

A Survey on LLM-Integrated Hardware Design Verification

Hao Zheng*, Jaime Rafael Imperial*, Bardia Nadimi*, Xiangfei Kong*

*Bellini College of Cybersecurity, AI, and Computing, University of South Florida
Tampa, FL, USA

Email: {haozheng, imperialj, bnadimi, xiangfeikong}@usf.edu

Abstract—Large language models (LLMs) are increasingly being integrated into hardware verification to automate specification interpretation, verification-artifact generation, debugging, formal reasoning, and tool orchestration. This survey provides a systematic review of LLM-assisted hardware functional verification across SystemVerilog assertion generation, stimulus and testbench generation, bug localization and design repair, model checking and equivalence checking, SAT/SMT optimization, and emerging agentic verification workflows. We organize the literature by methodology, verification objective, tool interaction, benchmark, and evaluation criterion, and examine both inference-time techniques—including prompting, retrieval, structured reasoning, and agentic workflows—and training-time adaptation. Across these areas, a common pattern emerges: LLMs are most effective as semantic reasoning, search, and orchestration components embedded within verification-aware workflows, while simulators, formal engines, coverage tools, and solvers provide executable feedback and correctness evidence. However, tool acceptance alone does not establish verification correctness, since assertions, tests, repairs, or proofs may satisfy available checks without faithfully capturing the complete design intent. We therefore identify semantic alignment between specifications and verification evidence, scalable integration with deterministic tools, generalization to unseen designs, and rigorous evaluation of correctness, cost, robustness, and human effort as key challenges. Finally, we discuss emerging directions toward specification-centered, neuro-symbolic, and persistent agentic verification systems that combine LLM flexibility with independently checkable verification evidence.

Index Terms—Large language models, hardware verification, formal verification, electronic design automation.

I. INTRODUCTION

Computing systems have become indispensable to modern society, supporting a wide range of applications with increasingly stringent requirements for functionality, performance, reliability, safety, security, and energy efficiency. At the hardware level, System-on-Chip (SoC) designs have become a dominant architecture, with modern chips integrating tens to hundreds of billions of transistors and increasingly heterogeneous and tightly coupled functional blocks [1, 2]. This growing hardware complexity, coupled with society’s increasing reliance on computing systems in safety-, security-, and mission-critical applications, makes ensuring hardware correctness and dependability increasingly challenging and essential. Consequently, rigorous hardware verification is a fundamental requirement for identifying design errors and establishing confidence that hardware implementations conform to their intended specifications before deployment.

Consequently, hardware verification has become one of the most challenging, costly, and resource-intensive stages of the IC design process [3], aiming to establish that hardware implementations conform to their functional requirements and design intent. While verification spans both pre-silicon verification and post-silicon validation, pre-silicon verification is particularly critical because defects that escape into fabrication can require costly silicon re-spins, cause substantial schedule delays, and incur significant financial losses [4]. Yet, growing hardware complexity has made pre-silicon verification increasingly difficult to scale. The 2024 Wilson Research Group Functional Verification Study [5] reports that peak IC/ASIC verification teams average over 14 engineers, compared with approximately 12 design engineers, while design engineers spend about 50% of their time on verification. Overall, verification is estimated to consume 60–70% of IC development cost and schedule [6, 7], highlighting the need for greater automation and productivity in modern hardware verification.

Large Language Models (LLMs) [8, 9, 10] are increasingly used across the Electronic Design Automation (EDA) stack because they can interpret and generate diverse hardware artifacts, including specifications, RTL, assertions, testbenches, and tool logs. Their flexible natural-language interface enables reasoning across heterogeneous information and interaction with EDA tools through retrieval-augmented and agentic workflows [11, 12]. Consequently, LLMs have been explored for specification analysis [13, 14], RTL and hardware generation [15, 16, 17, 18], design-space exploration [19], verification planning [20], assertion and testbench generation [21, 22, 23, 24, 25], bug localization and repair [26, 27], and autonomous verification [28]. Recent approaches increasingly integrate LLMs with simulators, formal verification engines, synthesis tools, and external knowledge to ground generation and enable iterative refinement [28, 29]. Despite substantial progress, significant challenges remain in ensuring correctness, mitigating hallucinations, preserving design intent, scaling to industrial-scale designs, and establishing rigorous evaluation methodologies tailored to hardware engineering tasks motivating systematic research into trustworthy and autonomous LLM-integrate hardware verification.

