LLM-Assisted Circuit Verification: A Comprehensive Survey

Hongduo Liu, Yuntao Lu, Mingjun Wang, Xufeng Yao, Bei Yu
Department of Computer Science and Engineering
The Chinese University of Hong Kong

***Abstract*—Circuit verification constitutes a significant bottleneck in modern electronic design, often consuming up to 70% of the development cycle due to the escalating complexity of circuits. The emergence of large language models (LLMs) presents a promising new frontier, demonstrating profound capabilities in code generation, debugging, and automated reasoning. This paper provides a comprehensive survey of LLM-assisted hardware verification. We systematically review state-of-the-art approaches that leverage LLMs for critical verification tasks, including assertion synthesis, testbench generation, automated debugging, and the development of collaborative verification frameworks. We analyze the effectiveness of various LLM strategies specifically designed for hardware verification applications, while also discussing the integration of LLMs with existing verification workflows and tools. Furthermore, the paper identifies key remaining challenges and outlines a forward-looking perspective on future research directions.**

I. Introduction

Verification in the circuit design flow ensures that implementations meet specifications across all design stages. It includes functional verification by generating test stimulus [1], [2] at the register-transfer and system levels to validate intended behavior, logic verification [3], [4] to ensure correctness of control and data paths, and physical verification [5], [6] to confirm timing closure and layout integrity. Verification is indispensable because post-silicon defects cannot be patched like software, and recalls are prohibitively expensive. For example, Pentium's FDIV bug [7] alone reportedly cost Intel around $475 million [8]. Additionally, chips are widely deployed in safety-critical domains, such as aerospace and medical devices, where rigorous verification is crucial to ensure reliability.

While indispensable, verification consumes a disproportionate share of development time, which arises from three compounding challenges. First, verification permeates the entire design flow, as each stage from front-end to back-end requires verification. Second, core verification tasks, such as equivalence checking and reachability analysis, are computationally intractable, and many of these problems are NP-hard. Third, modern VLSI designs face an exponential growth in state space due to the increasing number of transistors and architectural complexity. These factors make verification a primary bottleneck, as shown in Fig. 1. Verification can consume over 70% of total design time for many IC/ASIC projects in recent years.

Recently, LLMs, such as ChatGPT, [9], DeepSeek, [10], Claude, [11], and Gemini [12] have revolutionized natural language processing. They have demonstrated unprecedented performance across various tasks, including conversation, translation, coding, retrieval, summarization, and creative writing, among others. Empowered by the Transformer architecture [13], LLMs take in sequences of tokens (text split into subword units), build contextual representations with self-attention, and generate coherent outputs one token at a time. Trained on massive corpora and fine-tuned with human feedback, they learn patterns of language, reasoning heuristics, and task instructions, enabling them to follow prompts, adapt to new domains, and assist with complex, multi-step workflows.

The project is supported in part by Research Grants Council of Hong Kong SAR (No. CUHK14210723).

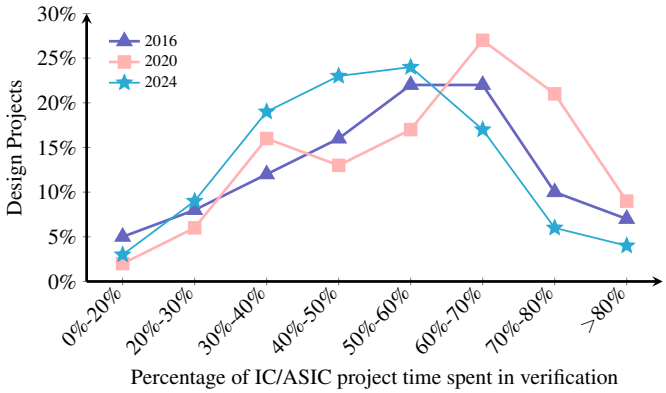

Fig. 1 Verification effort as a percentage of total IC/ASIC project time. The data is from the 2024 Wilson Research Group IC/ASIC functional verification trend report [27].

In hardware design and verification, engineers work with diverse artifacts, including natural language (NL) specifications, hardware description languages, assertions, constraints, test benches, and electronic design automation (EDA) scripts. LLMs are particularly well-suited for this domain because all of these artifacts are fundamentally language-based. Recent research has demonstrated that LLMs can generate and optimize RTL code [14]–[17], write EDA scripts [18]–[21], answer questions on EDA documentation [22], generate assertions [23], produce test stimuli [24], assist with debugging [25], and improve verification tools such as SAT solvers [26]. These capabilities collectively accelerate design iteration, improve traceability, and reduce the manual effort required to achieve design closure.

Recent literature has begun to explore the intersection of AI and hardware, yet existing surveys approach the topic from overly broad perspectives. One group of surveys, including those by Abdollahi et al. [28] and Alsaqer et al. [29], examines LLM applications across the entire hardware design spectrum, thereby situating verification as just one component among many. Similarly, EDA-focused reviews by Pan et al. [30], Fang et al. [31], and Zhong et al. [32] treat verification as a single stage within the larger design automation workflow. A second group, represented by the work of Wu et al. [33], focuses on hardware verification but surveys the entire landscape of machine learning techniques, from traditional methods to deep learning, not specifically LLMs. In contrast, this survey provides a comprehensive and exclusive analysis dedicated to LLM-assisted circuit verification.

To provide a structured overview, this survey follows the logical progression of a modern verification workflow, examining how LLMs can augment each key stage. We begin with assertion generation in Section II, where LLMs assist in formulating precise properties that capture the intended temporal and logical behavior of the design for formal verification. We then examine testbench generation for simulation-based verification in Section III, in which the design under test is exercised with synthesized stimuli, checkers, and coverage

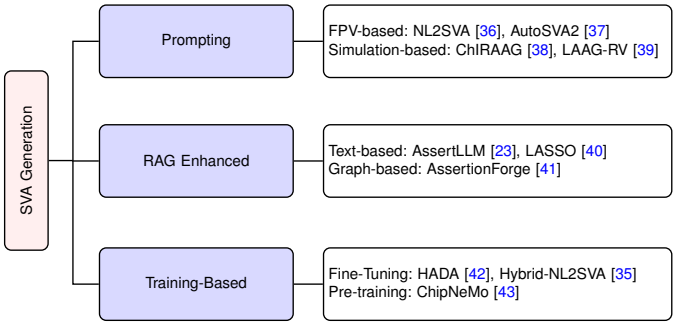

Fig. 2 Taxonomy of LLM-assisted SVA Generation techniques.

models to validate expected responses in a simulated environment. Next, we consider LLM-assisted debugging in Section IV, focusing on identifying failures, localizing root causes, and proposing candidate fixes from logs, traces, and waveforms. Finally, we explore LLM-based holistic verification frameworks in Section V, which integrate multiple verification capabilities into unified, end-to-end workflows.

