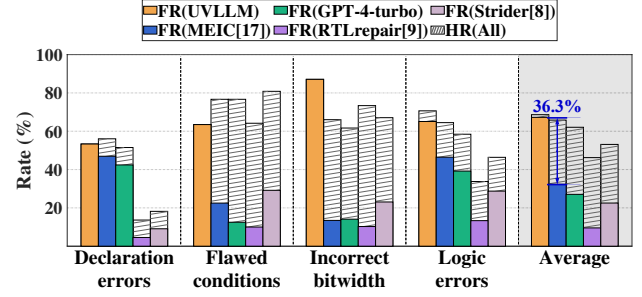

Fig. 6: HR vs. FR in Functional-Error Verification with Different Methods [8], [9], [17]. The differences between HR and FR are shaded.
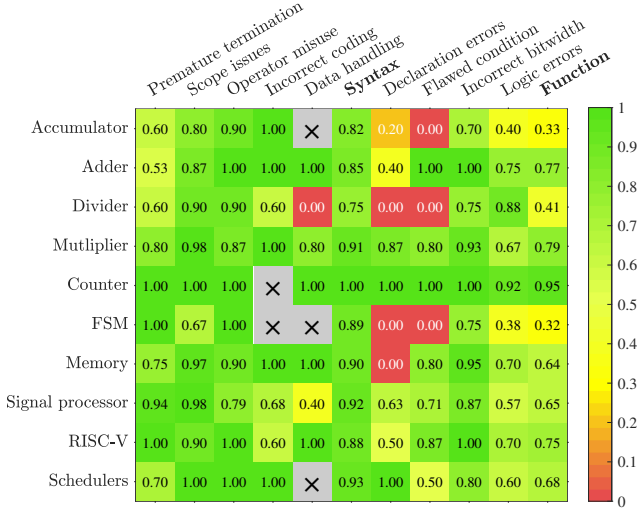

Fig. 7: Heat map result for FR. The symbol "×" represents an error that could not be imposed due to the limitations of the specific module structure. Syntax and Function represent the weighted mean of the FR for syntax errors and function errors, respectively.

HR measures the proportion of instances resolved under test cases.
Fix Rate (FR). To overcome limitations in test coverage, our framework includes Fix Rate (FR), which involves independent expert validation of the proposed fixes θ_i^*. After expert review, if the fix is confirmed effective across additional scenarios, it is designated as $\hat{\theta}_i^*$:

$$\mathbf{FR} = \sum_{i=1}^{n} \frac{\hat{\theta}_i^*}{n} \times 100\% \tag{2}$$

FR reflects the framework's repair effectiveness in broader conditions.
Execution Time. In evaluating the framework's performance, this paper emphasizes execution time as a critical metric, defined as the time interval from the initial design input into UVLLM to the completion of the final code output.

B. Results and Discussions

Result 1:The UVLLM framework's effectiveness in enhancing hardware design verification is demonstrated through a comparative evaluation of Fix Rates (FRs) among various methodologies, as depicted in Figures 5 and 6. This analysis encompasses traditional script-based approaches [8], [9], current LLM-aided methods [17], and a baseline incorporating GPT-4-turbo for benchmark repairs. Across all categories of errors—both syntax and functional—the UVLLM framework consistently achieves higher FRs.

Specifically, for syntax errors, the UVLLM framework records an average FR of 87.6%, representing a significant 26.9% improvement over the second performing method, MEIC. For functional errors, UVLLM continues to surpass other methods, registering a FR of

67.3%, which is a 36.3% enhancement relative to MEIC. Notably, in the case of *Incorrect bitwidth* as illustrated in Fig. 6, UVLLM's FR is more than double that of the next best method, RTLrepair. Moreover, even in less favorable scenarios, such as *Declaration errors*, UVLLM's FR remains approximately 5% above that of MEIC, its nearest competitor. These results clearly demonstrate that UVLLM consistently outperforms other methods, achieving significantly higher FRs across diverse scenarios.