A. Why a New Survey Focused on LLM-Driven Verification?

The rapid adoption of Large Language Models (LLMs) has spurred a growing body of survey literature on their

applications in Electronic Design Automation (EDA). Existing surveys consistently identify LLMs as a transformative technology across the chip design flow, but they differ substantially from the scope of this survey. Pan et al. [11] provide a broad EDA-wide treatment spanning system-level design, RTL design, logic synthesis, physical design, and analog design, with verification appearing as one component of a much larger design-automation landscape. Xu et al. [12] similarly frame LLMs as a general technology across the chip flow, including specification optimization, HDL generation, testing, debugging, synthesis, and back-end task execution. Consequently, they relegate functional verification to brief subsections, missing technical nuances like Universal Verification Methodology (UVM) infrastructure composition. Yang et al. [30] narrow the focus to Verilog code generation, explicitly organizing the literature around models, datasets, metrics, generation techniques, and alignment, but their inclusion criteria exclude studies focused solely on repair or testing without generation. The recent Agentic EDA survey [28] targets autonomous orchestration of the RTL-to-GDSII flow and emphasizes perception–cognition–action loops, tool execution, and the sim-to-silicon gap rather than the detailed methodology of verification tasks. The work in [31] focuses primarily on vulnerability identification (e.g., Common Weakness Enumerations) and threat modeling. While they leverage testbenches and assertions, their core metric is security compliance rather than general functional coverage closure. A verification-specific survey by Liu et al. [32] is closest in spirit, but it is compact and organized as a high-level overview of assertion generation, testbench generation, debugging, and holistic verification frameworks. Consequently, these works do not provide comprehensive coverage of the rapidly expanding landscape of LLM-driven hardware functional verification, including emerging techniques for testbench generation, assertion generation, coverage closure, bug localization, debugging, design repair, verification agents, benchmark development, and evaluation methodologies. This gap has become increasingly pronounced as the field has grown rapidly since 2024, with dozens of verification-focused studies appearing in leading EDA venues. As a result, researchers and practitioners currently lack a unified and systematic understanding of how LLMs are being applied across the full verification workflow, the associated benchmark ecosystems, and the remaining technical challenges. Table I summarizes the structural coverage gaps of existing literature across the four pillars of the proposed survey paper.

This survey provides a comprehensive review of recent research on LLM-integrated hardware functional verification, a rapidly emerging research area with unique workflows, artifacts, evaluation criteria, and tool-integration requirements that are not adequately captured by broader LLM-for-EDA surveys. Unlike prior reviews that treat verification as a secondary component of the design flow, this survey systematically organizes the literature across the complete verification pipeline, including SystemVerilog Assertion (SVA) generation, testbench and stimulus generation, bug localization, debugging, design

repair, and autonomous verification agents. In addition, the survey provides a unified comparison of datasets, benchmarks, tools, and evaluation frameworks used throughout the field, identifies common limitations such as hallucination, tool orchestration, and scalability, and highlights emerging research directions including multimodal LLMs, agentic verification systems, and human-in-the-loop verification workflows. By treating hardware verification as an integrated feedback-driven process rather than a collection of isolated applications, this survey establishes a comprehensive reference for researchers and practitioners and provides a roadmap for advancing LLM-integrated verification from proof-of-concept demonstrations to dependable engineering practice.

B. Paper Organization

The organization of this paper is as follows. Section II gives background necessary to the following sections. Section III surveys the latest research on SVA generations using LLMs. Section IV surveys the latest research on stimulus and testbench generation using LLMs. Section V surveys the latest research on bug localization and design repair using LLMs. Section VI surveys the latest research on formal verification using LLMs. Section VII concludes the paper, offers our views of current challenges to LLM-based verification methods, and highlight promising future research directions.

II. BACKGROUND

A. Overview of Hardware Design Flow

A typical hardware design flow, illustrated in Fig. 1, progresses through increasingly concrete representations from system requirements to fabricated silicon. System and architectural specifications define functional requirements, interfaces, and performance, power, and area (PPA) objectives, which are refined into microarchitecture and synthesizable RTL. The RTL is transformed through logic synthesis into a gate-level netlist and subsequently through physical design, including floorplanning, placement, clock-tree synthesis, and routing, followed by physical verification and signoff before tapeout and fabrication. Fabricated devices then undergo post-silicon validation for functionality, performance, and power.