II. Assertion Generation

System Verilog assertions (SVAs) serve as formal specifications that define the expected temporal and logical behavior of hardware designs, enabling both simulation-based and formal property verification (FPV). While SVAs are crucial for ensuring design correctness, their manual construction is labor-intensive and requires deep expertise in both hardware semantics and temporal logic [34]. The challenge has motivated significant research into leveraging LLMs to automate assertion generation from various sources, including NL specifications, design documentation, and RTL code.

The automation of assertion generation faces several fundamental challenges. First, NL descriptions and specification documents are inherently unstructured, containing background information, functional descriptions, and diagrams scattered across sections, making direct extraction difficult [23]. Second, even with structured specifications, translating NL into precise temporal logic requires both deep domain knowledge and expertise in SVA syntax, particularly for complex operators like overlapping (|->) and non-overlapping (|=>) implication [35]. Third, the lack of comprehensive training data and evaluation benchmarks has hindered systematic assessment of LLM capabilities in this domain. These challenges have led to diverse research directions, ranging from zero-shot prompting strategies to sophisticated retrieval-augmented architectures and domain-specific fine-tuning approaches. We investigate the approaches that LLM-based assertion-generation frameworks use to address challenges, as shown in Fig. 2.

A. Assertion Generation before LLMs

Before the emergence of LLMs, early work explored rule-based and grammatical learning methods for assertion generation. GLAsT [34] employed grammatical inference to automatically learn formal attribute grammars from small training sets of NL and SVA pairs, extending the E-GRIDS algorithm [44] to support attributed productions that map syntactic structures to semantic SVA code. The framework uses beam search with minimum description length as a pruning heuristic, iteratively refining grammars through merge (generalizing interchangeable terms), chunk (capturing common phrase sequences), and augment (extending phrases with additional symbols) operators.

Beyond grammatical inference approaches, annotation-based methods emerged as an alternative paradigm for assertion automation. AutoSVA [45] introduces a lightweight annotation language that captures request-response transaction patterns in RTL module interfaces, enabling automated generation of formal verification testbenches. The framework defines a transaction model with directional relations and seven transaction attributes (e.g., `val`, `ack`, `transid`, `stable`, `data`, `active`, `transid_unique`) that map interface signals to automatically generated liveness and safety properties. Evaluated on the Ariane RISC-V core and OpenPiton modules, AutoSVA generated 236 properties from just 110 lines of annotations, discovering two previously unknown bugs and achieving 100% proof rates on verified modules, thus demonstrating the effectiveness of domain-specific annotation languages for assertion generation.

B. Zero-Shot and Few-Shot Prompting in Early LLM Approaches

Early works primarily employed zero-shot or few-shot prompting of general-purpose LLMs, often incorporating iterative refinement through human-in-the-loop feedback or formal verification results. The pioneering NL2SVA [36] framework introduces circuit-aware translation by feeding both Verilog designs and generic NL properties into GPT-4, employing few-shot prompting with explanation dictionaries for chain-of-thought reasoning. Its key innovation is a bidirectional feedback loop that integrates Cadence JasperGold to automatically validate generated SVAs and provide counterexamples for iterative refinement, although scalability remains limited by GPT-4's token constraints.

Extending this iterative refinement paradigm, AutoSVA2 [37] systematically teaches GPT-4 to generate correct SVAs through 23 iterations of FPV-based feedback, identifying and addressing four error categories with explicit prompting rules for temporal operator semantics and signal type distinctions. Notably, the framework demonstrates that GPT-4's generative capabilities enable the synthesis of correct assertions even for buggy RTL, exposing previously unknown design flaws while achieving a 6× improvement in statement coverage over AutoSVA [45].

To improve specification processing, ChIRAAG [38] adopts a two-stage strategy that first transforms unstructured specifications into structured JSON format with seven labeled fields, then generates and refines assertions through Synopsys VCS simulation feedback. Evaluated on six OpenTitan designs, it achieved 73% first-pass correctness with sub-15-second generation times while reproducing all existing assertions and discovering additional properties. Building on this structured approach, LAAG-RV [39] further optimizes iteration efficiency through a custom GPT-4 environment pre-trained on verification literature and a one-time Verilog loop that synchronizes signal names post-generation, thereby reducing prompt iterations compared to ChIRAAG while maintaining functional equivalence.

C. Retrieval-Augmented Generation for SVA Generation

Addressing end-to-end specification processing, AssertLLM [23] employs a multi-component architecture with specialized GPT-4 modules for multi-modal specification analysis, signal mapping via code interpreter, and RAG-based SVA generation covering width, connectivity, and functionality assertions. Its FPV-based evaluation, using golden RTL references, achieves 89% correctness on the `I2C` protocol, over 8% higher than vanilla GPT-4, demonstrating substantial gains from handling complete specification documents without manual extraction.

Beyond template-based extraction, AssertionForge [41] unifies specification and RTL information through a knowledge graph con-

structed with hardware-specific schema (e.g., modules, signals, ports, registers, FIFOs, clocks) and relation types (e.g., hasSection, defines, connectsTo, triggersOperation). The framework performs two-stage fusion via a customized graph RAG and a PyVerilog-based RTL parser with fuzzy name matching. It then employs a guided random walk with adaptive sampling to compute transition probabilities, balancing structural importance, architectural signals, and node exploration. This approach achieves up to 100% code coverage over AssertLLM on moderate designs.

Complementing the data collection strategy, LASSO [40] enhances data quality by systematically filtering for vacuity using nine theorems to discard non-meaningful properties from CWE, version control, and FPV sources. The framework employs RAG-based prompt generation, GPT-4o-driven SVA synthesis with k-shot learning, and iterative refinement via a feedback loop when coverage falls below 80%. Hierarchical property generation at the submodule level, followed by top-module aggregation, achieves 88% average coverage across eight common evaluation platforms and OpenTitan IPs, detecting five bugs in Hack@DAC'24.

D. SVA Generation with Multi-Source Data Augmentation and Fine-Tuning

Recognizing that standard LLMs struggle with domain-specific SVA syntax and semantics, recent frameworks employ RAG to inject relevant verification knowledge into the generation process. Hybrid-NL2SVA [35] introduces three optimization strategies: dynamic code-centric chunking, which preserves explanatory context and achieves an 18.8% improvement in functionality, hybrid retrieval, combining global semantic and keyword-guided operator retrieval, with a 32.7% improvement, and SVA operator-based rechecking for timing verification. The framework constructs a synthetic dataset of 4,070 SVAs from 67 hardware textbooks with prompt-guided explanations, achieving 59.05% improvement in functionality through fine-tuning Qwen2.5 Coder 7B Instructt.