Result 2: The evaluation of the UVLLM framework's capability to ensure that repaired code adheres to specification requirements is conducted through an examination of the correctness of repaired code. Although many methods achieve high hit rates (HRs), their fixes often overfit to specific input-output (IO) pairs, revealing a discrepancy between HRs and FRs.

Figures 5 and 6 illustrate this disparity; the shaded areas represent the deviation between HR and FR, highlighting the failure of these methods to detect numerous errors, which leads to false negatives and insufficient coverage. Specifically, for syntax errors, UVLLM demonstrated no deviation across all scenarios, while deviations for other methods were observed in 4 out of 5 scenarios with an average of 5% variations, confirming the achievement of high coverage for syntax errors. In contrast, for functional errors, the deviations for other methods were notably higher (all above 30%) whereas UVLLM maintained a minimal deviation of only 1.4%. Notably, UVLLM had a maximum deviation of just 5.6% for *Logic errors*, while the other methods displayed deviations exceeding 40% for *Flawed conditions*.

These results suggest that while other methods struggle to achieve comprehensive coverage for functional errors, UVLLM effectively mitigates this limitation. These findings indicate that UVLLM significantly enhances the practicality of hardware design verification by integrating formal verification processes, to meet the specification requirements to the greatest extent possible.

Result 3: The UVLLM framework's repair capabilities across a diverse range of hardware modules were evaluated by analyzing the FRs of 27 common modules, each injected with nine distinct types of syntax and functional errors. These modules were categorized into ten representative types, such as *adders*, *counters*, and *FSMs*, to establish a comprehensive benchmark for evaluating the framework's generalization in various verification scenarios.

As illustrated in Fig. 7, where the framework's FRs were depicted using color coding, UVLLM exhibited exceptional adaptability, achieving robust FRs in simpler modules like *counters*. For instance, the FRs for syntax errors and functional errors in these modules reached 100% and 95%, respectively. In contrast, the FRs were lower in more complex modules, such as *FSMs*, with FRs for syntax errors and functional errors at 89% and 32%, respectively. This indicates that repairing more complex designs remains challenging.

Across the same types of module, syntax errors consistently exhibited higher FRs than functional errors, reflecting UVLLM's proficiency in addressing syntactic issues. This advantage stems from

TABLE II: Performance comparison of segmented approach across common modules with various error instances.

Types	Pre-processing[1]		Repair in MS Mode		Repair in SL Mode		UVLLM[2]		MEIC [17]		Speedup
	$FR/\%$	T_{exec}/s	$FR/\%$	T_{exec}/s	$FR/\%$	T_{exec}/s	$FR/\%$	T_{exec}/s	$FR/\%$	T_{exec}/s	
Arithmetic[3] s[4]	69.93	8.30	13.07	5.60	1.31	0.30	84.31	14.20	62.30	197.29	**13.89x**
Control s	80.91	7.23	8.18	3.38	0.00	0.00	89.09	10.61	63.64	129.02	**12.61x**
Memory s	60.00	10.29	28.33	5.14	0.00	0.00	88.33	15.43	54.55	147.17	9.53x
Miscellaneous s	79.65	8.31	7.67	4.77	1.18	0.39	88.50	13.47	66.67	62.67	4.65x
Syntax	**74.72**	8.49	11.29	5.06	0.98	0.27	**86.99**	**13.83**	**62.99**	**134.95**	9.76x
Arithmetic f	30.26	3.63	33.33	11.27	2.63	0.64	66.23	15.54	40.53	257.28	**16.56x**
Control f	29.93	3.33	30.61	9.37	5.44	0.84	65.99	13.54	10.91	163.96	**12.55x**
Memory f	25.00	4.35	58.33	11.22	3.33	0.87	86.67	16.44	22.73	256.49	**15.60x**
Miscellaneous f	21.25	3.86	49.06	11.54	5.63	0.90	75.94	16.30	40.07	65.37	4.01x
Function	25.96	3.82	**41.46**	11.19	4.50	0.78	**71.92**	**15.79**	**34.57**	**191.76**	12.14x
Overall	**51.27**	**6.16**	**25.80**	**7.79**	**2.68**	**0.49**	**79.75**	**14.77**	**52.14**	**153.84**	10.42x