Verification is a cross-cutting activity throughout this flow rather than an isolated design stage. At the RTL and gate levels, functional verification employs simulation, constrained-random testing, assertions, functional coverage, formal methods, and RTL-to-netlist equivalence checking. Physical implementation is validated through static timing analysis, design-rule checking (DRC), layout-versus-schematic (LVS) checking, and power- and signal-integrity analysis, followed by post-silicon validation. Because verification can expose design defects or specification violations at any stage, the flow is inherently iterative, with verification results potentially triggering revisions to earlier design stages. This survey focuses on functional (logic) verification at the RTL and gate levels.

TABLE I

COMPARISON OF EXISTING SURVEYS AND THE PROPOSED SURVEY. A CHECKMARK (✓) INDICATES SUBSTANTIAL COVERAGE OF A TOPIC, WHILE △ DENOTES PARTIAL OR HIGH-LEVEL DISCUSSION.

Survey	EDA-wide	RTL Gen.	SVA Gen.	Testbench / Stimulus	Debug	Repair	Benchmarks	Verification-Centric	Scope
Pan et al. [11]	✓	✓	△	△	△	△	△	×	Broad survey of LLM applications across EDA workflow
Xu et al. [12]	✓	✓	△	△	△	△	×	×	Overview of LLMs in design, testing, optimization, and automation
Yang et al. [30]	×	✓	×	×	×	×	✓	×	Focused on Verilog/RTL code generation models, datasets, and evaluation
Zang et al. [28]	✓	✓	×	×	△	△	×	×	Agentic EDA and autonomous RTL-to-GDSII workflows
Liu et al. [32]	×	×	✓	✓	✓	△	△	✓	Overview of LLM-assisted circuit verification
This Survey	×	△	✓	✓	✓	✓	✓	✓	Comprehensive survey of LLMs for verification, including assertion generation, testbench/stimulus generation, bug localization, debugging, RTL repair, datasets, benchmarks, evaluation methodologies, tool integration, and open challenges

Typical Hardware Design Flow with Verification

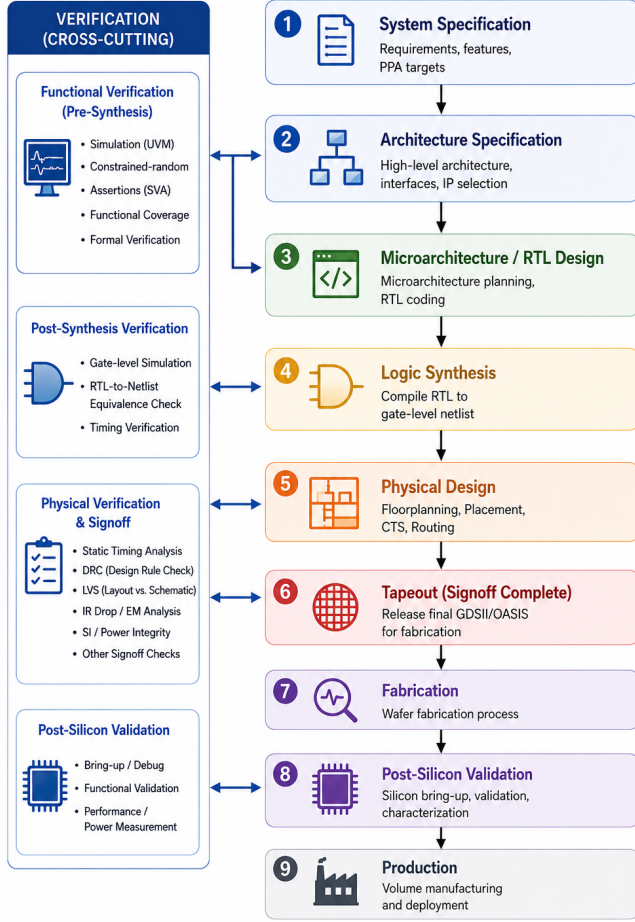


Fig. 1. A simplified overview of hardware design flow.