The scarcity of high-quality assertion-specification training pairs has motivated the development of systematic multi-source data augmentation approaches. HADA [**?**] constructs 19,448 instruction-input-output triplets from three diverse sources of common weakness enumeration (CWE) entries with GPT-4-generated RTL and SVAs, bug fixes from nine open-source projects, and RISC-V modules with AutoSVA2-based symbolic analysis [37]. Fine-tuning via LoRA and PISSA with 4-bit quantization on LLaMA 3/3.1/3.2 and GPT-4o-mini yields 21.71% functionality pass@10 on hardware formal verification evaluation (FVEval), finite state machine (FSM) versus zero for base models, with version control data providing the most diverse training signal.

Pursuing broader domain adaptation, ChipNeMo [46] and its formal verification variant [43] apply domain-adaptive pretraining on LLaMA2 7B/13B/70B models using 24B tokens of proprietary chip design data with a domain-adaptive tokenizer extension and supervised fine-tuning. On a benchmark of 81 handcrafted assertions across 13 testbenches, ChipNeMo-70B achieves 85.1%/88.9% syntax pass in 1-shot/3-shot scenarios versus base LLaMA2-70B's 76.5%/74.1%, though trailing GPT-4's 98.8%. However, all models exhibit low BLEU scores ranging from 0.417 to 0.549, indicating substantial room for improvement in translating high-level intent into precise SVA implementations.

III. Testbench and Test Generation Automation

Effective test generation is a cornerstone of the hardware verification workflow, as it is essential for uncovering design flaws by exercising the design under test (DUT) with a comprehensive set of stimuli. The advent of LLMs has introduced a promising path to automate the generation of diverse and meaningful testbenches and stimuli directly from design specifications and RTL code. We summarize the survey papers in Fig. 3.

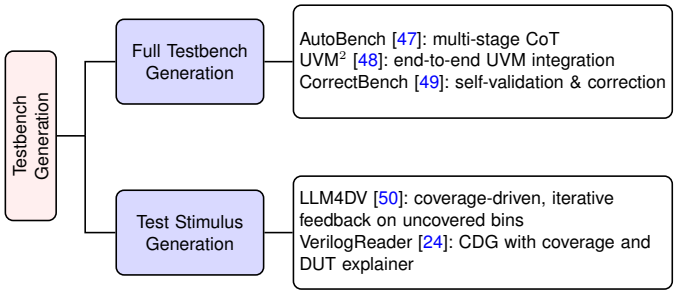

Fig. 3 Taxonomy of LLM-assisted testbench and test stimulus generation techniques.

A. Full Testbench Generation

One notable approach is presented in AutoBench [47], which introduces a hybrid testbench generation framework using LLMs. It strategically separates the generation process into Verilog drivers and Python-based checkers through a multi-stage chain-of-thought (CoT) approach. The framework directly addresses common LLM limitations such as laziness and hallucination by integrating self-enhancement mechanisms, including automatic debugging, scenario checking, and code standardization. To systematically measure the quality of the generated testbenches, AutoBench also proposes an automated evaluation framework (AutoEval) that assesses syntactic correctness, functional correctness, and coverage using mutant-based metrics.

While AutoBench employs a hybrid Verilog and Python methodology, UVM^2 [48] focuses on generating testbenches that adhere to established industry standards, such as the universal verification methodology (UVM). More specifically, UVM proposes an automated LLM-based framework for hardware verification that employs three specialized agents: an Analysis Agent for test planning, a Generation Agent for synthesizing hierarchical UVM components, and an Optimization Agent for iterative, coverage-driven stimulus refinement. Its generation process is distinguished by a dependency-driven workflow that follows component inter-dependencies and uses templates for regular components while leveraging guided LLM generation for more complex behavioral modules. Furthermore, the framework incorporates an iterative repair mechanism with coverage feedback loops to automatically correct errors and optimize testcases.

Beyond the initial generation of verification components, a paramount concern is ensuring the functional correctness of the testbench itself. CorrectBench [49] directly addresses this challenge by introducing an automatic testbench generation framework equipped with functional self-validation and self-correction mechanisms. Its validator employs a novel scenario-based approach, simulating multiple LLM-generated "imperfect" RTL designs and constructing an RTL-Scenario (RS) matrix to detect functional errors in the generated testbench systematically. When an error is identified, a conversational LLM-based corrector performs targeted fixes, utilizing detailed bug information through a two-stage reasoning process that determines the error's cause, location, and solution.

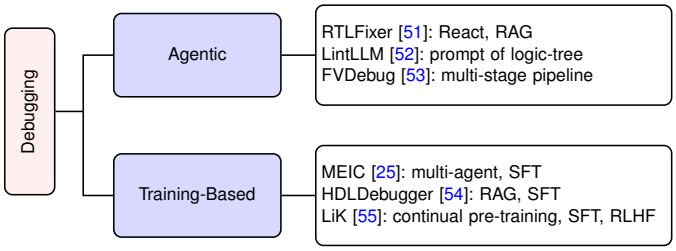

Fig. 4 Taxonomy of LLM-assisted RTL debugging techniques.

B. Test Stimulus Generation

In addition to frameworks that generate entire testbench structures, researchers have also explored the more targeted application of LLMs for automating test stimulus generation to close coverage gaps. The LLM4DV [50] framework, for instance, utilizes LLMs specifically to automate the generation of hardware test stimuli for design verification. This approach uses an iterative feedback loop where the LLM generates test inputs based on a coverage plan and receives feedback about uncovered bins. Its effectiveness is enhanced by a suite of six distinct prompting techniques, including missed-bin sampling, best-iterative-message sampling, dialogue restarting, and few-shot prompting, to guide the model toward producing more effective stimuli.

Similarly, VerilogReader [24] introduces an LLM-aided framework for coverage-directed test generation (CDG) that positions the LLM as a "verilog reader" to understand hardware code logic and generate test stimuli for unexplored branches. The framework employs two key modules: a coverage explainer that transforms simulator coverage reports into LLM-readable formats with natural language flags for uncovered lines, and a DUT Explainer that provides design descriptions and test guidance to enhance comprehension. Through a two-round prompt generation process, the LLM first analyzes the design and coverage status in natural language, then reformulates its response into a standardized JSON format for hardware stimulus generation.

IV. Automated RTL Debugging

The verification of RTL designs presents numerous challenges, with debugging standing out as a notoriously manual, expertise-driven, and time-intensive endeavor. Identifying the root cause of a failure can consume a substantial portion of the project timeline. The emergence of LLMs has catalyzed a transform-based approach to this problem, leveraging their sophisticated capabilities in code comprehension, logical reasoning, and error localization. This section reviews recent research works in LLM-aided debugging, which primarily explores two methodological paradigms: agentic frameworks that interact with external tools and specialized models enhanced through domain-specific training. We summarize the survey papers in Fig. 4.