[1] The repair operation of UVLLM comprises three stages: Pre-processing, Repair in Mismatch Signal (MS) Mode, and Repair in Suspicious Line (SL) Mode. The columns labeled FR and T_{exec} indicate the contributions of each stage to the fix rate and execution time, respectively.
[2] The column labeled UVLLM summarizes the total contributions across all stages of the repair operation.
[3] Modules are grouped as Arithmetic (Accumulator, Adder, Divider, Multiplier), Control (Counter, FSM), Memory, and Miscellaneous (other modules).
[4] Errors are categorized as syntax ("s") and function ("f").

the extensive training of the LLM on a substantial corpus of HDL code data, enhancing its syntactic understanding. Additionally, the compiler and linter contribute detailed localization information that aids in the repair of syntax errors.

Overall, the framework achieved an FR of 86.99% for syntax errors and 71.92% for functional errors, representing its reliability across diverse modules and error scenarios.

Result 4: The UVLLM framework's evaluation primarily focuses on how its distinct stages contribute to the fix rate and execution time during the repair operation, providing insights into each segment of the verification process.

Table II details the repair process, which unfolds in several stages, each playing a different role in resolving syntax and functional errors. The Pre-processing stage was particularly effective in addressing syntax errors, successfully resolving 74.72% of these cases as highlighted. For functional errors, the Mismatch Signal (MS) mode in the Repair stage was most effective, correcting 41.46% of the instances. The adoption of segmented steps enables UVLLM to work effectively and adapt flexibly to various verification scenarios.

For errors strictly related to syntax, the majority were successfully corrected during the pre-processing stage. However, 11.29% of syntax-only errors persisted and advanced to the subsequent repair stage in MS mode. Similarly, the attempt to resolve 25.96% of functional errors inadvertently introduced new syntax issues, which were then addressed by the pre-processor. This demonstrates UVLLM's ability to compensate for new errors introduced in earlier stages and mitigate the uncertainties associated with LLMs.

In terms of execution time, the segmented repair operation shows that the pre-processing stage, despite handling over 50% of all benchmark repairs, typically requires less time than repairs conducted in MS mode. This demonstrates the efficiency benefits of incorporating a robust pre-processing stage.

Result 5: The UVLLM framework's execution efficiency was evaluated against existing methods from a multidimensional perspective, mainly focusing on two key metrics: Failure Rates (FRs) and execution time T_{exec}, as detailed in Table II.

While Fig.5 and Fig.6 show the variations in FR performance across different error types, UVLLM consistently surpassed MEIC across all module types. For instance, within the *Miscellaneous* modules for syntax errors in Table II, UVLLM achieved an FR of 88.50%, which is 21.83% higher than MEIC's 66.67%. In handling functional errors within *Arithmetic* modules, characterized by their complex logic, UVLLM maintained an FR of 66.23%, significantly outperforming MEIC's 40.53%. These results underscore UVLLM's robust capability to effectively resolve a wide spectrum of errors across different modules compared to existing methods, thus proving

TABLE III: Ablation study. UVLLM$_{pair}$ and UVLLM$_{comp}$ represent UVLLM with LLMs generating code pairs and complete codes.

Framework	$FR/\%$		T_{exec}/s	
	Syntax	Func.	Syntax	Func.
$UVLLM_{pair}$	86.99	71.92	13.83	15.79
$UVLLM_{comp}$	70.41	59.25	35.60	71.84

its effectiveness and reliability in boosting system performance.