B. Verification Concepts and Techniques

Hardware functional verification aims to establish that an implementation conforms to its intended functional and temporal behavior. It primarily combines simulation-based verification, formal methods, and hardware-assisted techniques. Simulation remains the dominant approach, exercising the design with directed or constrained-random stimuli and using coverage metrics to guide verification progress; methodologies such as the Universal Verification Methodology (UVM) support reusable and modular verification environments [33]. Formal

methods, including model checking and equivalence checking, mathematically establish specified properties or behavioral equivalence and can expose corner cases that are difficult to reach through simulation [34, 35, 36]. These techniques commonly rely on Boolean SAT [37, 38] and Satisfiability Modulo Theories (SMT) solvers [39, 40], while emulation and FPGA-based prototyping accelerate the execution of large workloads that are impractical to explore through software simulation.

A typical functional-verification flow develops a testbench that generates stimulus, drives the design under test (DUT), monitors responses, and checks observed behavior against expected results. Verification engineers must also develop test scenarios, assertions, checkers, and coverage models, followed by iterative simulation, failure diagnosis, and regression testing. Coverage metrics help identify untested functionality and guide subsequent verification, but no single metric can establish verification completeness [33]. Moreover, verification remains difficult to scale because modern designs exhibit enormous input and state spaces, long temporal behaviors, and complex interactions among components and software. Simulation may fail to exercise difficult-to-reach behaviors, while formal methods face state-space explosion and scalability limitations [34, 41]. Developing verification artifacts, diagnosing failures determining whether apparent coverage gaps represent genuine verification deficiencies also requires substantial hardware expertise and engineering effort. These challenges motivate increasing interest in automation, including LLM-based methods for generating verification artifacts, exploring design behavior, and assisting verification and debugging.

C. Large Language Models

Large Language Models (LLMs) are deep learning models trained on large-scale text and source-code corpora to learn general-purpose representations of language, programming constructs, and domain knowledge. Most modern LLMs are based on the Transformer architecture [42], whose self-attention mechanism enables effective modeling of long-range dependencies and highly parallelized training. LLMs are typically developed through large-scale pre-training followed by post-training, such as supervised fine-tuning and preference- or feedback-based optimization, which adapts general capabilities toward instruction following and task-specific behavior.

LLMs can be adapted to downstream tasks through prompting, including few-shot and reasoning-oriented strategies such

as chain-of-thought prompting [43]. These capabilities are particularly relevant to hardware verification, where an LLM may need to interpret specifications, identify relationships among signals and modules, generate assertions or test scenarios, and reason about verification failures. However, LLM-generated reasoning and artifacts are not correctness guarantees, motivating the use of external knowledge and deterministic verification mechanisms.

Retrieval-Augmented Generation (RAG) [44] supplements model knowledge with task-specific information retrieved at inference time. For hardware verification, retrieved context may include specifications, RTL modules, interface descriptions, protocol documentation, assertions, verification plans, or debugging records. LLMs can also interact with external tools, using approaches such as ReAct [45] and Toolformer [46], to invoke simulators, formal engines, compilers, coverage analyzers, or other EDA tools and use their outputs to refine subsequent actions. This tool-grounded interaction is particularly important for verification because deterministic tools can independently evaluate generated artifacts.

Agentic LLM systems extend these capabilities by decomposing complex tasks into multiple steps, maintaining intermediate state, invoking tools, and iteratively refining results; multi-agent frameworks can further assign specialized roles such as planning, generation, verification, and debugging [47]. These workflows, however, must manage the limited context available to each model invocation. Large RTL hierarchies, specifications, testbenches, waveforms, and tool logs can exceed practical context limits, motivating long-context models, retrieval, summarization, and structured representations. Together, reasoning, retrieval, tool use, and agentic orchestration provide the foundation for applying LLMs to hardware verification, while deterministic verification tools remain essential for validating their outputs.

D. LLMs for EDA

The integration of Large Language Models (LLMs) into Electronic Design Automation (EDA) flows is enabling increasingly autonomous and machine-augmented hardware design workflows. Early efforts such as VeriGen demonstrated domain adaptation by fine-tuning open-source models on Verilog corpora [1]. Subsequent approaches have explored natural-language-driven RTL generation and iterative refinement, with frameworks such as AutoChip incorporating compiler and simulation feedback into multi-turn generation loops to identify and correct syntactic and functional errors [2], while RTLcoder demonstrated the effectiveness of domain-specific training data and lightweight open-source models on benchmarks such as RTLLM and VerilogEval [3].