A. Agentic Frameworks for Tool-Integrated Debugging

The first paradigm treats the general-purpose LLM as an autonomous reasoning agent within a larger, interactive system. In these frameworks, the model's internal parameters remain unchanged; instead, its powerful inference capabilities are orchestrated through sophisticated prompting and structured interaction with external EDA tools. The core innovation lies in designing a feedback loop where the LLM proposes actions or analyses, and the EDA tool provides ground-truth feedback, which guides the agent's subsequent steps.

A clear illustration of this is RTLFixer [51], which deploys an LLM agent to resolve Verilog syntax errors. Guided by ReAct (Reason+Act) prompting, the agent iteratively modifies the code and consults a compiler, using the compiler's error messages as feedback to refine its corrections. This agentic loop is augmented with a RAG to provide context on common error patterns. Similarly, FVDebug [53] automates the complex task of root-cause analysis for formal verification failures. It constructs a causal graph from tool-generated counter-examples and employs an agent, the "Insight Rover," to perform agentic exploration, navigate the graph, and form hypotheses. The entire process is driven by the LLM's ability to interpret and act upon the structured output of the formal verification tool. LintLLM [52] also embodies this approach by using a "Defect Tracker" agent that directs the LLM to fix potential defects and then re-runs a linting tool to observe the impact, thereby deducing the primary cause of multiple reported warnings. In all these cases, the LLM acts as an intelligent orchestrator, synergizing its reasoning abilities with the precision of established EDA toolchains.

B. Training-based Approaches

In contrast to agentic approaches, the second paradigm focuses on modifying the internal parameters of the LLM itself to embed domain-specific expertise. Through techniques like fine-tuning and reinforcement learning, these methods aim to create specialized models that possess intrinsic knowledge of HDL principles and debugging heuristics. This reduces reliance on extensive, real-time interaction with external tools, making the model a more self-contained expert.

This strategy is evident in MEIC [25], which employs a fine-tuned debug agent as a core component of its iterative debugging framework. The fine-tuning process refines a general model to account for the specific nuances of HDL error correction. A more intensive training regimen is detailed in HDLDebugger [54], which utilizes retrieval-augmented fine-tuning. The methodology involves first generating a synthetic dataset of buggy HDL code and then training the model on this data, enabling it to produce self-guided thought processes before generating a corrected code block. The culmination of this training-centric approach is exemplified by LiK [55], a model developed exclusively for localizing functional bugs. LiK undergoes a comprehensive, multi-stage training process involving continual pre-training, supervised fine-tuning (SFT), and reinforcement learning with human feedback (RLHF). The result is a highly specialized model that can accurately localize bugs using only the design specification and the buggy code, conspicuously eliminating the need for testbenches or EDA tools during its inference process. This signifies a fundamental shift of domain expertise from an external feedback loop into the intrinsic weights of the model itself.

V. Advanced and Collaborative Verification Framework

While the preceding sections focused on applying LLMs to discrete verification tasks, a growing body of research explores more holistic and advanced frameworks that move beyond single-task applications. These efforts aim to construct integrated systems that can handle more complex, multi-faceted verification challenges. This section surveys these advanced architectures, grouping them into three primary categories: multi-agent systems that simulate collaborative verification workflows, novel verification-driven training and optimization techniques designed to enhance core LLM capabilities for RTL generation, and the development of new verification paradigms,

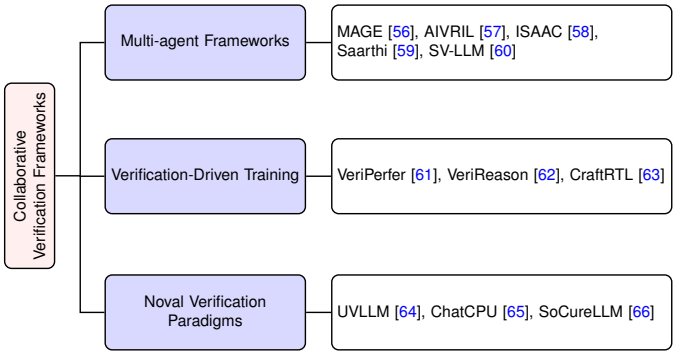

Fig. 5 Taxonomy of LLM-assisted advanced and collaborative verification frameworks.

including applications in the specialized domain of hardware security. The introduced research works are summarized in Fig. 5

A. Multi-Agent Frameworks for Collaborative Verification

This subsection covers papers that decompose the complex verification process into a series of specialized tasks, each handled by a distinct LLM-powered agent. These frameworks mimic human expert collaboration, employing agents for RTL generation, testbench creation, simulation, debugging, and formal analysis to achieve end-to-end automation.

Several frameworks have been proposed to tackle the end-to-end design and verification (D&V) process. One prominent example is MAGE [56], which introduces a multi-agent architecture that decomposes RTL design into specialized tasks handled by four agent types: RTL generation, testbench generation, simulation, and debugging agents. The system employs a novel high-temperature sampling strategy that generates multiple RTL candidates and ranks them using simulation-based scoring to identify the most promising solutions for further refinement. Additionally, MAGE implements a novel Verilog-state checkpoint checking mechanism that enables early detection of functional errors and delivers precise feedback for targeted fixes, significantly enhancing the functional correctness of the generated RTL code.

Along similar lines, AIVRIL [57] introduces a multi-agent framework that also focuses on the iterative refinement of LLM-generated RTL code. It combines automated syntax correction, AutoReview, and functional verification loops, AutoDV, to enhance code quality. The system utilizes a Code Agent for RTL generation and a Review Agent for error analysis, iteratively refining designs based on feedback from EDA tools, such as compilers and coverage analyzers. Operating in a ReAct-based paradigm, the framework is both tool-agnostic and LLM-agnostic, allowing for seamless integration with various models and EDA tools without requiring fine-tuning.

While MAGE and AIVRIL focus on the general RTL design and verification cycle, other multi-agent systems have been developed to target specific, critical sub-domains of the verification process. For instance, in the domain of simulation-based verification, ISAAC [58] combines an LLM-driven, multi-agent stimulus engine with micro-architectural awareness and a history-guided bug mechanism. This approach generates targeted tests that accelerate coverage convergence and expose corner cases. On the back end, it introduces a lightweight forward-snapshot mechanism and a decoupled co-simulation architecture between the instruction set simulator (ISS) and the design under test (DUT), enabling a single ISS to drive multiple DUTs in parallel.

Shifting the focus from simulation to formal methods, Saarthi [59] employs a multi-agent workflow where specialized AI agents collaborate sequentially to perform end-to-end formal verification. This includes a verification lead for planning, engineer agents for SVA generation, and critic agents for iterative refinement through reflection patterns. To mitigate common LLM issues, such as hallucination and infinite loops, the system implements human-in-the-loop (HIL) intervention, allowing human experts to provide corrective feedback when agents cannot converge autonomously.