In terms of operational efficiency, UVLLM also demonstrated a substantial reduction in execution time. For example, when processing syntax errors within the *Miscellaneous* modules, UVLLM recorded an average execution time of 13.47s, marking a speedup of 4.65x compared with MEIC. This advantage was even more pronounced when addressing complex functional errors, where UVLLM achieved a speedup of up to 16.56x over MEIC in *Arithmetic* modules. On average, UVLLM operated 10.42x faster than MEIC, while simultaneously achieving higher test coverage and pass rates. These findings highlight UVLLM's potential to significantly enhance the design verification process for practical deployment by merging increased automation with superior efficiency.

C. Ablation Study

Our research includes the ablation study designed to evaluate the impact of iteration strategies on the effectiveness of the framework. **Repair generation form.** We initially employ an approach that uses original-repair code pairs to facilitate the generation of new code by leveraging outputs from LLMs. However, the ablation study examines an alternative method where entire code snippets are directly produced by the LLMs, omitting the generation of repair pairs.

As shown in Table III, generating complete code snippets resulted in a slight decline in repair accuracy and an increase in execution time compared to generating original-repair code pairs. Nevertheless, there were specific scenarios where this direct generation method proved superior. This advantage is mainly due to the ability of the direct generation method to handle minor errors that pose significant challenges for LLMs in terms of search efficiency. In some cases, regenerating the entire code is more effective than trying to modify or replace segments of the existing code. For instance, correcting the error "$module\ a(A); ...(Missing\ Definition\ of\ Port\ A)...endmodule$" proves challenging for the replacement strategy, primarily because the essential context is frequently overlooked, whereas the reproduction method handles it more straightforwardly.

V. CONCLUSION

In this work, an automated universal verification framework, **UVLLM**, which comprehensively addresses main phases of hardware

design verification, including testbench construction, test execution, result analysis, and repair, is proposed. Tested on the proposed benchmark, UVLLM achieves average syntax and functional error fault rates of 86.99% and 71.92%, respectively, while maintaining nearly 100% test coverage. Additionally, UVLLM performs 10.42 times faster than the previous MEIC framework. The framework demonstrates that it is feasible to employ the LLMs for the purpose of Verilog code verification, irrespective of the initial code state. The utilization of reasonable segmentation and feedback engineering leads to an improvement in the verification efficiency of the framework.

REFERENCES

[1] S. Lahti, P. Sjövall, J. Vanne, and T. D. Hämäläinen, "Are we there yet? a study on the state of high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 5, pp. 898–911, 2018.

[2] Y. Liu, L. Zhang, and Z. Zhang, "A survey of test based automatic program repair." *J. Softw.*, vol. 13, no. 8, pp. 437–452, 2018.

[3] K. Liu, L. Li, A. Koyuncu, D. Kim, Z. Liu, J. Klein, and T. F. Bissyandé, "A critical review on the evaluation of automated program repair systems," *Journal of Systems and Software*, vol. 171, p. 110817, 2021.

[4] Q. Zhang, T. Zhang, J. Zhai, C. Fang, B. Yu, W. Sun, and Z. Chen, "A critical review of large language model on software engineering: An example from chatgpt and automated program repair," *arXiv preprint arXiv:2310.08879*, 2023.

[5] X. Yin, C. Ni, S. Wang, Z. Li, L. Zeng, and X. Yang, "Thinkrepair: Self-directed automated program repair," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1274–1286.

[6] K. Xu, G. L. Zhang, X. Yin, C. Zhuo, U. Schlichtmann, and B. Li, "Automated c/c++ program repair for high-level synthesis via large language models," in *Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD*, 2024, pp. 1–9.

[7] H. Ahmad, Y. Huang, and W. Weimer, "Cirfix: automatically repairing defects in hardware design code," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 990–1003.

[8] D. Yang, J. He, X. Mao, T. Li, Y. Lei, X. Yi, and J. Wu, "Strider: Signal value transition-guided defect repair for hdl programming assignments," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 43, no. 5, pp. 1594–1607, 2024.