Hardware verification is another major application area. LLM-based approaches have been developed to automate verification collateral generation, including testbenches through frameworks such as AutoBench [4], and assertion generation from natural-language design specifications through systems such as AssertLLM [5]. These methods reduce the manual effort required to translate design intent into executable veri-

fication artifacts while retaining conventional simulation and formal-verification tools as the final validation mechanisms.

Beyond RTL generation and verification, LLMs are being applied across the EDA optimization stack. At the HLS level, approaches such as C2HLSC and HLSPilot use LLMs to transform and optimize HLS code [6], [7]. At the physical-design level, agent-based frameworks such as ChatEDA employ LLMs to decompose design tasks, generate tool scripts, and orchestrate conventional EDA tools from RTL through physical implementation [8]. Collectively, these efforts exploit three capabilities of LLMs—cross-domain semantic mapping, pattern recognition, and automated reasoning—to bridge high-level design intent with executable artifacts, including HDL code, verification programs, and EDA scripts. Rather than replacing established EDA algorithms, LLMs increasingly serve as intelligent interfaces and assistants that automate repetitive tasks, guide design-space exploration, and enable engineers to focus on architecture, corner cases, and other tasks requiring domain expertise.

III. SYSTEMVERILOG ASSERTION GENERATION

This section first overviews SVA generation and the motivations of using LLMs. Then, it surveys recent LLM-based SVA generation methods at inference time and training time as summarized in Table II. Next, it reviews evaluation methods and benchmarks for SVA generation methods as summarized in Table III. Finally, remaining challenges and future directions are highlighted.

A. Overview and Motivation

Assertion-Based Verification (ABV) expresses expected hardware behavior as executable properties that can be evaluated through simulation or formally proven against an implementation. SystemVerilog Assertions (SVAs) are the dominant assertion language, used in approximately 80% of projects employing ABV [5]. SVAs capture functional and temporal requirements, such as protocol rules, request–response relationships, ordering constraints, and reset behavior, thereby providing an explicit representation of design intent. They improve observability and debugging and enable formal exploration of behaviors that may be difficult to exercise through simulation.

Despite these advantages, developing high-quality SVAs remains knowledge-intensive and time-consuming. Verification engineers must interpret requirements distributed across natural-language specifications, tables, diagrams, and timing descriptions; identify the behaviors that require verification; map abstract concepts to implementation signals; and encode them using precise temporal semantics. Ambiguous or incomplete specifications further widen this semantic gap between high-level design intent and executable assertions.

Accordingly, automated SVA generation must address more than syntactic correctness. A useful assertion must be *syntactically valid*, *semantically consistent with the intended requirement*, and *correctly aligned with implementation signals and cycle-level timing*. These requirements are distinct: an