Finally, to address the multifaceted challenge of security verification, SV-LLM [60] presents a multi-agent LLM framework that partitions the flow into specialized agents for verification question answering, security asset identification, threat modeling, test plan and property generation, vulnerability detection, and simulation-based bug validation. It couples a supervisor–orchestrator architecture for intent parsing, input completion, task planning, and coordinated execution with task-appropriate learning paradigms. These include in-context learning for lightweight reasoning tasks, fine-tuning for vulnerability detection, and RAG for knowledge-intensive interactions, creating a comprehensive solution for SoC security.

B. Verification-Driven Training for RTL Generation

The papers in this subsection focus on improving the RTL generation capabilities of LLMs by integrating verification results directly into the model training loop. By utilizing feedback from simulators and testbenches, these methods instruct the model to favor functionally correct code, thereby directly addressing the issues of hallucination and logical errors.

For instance, VeriPrefer [61] addresses the lack of sufficient functional verification data by introducing an automatic testbench generation pipeline. This pipeline decomposes the process and uses feedback from the Synopsys VCS simulator to reduce hallucination and ensure correctness. The method applies direct preference optimization (DPO), a reinforcement learning algorithm, to align Verilog code generation with functional correctness by training on preference pairs based on testbench outcomes, where code passing more test cases is labeled as preferred. Ultimately, this verification-driven approach integrates verification insights from testbenches into LLM training, consistently outperforming state-of-the-art baselines across multiple benchmarks.

Building on a similar principle, VeriReason [62] extends reinforcement learning approaches through guided reward proximal optimization (GRPO), which is tailored explicitly for RTL code generation. It combines testbench evaluations with structural heuristics to enhance specification-code alignment and eliminate false positives. Furthermore, the iterative use of GRPO embeds intrinsic self-checking and reasoning capabilities, enabling autonomous detection and correction of functional errors.

Alternatively, CraftRTL [63] focuses on curated supervised fine-tuning to enhance functional correctness. It employs a targeted code repair strategy, where common model errors identified from benchmark failures are programmatically injected into correct code to create a dataset of error-fix examples. This is complemented by a correct-by-construction method that generates guaranteed-correct data for non-textual representations, such as Karnaugh maps. By directly training the model to correct its own frequent mistakes and accurately interpret complex inputs, this approach enhances performance in Verilog Generation.

C. Novel Verification Paradigms

This subsection details papers that either propose fundamentally new verification workflows or apply LLMs to the critical and spe-

cialized domain of hardware security. These approaches integrate LLMs with established methodologies, such as UVM, invent new co-simulation techniques, or develop unique strategies for security policy verification.

On the functional verification front, researchers are developing novel workflows that integrate LLMs into complex simulation environments. One prominent example is UVLLM [64], which integrates LLMs with the industry-standard UVM. It proposes a four-stage pipeline combining pre-processing, UVM-based testing, dynamic error localization, and iterative repair. To ensure robustness, the framework employs a cost-efficient joint LLM-script approach for syntax correction, uses LLM-generated reference models for functional verification, and implements a rollback mechanism based on UVM scoreboard metrics to prevent error accumulation from LLM hallucinations.

Taking a different approach to system-level functional verification, ChatCPU [65] introduces a module-interchangeable verification paradigm (MVP). This technique pairs LLM-generated RTL modules with a cycle-accurate CPU reference model derived from a Processor Description Language. Each module is first unit-tested via the Verilator tool and then "swapped" into the reference model for co-simulation. This clever methodology enables targeted localization of discrepancies to the newly integrated RTL and effectively scales verification from the module to the full-CPU level.

Beyond functional correctness, another critical frontier is hardware security. SoCureLLM [66] pioneers the application of LLMs to this domain. To handle large system-on-chip (SoC) designs, it partitions them into manageable Verilog snippets and augments each with concise technical summaries, mitigating LLM context-length and memorization constraints. For verification, it introduces a tailored "dual hypotheses" prompting strategy, contrasting breach versus adherence analyses with iterative feedback to reach conclusive decisions about security policy compliance. The work also demonstrates how LLMs can derive actionable security policies by locating potential attack points via threat-model prompts and refining the results through manual fidelity checks to curate a practical policy database.

VI. Challenges and Future Directions

While LLMs hold immense potential to revolutionize circuit verification, the LLM practical applications are hindered by several challenges. In this section, we explore critical obstacles and solutions.

Challenge 1: Prompting Requires Domain Knowledge. One of the fundamental challenges in applying LLMs to circuit verification is that effective prompting requires substantial domain expertise, and different prompting strategies can significantly impact the quality of the results. Crafting prompts that elicit accurate and helpful responses from LLMs demands a deep understanding of verification methodologies, hardware design principles, and formal verification techniques. Even minor variations in prompt formulation, such as the order of instructions, the level of detail provided, or the specific terminology used, can lead to significantly different outcomes in tasks like assertion generation, bug localization, or testbench synthesis.

Automatic Prompt Optimization [67]–[70] emerges as a promising solution to mitigate the data scarcity problem in circuit verification tasks. Rather than relying on extensive fine-tuning with labeled datasets for each verification subtask, such as RTL bug localization, testbench generation, or verification IP configuration, automatic prompt optimization techniques can iteratively refine prompts to elicit better performance from pretrained LLMs. For example, Proof2silicon [71] proposed a small language model trained with proximal policy optimization (PPO) to edit prompts given to a frozen, larger LLM iteratively, enabling the generation of hardware designs. By systematically discovering more effective prompt formulations, these methods can steer LLMs toward higher accuracy on verification tasks while fully leveraging the hardware design and verification knowledge already embedded in large pretrained models.

Challenge 2: Data Scarcity and Costly Labeling. The scarcity of high-quality training data and the prohibitive cost of expert annotation represent critical bottlenecks in developing LLM-based circuit verification systems. Circuit verification demands deep domain expertise to create training datasets, whether for labeling correct versus incorrect assertions, annotating verification plans, classifying bug reports, or identifying optimal abstraction strategies for formal verification.

Except for synthetic data generation and data augmentation, as mentioned in [31], reinforcement Learning [72]–[74] offers a transformative approach to overcoming both data scarcity and labeling costs by leveraging the rich, automatable feedback inherent in verification workflows. Unlike supervised fine-tuning, which requires expensive expert annotations, RL can train models to develop sophisticated verification strategies by learning from reward signals naturally available during the verification process. For example, the assertion proving time or bug hunting for different methods provided by verification tools enables models to learn optimal decision-making for complex strategic choices, such as which properties to prove first, how to partition large designs, what abstractions to introduce, and when to switch between simulation and formal verification.