[9] K. Laeufer, B. Fajardo, A. Ahuja, V. Iyer, B. Nikolić, and K. Sen, "Rtl-repair: Fast symbolic repair of hardware design code," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2024, pp. 867–881.

[10] M. Liu, N. Pinckney, B. Khailany, and H. Ren, "Verilogeval: Evaluating large language models for verilog code generation," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 2023, pp. 1–8.

[11] S. Thakur, B. Ahmad, H. Pearce, B. Tan, B. Dolan-Gavitt, R. Karri, and S. Garg, "Verigen: A large language model for verilog code generation," *arXiv preprint arXiv:2308.00708*, 2023.

[12] J. Blocklove, S. Garg, R. Karri, and H. Pearce, "Chip-chat: Challenges and opportunities in conversational hardware design," *arXiv preprint arXiv:2305.13243*, 2023.

[13] M. DeLorenzo, A. B. Chowdhury, V. Gohil, S. Thakur, R. Karri, S. Garg, and J. Rajendran, "Make every move count: Llm-based high-quality rtl code generation using mcts," *arXiv preprint arXiv:2402.03289*, 2024.

[14] M. Liu, T.-D. Ene, R. Kirby, C. Cheng, N. Pinckney, R. Liang, J. Alben, H. Anand, S. Banerjee, I. Bayraktaroglu *et al.*, "Chipnemo: Domain-adapted llms for chip design," *arXiv preprint arXiv:2311.00176*, 2023.

[15] Y. Tsai, M. Liu, and H. Ren, "Rtlfixer: Automatically fixing rtl syntax errors with large language models," *arXiv preprint arXiv:2311.16543*, 2023.

[16] X. Yao, H. Li, T. H. Chan, W. Xiao, M. Yuan, Y. Huang, L. Chen, and B. Yu, "Hdldebugger: Streamlining hdl debugging with large language models," *arXiv preprint arXiv:2403.11671*, 2024.

[17] K. Xu, J. Sun, Y. Hu, X. Fang, W. Shan, X. Wang, and Z. Jiang, "Meic: Re-thinking rtl debug automation using llms," *arXiv preprint arXiv:2405.06840*, 2024.

[18] W. Fu, K. Yang, R. G. Dutta, X. Guo, and G. Qu, "Llm4sechw: Leveraging domain-specific large language model for hardware debugging," in *2023 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*. IEEE, 2023, pp. 1–6.

[19] J. White, Q. Fu, S. Hays, M. Sandborn, C. Olea, H. Gilbert, A. El-nashar, J. Spencer-Smith, and D. C. Schmidt, "A prompt pattern catalog to enhance prompt engineering with chatgpt," *arXiv preprint arXiv:2302.11382*, 2023.

[20] P. Sahoo, A. K. Singh, S. Saha, V. Jain, S. Mondal, and A. Chadha, "A systematic survey of prompt engineering in large language models: Techniques and applications," *arXiv preprint arXiv:2402.07927*, 2024.

[21] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

[22] OpenAI, "GPT-4 pricing for OpenAI API," 2024. [Online]. Available: https://openai.com/api/pricing/

[23] F. Plasencia-Balabarca, E. Mitacc-Meza, M. Raffo-Jara, and C. Silva-Cárdenas, "Robust functional verification framework based in uvm applied to an aes encryption module," in *2018 New Generation of CAS (NGCAS)*. IEEE, 2018, pp. 194–197.

[24] C. Spear, *SystemVerilog for Verification: A Guide to Learning the Testbench Language Features*. Springer, 2008, vol. 161.

[25] L. Séméria and A. Ghosh, "Methodology for hardware/software co-verification in c/c++ (short paper)," in *Proceedings of the 2000 Asia and South Pacific Design Automation Conference*, 2000, pp. 405–408.

[26] Y. Nakamura, K. Hosokawa, I. Kuroda, K. Yoshikawa, and T. Yoshimura, "A fast hardware/software co-verification method for system-on-a-chip by using a c/c++ simulator and fpga emulator with shared register communication," in *Proceedings of the 41st annual Design Automation Conference*, 2004, pp. 299–304.