TABLE II
REPRESENTATIVE RELATED WORK ON LLM-ASSISTED SVA GENERATION ACROSS TRAINING- AND INFERENCE-TIME TECHNIQUES

Work	Pre- or Post-RTL	Technique stage	Generation style	Feedback / evaluation	Main contribution	Key limitation
nl2sva [48]	Post-RTL	Inference	Circuit-aware few-shot prompting with editable sub-translations	JasperGold/model-checker feedback with human review	Early circuit-aware NL-to-SVA translation	Proof-of-concept evaluation; substantial human interaction
AssertLLM [23]	Pre-RTL	Inference	Multi-LLM task decomposition with structured text/waveform IRs	JasperGold FPV on I2C; syntax and proof pass rates	Full-specification multimodal generation using intermediate representations	Limited benchmark scope; no verifier-in-the-loop refinement
AutoSVA2 [49]	Post-RTL	Inference	GPT-4 RTL prompting with engineered syntax/semantic rules	FPV counterexamples and coverage on CVA6 PTW/TLB	Integrates direct RTL-to-SVA generation with AutoSVA	Requires engineer auditing/refinement and formal-tool access
AssertionForge [25]	Post-RTL	Inference	Spec+RTL knowledge graph with RAG/SSR and GRW-AS retrieval	JasperGold proof plus COI statement/branch/functional/toggle coverage	Structured spec-RTL grounding for broader assertion generation	KG overhead; no verifier-in-the-loop SVA refinement
ChiRAAG [50]	Post-RTL	Inference	Structured-spec prompting with iterative simulation-log repair	Simulation/test-case feedback on OpenTitan designs	Early automatic simulator-feedback refinement of generated SVAs	Few test cases; limited completeness/coverage guarantees
SANGAM [24]	Post-RTL	Inference	Multimodal agents with Monte Carlo Tree Self-Refine (MCTSr)	JasperGold proof/coverage feedback on I2C and RV-Timer	Search-based self-refinement for multimodal SVA generation	Two-design evaluation; compute/tool-intensive refinement
AssertCraft [51]	Post-RTL	Training + Inference	Subtask-focused fine-tuning plus iterative compiler/FPV-guided prompting	Custom compiler and FPV on HDLBits-derived designs	Combines task-specific fine-tuning with automated syntax/semantic repair	Requires curated training data and golden RTL; gains are coupled to refinement loop
AssertCoder [52]	Pre-RTL	Inference	Multimodal extraction plus semantic analyzers and multi-step CoT	Mutation-based model checking and refinement on three designs	Unified text/table/diagram/formula-to-SVA generation	Extraction errors propagate; evaluation limited to three designs
Spec2Assertion [53]	Pre-RTL	Inference	Progressive regularization with CoT and property decomposition	JasperGold FPV plus RTL-derived assertion-importance score	Explicit pre-RTL generation and assertion-quality metric	Evaluation metric still requires golden RTL
AssertGen [54]	Post-RTL	Inference	CoT objectives with cross-layer spec-to-RTL signal bridging	FPV, COI/proof-core, and mutation coverage on five designs	Generates assertions over internal cross-layer signal chains	Requires RTL/signal artifacts; no closed-loop refinement
ChatSVA [55]	Pre-RTL	Training + Inference	Four task-specific agents with AgentBridge data generation, SFT, and RAG	Formal verification on 24 designs; pass rate and coverage	Specialized trained agents for high-coverage end-to-end SVA generation	Heavy data/training pipeline; formal evaluation still depends on RTL oracle
SuperSAGA [56]	Post-RTL	Inference	Supervisor-subordinate agents with RAG and verification plan	Iterative formal-tool feedback; OpenTitan coverage evaluation	Agent hierarchy for generation, debug, and refinement	Requires human-reviewed verification plans and formal tools
SVAgent [57]	Post-RTL	Inference	Security-property decomposition with RTL/FSM analysis	Pyverilog/TrustHUB; syntax and functional accuracy	Security-focused SVA generation with structured reasoning chains	Limited to selected threat models/vulnerability classes
Saarthi [58]	Post-RTL	Inference	Multi-agent vPlan-to-SVA, proof/CEX repair, and coverage closure	Formal proof, counterexamples, coverage, and human escalation	End-to-end AI formal-verification engineer workflow	Human intervention remains for unresolved agent loops
ProofLoop [59]	Post-RTL	Inference	ReAct agent with AST-RAG and JasperGold structural queries	Three-round JasperGold proof refinement on FVEval Design2SVA	Adaptive RTL retrieval with solver-in-the-loop generation	Requires retrieval infrastructure and commercial FV tooling
KG-Based Agentic FV [60]	Post-RTL	Inference	Spec+RTL+feedback KG with syntax/CEX/coverage refinement loops	Formal diagnostics, CEX, and coverage on seven benchmarks	Persistent traceable grounding across the FV workflow	Complex KG/agent state; design-dependent convergence
Validatable Generation [61]	Pre-RTL	Training	Fine-tuned CodeT5/PLBART for NL-to-SVA with back-translation validation	Back-translation/cycle-consistency similarity; no RTL, formal check	Early supervised NL-to-SVA translation with an interpretable validation loop	Property-level input only; no full-specification extraction or RTL validation
VERT [62]	Post-RTL	Training	LoRA fine-tuning of LLaMA 3.1 and DeepSeek Coder on Verilog/SVA pairs	Syntax plus mutation, formal, and simulation-based functional checks	Large domain-specific SVA dataset showing strong gains from local fine-tuning	RTL-snippet task; limited evidence for full specification-driven generation
Hybrid-NL2SVA [63]	Pre-RTL	Training + Inference	Qwen2.5-Coder supervised fine-tuning combined with customized HybridRetrieval RAG	Formal evaluation on 40 designs and 229 annotated SVAs	Demonstrates complementary gains from SFT and retrieval for NL-to-SVA	Targets property-level translation rather than property extraction/full-spec generation