Challenge 3: Long Context Processing for Interactive Verification. Modern LLMs face significant limitations when working with HDL code bases that span thousands of lines and involve complex hierarchies. The core issue is that the complete design specification, testbench code, and supporting documentation can easily surpass context window capacity. This overflow causes critical information loss. The system forgets which approaches have been attempted, what errors have surfaced, and which partial solutions have shown potential. The problem intensifies in verification workflows that depend on maintaining detailed historical records. Without access to past attempts, failed strategies, and incremental progress, the model cannot engage in efficient, informed problem-solving.

Memory agents [75]–[77] address this fundamental constraint by providing persistent storage and intelligent retrieval mechanisms that extend beyond the LLM's fixed context window. In verification scenarios requiring multiple iterations, such as debugging complex code or refining generated text through successive attempts, a memory agent can track the entire problem-solving trajectory, including failed approaches, error patterns, successful partial solutions, and learned heuristics. This capability enables the verification process to build upon previous attempts rather than repeatedly encountering the same issues or losing track of promising directions, which can significantly improve efficiency and solution quality.

VII. Conclusion

In this paper, we survey the burgeoning field of LLM-assisted circuit verification, primarily covering topics such as assertion generation, testbench synthesis, debugging, and integrated verification frameworks. We have demonstrated that advanced methods, such as RAG, domain-specific fine-tuning, reinforcement learning, and multi-agent systems, are enabling a new paradigm of automation that alleviates the primary bottleneck in hardware verification. Ultimately, we also outline some challenges associated with using LLMs for circuit verification and propose future directions to address these challenges.

REFERENCES

[1] Y. Lu, C. Bai, Y. Zhao, Z. Zheng, Y. Lyu, M. Liu, and B. Yu, "DeepVerifier: Learning to Update Test Sequences for Coverage-Guided Verification," *ACM TODAES*, 2025.

[2] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks, "Fuzzing Hardware like Software," in *Proc. USENIX Security Symposium*, 2022, pp. 3237–3254.

[3] H. Liu, P. Liao, J. Huang, H.-L. Zhen, M. Yuan, T.-Y. Ho, and B. Yu, "Parallel Gröbner Basis Rewriting and Memory Optimization for Efficient Multiplier Verification," in *Proc. DATE*. IEEE, 2024, pp. 1–6.

[4] S. Zhou, W. Zhang, X. Zhang, Z. Jiang, H. You, and S. Cai, "Parallel Dynamic Partitioning for Datapath Combinational Equivalence Checking," in *Proc. DAC*. IEEE, 2025, pp. 1–7.

[5] Z. He, Y. Zuo, J. Jiang, H. Zheng, Y. Ma, and B. Yu, "OpenDRC: An Efficient Open-Source Design Rule Checking Engine with Hierarchical GPU Acceleration," in *Proc. DAC*. IEEE, 2023, pp. 1–6.

[6] J. Xu, Z. He, S. Yin, Y. Pu, W. Yu, and B. Yu, "EasyMRC: Efficient Mask Rule Checking via Representative Edge Sampling," *ACM TODAES*, vol. 30, no. 3, pp. 1–19, 2025.

[7] H. Sharangpani and M. Barton, "Statistical Analysis of Floating Point Flaw in the Pentium™ Processor (1994)," *Intel Corporation*, vol. 30, 1994.

[8] T. Kim. Intel's Alleged Security Flaw Could Cost Chipmaker a Lot of Money, Bernstein Says. [Online]. Available: https://www.cnbc.com/2018/01/03/intels-alleged-security-flaw-could-cost-chipmaker-a-lot-of-money-bernstein.html

[9] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "GPT-4 Technical Report," *arXiv preprint arXiv:2303.08774*, 2023.

[10] A. Liu, B. Feng, B. Xue, B. Wang, B. Wu, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan *et al.*, "Deepseek-V3 Technical Report," *arXiv preprint arXiv:2412.19437*, 2024.

[11] Anthropic, "Claude," https://claude.com/product/overview, 2025, accessed: 2025-10-13.

[12] Google DeepMind, "Gemini," https://gemini.google.com/, 2025, accessed: 2025-10-13.

[13] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention Is All You Need," *Proc. NIPS*, vol. 30, 2017.

[14] S. Liu, W. Fang, Y. Lu, J. Wang, Q. Zhang, H. Zhang, and Z. Xie, "RTLCoder: Fully Open-Source and Efficient LLM-Assisted RTL Code Generation Technique," *IEEE TCAD*, 2024.

[15] Z. Pei, H.-L. Zhen, M. Yuan, Y. Huang, and B. Yu, "BetterV: Controlled Verilog Generation with Discriminative Guidance," in *Proc. ICML*, 2024, pp. 40 145–40 153.

[16] X. Yao, W. Zhao, Q. Sun, C. Zhuo, and B. Yu, "High-level Synthesis Directives Design Optimization via Large Language Model," *ACM TODAES*, vol. 30, no. 5, pp. 1–24, 2025.

[17] X. Yao, Y. Wang, X. Li, Y. Lian, R. Chen, L. Chen, M. Yuan, H. Xu, and B. Yu, "RTLRewriter: Methodologies for Large Models Aided RTL Code Optimization," in *Proc. ICCAD*, 2024, pp. 1–7.

[18] H. Wu, Z. He, X. Zhang, X. Yao, S. Zheng, H. Zheng, and B. Yu, "ChatEDA: A Large Language Model Powered Autonomous Agent for EDA," *IEEE TCAD*, vol. 43, no. 10, pp. 3184–3197, 2024.

[19] Z. Wang, Y. Shen, X. Yao, W. Zhao, Y. Bai, F. Farnia, and B. Yu, "ChatPattern: Layout Pattern Customization via Natural Language," in *Proc. DAC*, 2024, pp. 1–6.

[20] X. Yao, J. Jiang, Y. Zhao, P. Liao, Y. Lin, and B. Yu, "Evolution of Optimization Algorithms for Global Placement via Large Language Models," *arXiv preprint arXiv:2504.17801*, 2025.

[21] G. Chen, H. Yang, B. Yu, and H. Ren, "Intelligent OPC Engineer Assistant for Semiconductor Manufacturing," in *Proc. AAAI*, vol. 39, no. 22, 2025, pp. 23 144–23 151.

[22] Y. Pu, Z. He, T. Qiu, H. Wu, and B. Yu, "Customized Retrieval Augmented Generation and Benchmarking for EDA Tool Documentation QA," in *Proc. ICCAD*, 2024, pp. 1–9.

[23] Z. Yan, W. Fang, M. Li, M. Li, S. Liu, Z. Xie, and H. Zhang, "AssertLLM: Generating Hardware Verification Assertions from Design Specifications via Multi-LLMs," in *Proc. ASPDAC*, 2025, pp. 614–621.