[27] V. Radu, D. Dranga, C. Dumitrescu, A. I. Tabirca, and M. C. Stefan, "Generative ai assertions in uvm-based system verilog functional verification," *Systems*, vol. 12, no. 10, p. 390, 2024.

[28] Z. Ji, T. Yu, Y. Xu, N. Lee, E. Ishii, and P. Fung, "Towards mitigating llm hallucination via self reflection," in *Findings of the Association for Computational Linguistics: EMNLP 2023*, 2023, pp. 1827–1843.

[29] X. Amatriain, "Measuring and mitigating hallucinations in large language models: amultifaceted approach," 2024.

[30] B. A. Galitsky, "Truth-o-meter: Collaborating with llm in fighting its hallucinations," 2023.

[31] Y. Chen, Q. Fu, Y. Yuan, Z. Wen, G. Fan, D. Liu, D. Zhang, Z. Li, and Y. Xiao, "Hallucination detection: Robustly discerning reliable answers in large language models," in *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management*, 2023, pp. 245–255.

[32] Z. Ji, N. Lee, R. Frieske, T. Yu, D. Su, Y. Xu, E. Ishii, Y. J. Bang, A. Madotto, and P. Fung, "Survey of hallucination in natural language generation," *ACM Computing Surveys*, vol. 55, no. 12, pp. 1–38, 2023.

[33] N. Lambert, V. Pyatkin, J. Morrison, L. Miranda, B. Y. Lin, K. Chandu, N. Dziri, S. Kumar, T. Zick, Y. Choi *et al.*, "Rewardbench: Evaluating reward models for language modeling," *arXiv preprint arXiv:2403.13787*, 2024.

[34] OpenAI, "Structured outputs - OpenAI API," 2024. [Online]. Available: https://platform.openai.com/docs/guides/structured-outputs

[35] S. Sudakrishnan, J. Madhavan, E. J. Whitehead Jr, and J. Renau, "Understanding bug fix patterns in verilog," in *Proceedings of the 2008 international working conference on Mining software repositories*, 2008, pp. 39–42.

[36] J. Ma, G. Zuo, K. Loughlin, H. Zhang, A. Quinn, and B. Kasikci, "Debugging in the brave new world of reconfigurable hardware," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 946–962.

[37] V. Antinyan, M. Staron, and A. Sandberg, "Evaluating code complexity triggers, use of complexity measures and the influence of code complexity on maintenance time," *Empirical Software Engineering*, vol. 22, pp. 3057–3087, 2017.

[38] Y. Lu, S. Liu, Q. Zhang, and Z. Xie, "Rtllm: An open-source benchmark for design rtl generation with large language model," *arXiv preprint arXiv:2308.05345*, 2023.

[39] Synopsys, "VCS: Synopsys Verification Continuum," 2024. [Online]. Available: https://www.synopsys.com/verification/simulation/vcs.html

[40] I. V. Team, "Icarus Verilog," 2024. [Online]. Available: http://iverilog.icarus.com/

[41] Siemens, "ModelSim," 2024. [Online]. Available: https://eda.sw.siemens.com/en-US/ic/modelsim/

[42] C. Wolf, "Yosys Open SYnthesis Suite," 2024. [Online]. Available: http://www.clifford.at/yosys/

[43] W. Snyder, "Verilator," 2024. [Online]. Available: https://www.veripool.org/wiki/verilator

[44] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[45] S. Fakhoury, A. Naik, G. Sakkas, S. Chakraborty, and S. K. Lahiri, "Llm-based test-driven interactive code generation: User study and empirical evaluation," 2024.

[46] R. Tian, Y. Ye, Y. Qin, X. Cong, Y. Lin, Z. Liu, and M. Sun, "Debugbench: Evaluating debugging capability of large language models," *arXiv preprint arXiv:2401.04621*, 2024.