TABLE III
REPRESENTATIVE DATASETS AND BENCHMARKS FOR LLM-ASSISTED SVA GENERATION

Dataset / benchmark	Primary use	Scale	Inputs / targets	Construction / source	Evaluation / intended use	Key limitation
AssertLLM [23]	Evaluation	20 designs	Complete specifications + golden RTL; generated SVAs are evaluated against the design	Open-source IP and system-level designs with full natural-language specifications and verified RTL	Syntax, FPV pass/fail, and COI/proof coverage using formal verification	Small design set and assumes bug-free golden RTL; formal success is only a proxy for specification alignment
VERT [62]	Training	20,000 samples	RTL/code snippets $\rightarrow$ SVA	Synthetic if-else, case, and combined-logic snippets using variables extracted from open-source HDL projects; synchronous and asynchronous cases	Domain-specific fine-tuning of open-source LLMs for SVA generation	Synthetic snippet-level data does not represent complete specifications, modules, or hierarchical design intent
AssertionBench [64]	Evaluation + training	100 designs	Verilog design $\rightarrow$ assertion set	Curated OpenCores designs; assertions mined with GoldMine/HARM and formally verified with JasperGold	In-context learning and fine-tuning studies; syntax and FPV proof/counterexample outcomes	RTL-to-SVA setting lacks natural-language specification intent and inherits biases of mined reference assertions
FVEval [65]	Evaluation	571 cases	NL/testbench $\rightarrow$ SVA; synthetic NL $\rightarrow$ SVA; RTL $\rightarrow$ SVA	79 expert NL2SVA cases, 300 synthetic NL2SVA cases, and 192 generated RTL cases (96 pipelines + 96 FSMs)	Syntax and formal functional/equivalence checking across three assertion-generation settings	Human subset is small; two subsets are synthetic and do not model full specification-driven verification
Hybrid-NL2SVA [63]	Training + evaluation	4,070 training examples; 40 designs / 229 SVAs for evaluation	Natural-language property/explanation $\rightarrow$ SVA	Synthetic fine-tuning corpus with prompt-guided SVA construction explanations; separate formally verified annotated evaluation set	Fine-tuning/RAG evaluation using syntax and functional/formal matching	Focuses on property-level NL2SVA translation rather than extracting properties from complete specifications or RTL
CVDP-cid14 [66]	Evaluation	98 assertion tasks	Verification prompt/context $\rightarrow$ SVA	Human-authored tasks within the 783-problem CVDP suite; 68 non-agentic and 30 agentic assertion-generation problems	Executable test-harness/simulation-based pass evaluation in realistic single- and multi-step settings	Assertion generation is only one CVDP category and simulation/harness success does not establish formal or semantic correctness
AssertLLM2 [67]	Evaluation	83 designs, 13 categories	Structured specification + golden/buggy RTL $\rightarrow$ SVA	Real-world open-source designs (filtered for > 200 RTL lines), normalized specifications, dependency-complete RTL, and systematic mutations	Syntax, formal provability, COI/proof/formal coverage, and mutation-based bug detection in bug-prevention and bug-hunting settings	Still relatively small and untested among the broader LLM for SVA generation community

assertion may compile successfully yet express the wrong behavior, while an assertion formally proven against RTL may still misrepresent the specification. Signal and timing alignment is particularly challenging because specifications often describe behavior abstractly, whereas SVAs must identify concrete signals, sampling events, delays, and temporal relationships. Beyond individual assertions, practical generation should produce an assertion set that provides adequate specification coverage while avoiding vacuous, redundant, or inconsistent properties.

1) Automated SVA Generation Before LLMs: Before LLMs, automated assertion generation primarily relied on templates, rules, static analysis, and mining of simulation traces. Gold-Mine [68] mines candidate invariants from simulation traces, maps them to predefined assertion templates, and applies formal verification to eliminate invalid candidates. A-Team [69] combines templates with data mining and coverage-guided heuristics, while HARM [70] uses user-provided propositions, templates, and metrics to generate temporal properties.