[24] R. Ma, Y. Yang, Z. Liu, J. Zhang, M. Li, J. Huang, and G. Luo, "VerilogReader: LLM-Aided Hardware Test Generation," in *Proc. LAD*. IEEE, 2024, pp. 1–5.

[25] K. Xu, J. Sun, Y. Hu, X. Fang, W. Shan, X. Wang, and Z. Jiang, "MEIC: Re-Thinking RTL Debug Automation Using LLMs," in *Proc. ICCAD*, 2024, pp. 1–9.

[26] Y. Zhang, H.-L. Zhen, Z. Pei, Y. Lian, L. Yin, M. Yuan, and B. Yu, "DiLA: Enhancing LLM Tool Learning with Differential Logic Layer," *arXiv preprint arXiv:2402.11903*, 2024.

[27] Wilson Research Group, "2024 Wilson Research Group IC/ASIC Functional Verification Trend Report," Siemens Digital Industries Software, White Paper, 2024, accessed 2025-10-15. [Online]. Available: https://resources.sw.siemens.com/en-US/white-paper-2024-wilson-research-group-ic-asic-functional-verification-trend-report/

[28] M. Abdollahi, S. F. Yeganli, M. Baharloo, and A. Baniasadi, "Hardware Design and Verification with Large Language Models: A Scoping Review, Challenges, and Open Issues," *Electronics*, vol. 14, no. 1, p. 120, 2024.

[29] S. Alsaqer, S. Alajmi, I. Ahmad, and M. Alfailakawi, "The Potential of LLMs in Hardware Design," *Proc. JER*, 2024.

[30] J. Pan, G. Zhou, C.-C. Chang, I. Jacobson, J. Hu, and Y. Chen, "A Survey of Research in Large Language Models for Electronic Design Automation," *ACM TODAES*, vol. 30, no. 3, pp. 1–21, 2025.

[31] W. Fang, J. Wang, Y. Lu, S. Liu, Y. Wu, Y. Ma, and Z. Xie, "A Survey of Circuit Foundation Models: Foundation AI Models for VLSI Circuit Design and EDA," *arXiv preprint arXiv:2504.03711*, 2025.

[32] R. Zhong, X. Du, S. Kai, Z. Tang, S. Xu, H.-L. Zhen, J. Hao, Q. Xu, M. Yuan, and J. Yan, "LLM4EDA: Emerging Progress in Large Language Models for Electronic Design Automation," *arXiv preprint arXiv:2401.12224*, 2023.

[33] N. Wu, Y. Li, H. Yang, H. Chen, S. Dai, C. Hao, C. Yu, and Y. Xie, "Survey of Machine Learning for Software-Assisted Hardware Design Verification: Past, Present, and Prospect," *ACM TODAES*, vol. 29, no. 4, pp. 1–42, 2024.

[34] C. B. Harris and I. G. Harris, "GLAST: Learning Formal Grammars to Translate Natural Language Specifications into Hardware Assertions," in *Proc. DATE*. IEEE, 2016, pp. 966–971.

[35] W. Xiao, D. Ekberg, S. Garg, and R. Karri, "Hybrid-NL2SVA: Integrating RAG and Finetuning for LLM-Based NL2SVA," in *Proc. MLCAD*, 2025, pp. 1–10.

[36] C. Sun, C. Hahn, and C. Trippel, "Towards Improving Verification Productivity with Circuit-Aware Translation of Natural Language to SystemVerilog Assertions," in *Proc. DAV*, 2023.

[37] M. Orenes-Vera, M. Martonosi, and D. Wentzlaff, "Using LLMs to Facilitate Formal Verification of RTL," *arXiv preprint arXiv:2309.09437*, 2023.

[38] B. Mali, K. Maddala, V. Gupta, S. Reddy, C. Karfa, and R. Karri, "ChIRAAG: ChatGPT Informed Rapid and Automated Assertion Generation," in *Proc. ISVLSI*. IEEE, 2024, pp. 680–683.

[39] K. Maddala, B. Mali, and C. Karfa, "LAAG-RV: LLM Assisted Assertion Generation for RTL Design Verification," in *Proc. ITC*. IEEE, 2024, pp. 1–6.

[40] D. R. Ankireddy, S. Paria, A. Dasgupta, S. Ray, and S. Bhunia, "LASSO: LLM-Aided Security Property Generation for Assertion-Based SoC Verification," in *Proc. MLCAD*. IEEE, 2025, pp. 1–10.

[41] Y. Bai, G. B. Hamad, S. Suhaib, and H. Ren, "AssertionForge: Enhancing Formal Verification Assertion Generation with Structured Representation of Specifications and RTL," *arXiv preprint arXiv:2503.19174*, 2025.

[42] W. Fu, Y. Wang, Z. Lu, X. Guo, and G. Qu, "HADA: Leveraging Multi-Source Data to Train Large Language Models for Hardware Security Assertion Generation," in *Proc. MLCAD*. IEEE, 2025, pp. 1–7.

[43] M. Liu, M. Kang, G. B. Hamad, S. Suhaib, and H. Ren, "Domain-Adapted LLMs for VLSI Design and Verification: A Case Study on Formal Verification," in *Proc. VTS*. IEEE, 2024, pp. 1–4.

[44] G. Petasis, G. Paliouras, V. Karkaletsis, C. Halatsis, and C. D. Spyropoulos, "e-GRIDS: Computationally Efficient Gramatical Inference from Positive Examples," *Grammars*, vol. 7, pp. 69–110, 2004.

[45] M. Orenes-Vera, A. Manocha, D. Wentzlaff, and M. Martonosi, "AutoSVA: Democratizing Formal Verification of RTL Module Interactions," in *Proc. DAC*. IEEE, 2021, pp. 535–540.

[46] 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.

[47] R. Qiu, G. L. Zhang, R. Drechsler, U. Schlichtmann, and B. Li, "AutoBench: Automatic Testbench Generation and Evaluation Using

LLMs for HDL Design," in *Proc. MLCAD*. ACM/IEEE, 2024, pp. 1–10.
[48] J. Ye, Y. Hu, K. Xu, D. Pan, Q. Chen, J. Zhou, S. Zhao, X. Fang, X. Wang, N. Guan *et al.*, "From Concept to Practice: an Automated LLM-aided UVM Machine for RTL Verification," *Proc. ICCAD*, pp. 1–8, 2025.
[49] R. Qiu, G. L. Zhang, R. Drechsler, U. Schlichtmann, and B. Li, "CorrectBench: Automatic Testbench Generation with Functional Self-Correction using LLMs for HDL Design," in *Proc. DATE*. IEEE, 2025, pp. 1–7.
[50] Z. Zhang, G. Chadwick, H. McNally, Y. Zhao, and R. Mullins, "LLM4DV: Using large language models for hardware test stimuli generation," in *Proc. FCCM*, 2025.
[51] Y. Tsai, M. Liu, and H. Ren, "RTLFixer: Automatically Fixing RTL Syntax Errors with Large Language Model," in *Proc. DAC*, 2024, pp. 1–6.
[52] Z. Fang, R. Chen, Z. Yang, Y. Guo, H. Dai, and L. Wang, "LintLLM: An Open-Source Verilog Linting Framework Based on Large Language Models," 2025, p. 673–680.
[53] Y. Bai, G. B. Hamad, C.-T. Ho, S. Suhaib, and H. Ren, "FVDebug: An LLM-Driven Debugging Assistant for Automated Root Cause Analysis of Formal Verification Failures," *arXiv preprint arXiv:2510.15906*, 2025.
[54] 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," *ACM TODAES*, 2024.
[55] B. Yao, N. Wang, J. Zhou, X. Wang, H. Gao, Z. Jiang, and N. Guan, "Location Is Key: Leveraging Large Language Model for Functional Bug Localization in Verilog," in *Proc. DAC*, 2025, pp. 1–7.
[56] Y. Zhao, H. Zhang, H. Huang, Z. Yu, and J. Zhao, "Mage: A Multi-Agent Engine for Automated RTL Code Generation," in *Proc. DAC*. IEEE, 2025, pp. 1–7.
[57] H. Sami, P.-E. Gaillardon, V. Tenace *et al.*, "AIVRIL: AI-Driven RTL Generation with Verification In-the-Loop," *arXiv preprint arXiv:2409.11411*, 2024.
[58] J. Sun, Y. Hu, D. You, Y. Du, H. Wang, X. Fang, W. Shan, N. Guan, and Z. Jiang, "ISAAC: Intelligent, Scalable, Agile, and Accelerated CPU Verification via LLM-Aided FPGA Parallelism," *arXiv preprint arXiv:2510.10225*, 2025.
[59] A. Kumar, D. N. Gadde, K. K. Radhakrishna, and D. Lettnin, "Saarthi: The First AI Formal Verification Engineer," *Proc. DVCON*, 2025.
[60] D. Saha, S. Tarek, H. A. Shaikh, K. T. Hasan, P. S. Nalluri, M. A. Hasan, N. Alam, J. Zhou, S. K. Saha, M. Tehranipoor *et al.*, "SV-LLM: An Agentic Approach for SoC Security Verification Using Large Language Models," *arXiv preprint arXiv:2506.20415*, 2025.
[61] N. Wang, B. Yao, J. Zhou, Y. Hu, X. Wang, N. Guan, and Z. Jiang, "Insights from Verification: Training a Verilog Generation LLM with Reinforcement Learning with Testbench Feedback," *arXiv preprint arXiv:2504.15804*, 2025.
[62] Y. Wang, G. Sun, W. Ye, G. Qu, and A. Li, "VeriReason: Reinforcement Learning with Testbench Feedback for Reasoning-Enhanced Verilog Generation," *arXiv preprint arXiv:2505.11849*, 2025.
[63] M. Liu, Y.-D. Tsai, W. Zhou, and H. Ren, "CraftRTL: High-Quality Synthetic Data Generation for Verilog Code Models with Correct-by-Construction Non-Textual Representations and Targeted Code Repair," *Proc. ICLR*, 2025.
[64] Y. Hu, J. Ye, K. Xu, J. Sun, S. Zhang, X. Jiao, D. Pan, J. Zhou, N. Wang, W. Shan *et al.*, "UVLLM: An Automated Universal RTL Verification Framework Using LLMs," in *Proc. DAC*, 2025.
[65] X. Wang, G.-W. Wan, S.-Z. Wong, L. Zhang, T. Liu, Q. Tian, and J. Ye, "ChatCPU: An Agile CPU Design and Verification Platform with LLM," in *Proc. DAC*, 2024, pp. 1–6.
[66] S. Tarek, D. Saha, S. K. Saha, M. Tehranipoor, and F. Farahmandi, "SoCureLLM: An LLM-Driven Approach for Large-Scale System-on-Chip Security Verification and Policy Generation," in *Proc. HOST*. IEEE, 2025, pp. 335–345.
[67] Z. Li, B. Peng, P. He, M. Galley, J. Gao, and X. Yan, "Guiding Large Language Models via Directional Stimulus Prompting," *Proc. NIPS*, vol. 36, pp. 62 630–62 656, 2023.
[68] X. Wang, C. Li, Z. Wang, F. Bai, H. Luo, J. Zhang, N. Jojic, E. P. Xing, and Z. Hu, "Promptagent: Strategic planning with language models enables expert-level prompt optimization," *Proc. ICLR*, 2024.
[69] J. Xiang, J. Zhang, Z. Yu, X. Liang, F. Teng, J. Tu, F. Ren, X. Tang, S. Hong, C. Wu *et al.*, "Self-supervised prompt optimization," *Proc. ACL*, 2025.
[70] L. A. Agrawal, S. Tan, D. Soylu, N. Ziems, R. Khare, K. Opsahl-Ong, A. Singhvi, H. Shandilya, M. J. Ryan, M. Jiang *et al.*, "Gepa: Reflective prompt evolution can outperform reinforcement learning," *arXiv preprint arXiv:2507.19457*, 2025.
[71] M. Jha, J. Wan, and D. Chen, "Proof2Silicon: Prompt Repair for Verified Code and Hardware Generation via Reinforcement Learning," *arXiv preprint arXiv:2509.06239*, 2025.
[72] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," *arXiv preprint arXiv:1707.06347*, 2017.
[73] R. Rafailov, A. Sharma, E. Mitchell, C. D. Manning, S. Ermon, and C. Finn, "Direct Preference Optimization: Your Language Model Is Secretly a Reward Model," *Proc. NIPS*, vol. 36, pp. 53 728–53 741, 2023.
[74] Z. Shao, P. Wang, Q. Zhu, R. Xu, J. Song, X. Bi, H. Zhang, M. Zhang, Y. Li, Y. Wu *et al.*, "DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models," *arXiv preprint arXiv:2402.03300*, 2024.
[75] P. Chhikara, D. Khant, S. Aryan, T. Singh, and D. Yadav, "Mem0: Building Production-Ready AI Agents with Scalable Long-Term Memory," *arXiv preprint arXiv:2504.19413*, 2025.
[76] Y. Wang and X. Chen, "Mirix: Multi-Agent Memory System for LLM-Based Agents," *arXiv preprint arXiv:2507.07957*, 2025.
[77] S. Yan, X. Yang, Z. Huang, E. Nie, Z. Ding, Z. Li, X. Ma, K. Kersting, J. Z. Pan, H. Schütze *et al.*, "Memory-R1: Enhancing Large Language Model Agents to Manage and Utilize Memories via Reinforcement Learning," *arXiv preprint arXiv:2508.19828*, 2025.