These approaches provide predictable and structured generation, but their expressiveness is constrained by prede-

defined templates, propositions, or observed execution behavior. Consequently, they can struggle with previously unseen property structures, complex temporal relationships, and requirements expressed only at the specification level. Trace-mining methods may also infer properties that happen to hold for the observed implementation rather than properties representing intended behavior. More fundamentally, conventional approaches have limited capability to interpret unstructured specifications and determine *what properties should be verified*, leaving a semantic gap between design requirements and generated assertions.

2) *Motivation for LLM-Based SVA Generation*: LLMs offer a complementary capability by interpreting natural-language requirements, generating code, and reasoning across heterogeneous design information. They can potentially extract behavioral relationships from specifications, translate them into temporal properties, and, when RTL is available, map specification concepts to concrete signals and implementation structures. Retrieval and multimodal techniques can further incorporate information from large documents, tables, diagrams, and waveforms. These capabilities make LLMs particularly attractive for bridging the specification-to-assertion semantic gap.

However, LLM generation is probabilistic and provides no correctness guarantee. Generated assertions may contain syntax errors, incorrect temporal relationships, erroneous signal mappings, or properties that appear plausible but do not reflect design intent. Recent approaches therefore increasingly combine LLM generation with structured design representations, retrieval, simulation, formal verification, coverage analysis, and iterative refinement. In such workflows, LLMs provide semantic interpretation and candidate generation, while deterministic verification tools provide feedback for validation and refinement.

A key distinction is whether assertions are generated before or after RTL becomes available. *Pre-RTL* approaches derive properties primarily from specifications and therefore preserve greater independence from implementation errors, but face difficulties in mapping abstract requirements to concrete signals and cycle-level timing. *Post-RTL* approaches use implementation information to improve signal and timing grounding and produce more readily executable assertions, but risk generating properties that reproduce erroneous RTL behavior rather than independently checking the specification. Thus, effective LLM-based SVA generation must balance *specification fidelity* with *implementation grounding*, using RTL to resolve implementation details without allowing it to override the verification intent established by the specification.

B. Inference-Time Methods

Inference-time approaches generate SVAs using pretrained LLMs without modifying their model parameters. Compared with fine-tuning-based methods, they can readily incorporate new specifications, RTL designs, retrieved knowledge, and verification-tool feedback. Existing methods have evolved from direct natural-language translation toward structured,

multimodal, tool-grounded, and agentic workflows that address different stages of the specification-to-SVA transformation.

1) *Structured Decomposition and Design Grounding*: NL2SVA [48] is among the earliest LLM-based approaches and translates manually provided natural-language property descriptions into an intermediate representation containing signal information, partial translations, and candidate SVAs. Model checking is subsequently used to validate the assertions, with human intervention when refinement is required. Although this workflow demonstrates the feasibility of combining LLM generation with formal verification, it assumes that relevant requirements have already been extracted from the specification and therefore does not automate the more difficult task of identifying verification properties from complete design documents.

AssertLLM [23] extends this direction toward end-to-end generation from complete specifications. Rather than directly prompting an LLM with the entire document, it decomposes the task into specification analysis, signal mapping, and assertion generation, thereby producing structured intermediate descriptions before generating SVAs. It also incorporates assertion-related knowledge through retrieval and processes textual and waveform information. This decomposition reduces the reasoning burden of a single inference step and makes intermediate results more inspectable. However, because AssertLLM derives its design knowledge primarily from the specification, implementation details absent from the specification can remain unresolved.

AssertionForge [25] addresses this limitation by jointly grounding generation in the specification and RTL. It constructs a hardware-specific knowledge graph (KG) from specification concepts and fuses it with RTL entities and relationships, followed by multi-resolution context extraction for SVA generation. Compared with the primarily specification-centered decomposition of AssertLLM, this representation explicitly connects high-level requirements with implementation-level signals and dependencies, improving the context available for assertion generation. The tradeoff is increased pre-processing complexity and greater dependence on the RTL: errors in KG construction or an incorrect implementation can propagate into generated properties, potentially weakening their independence as specification-level checks.

AutoSVA2 [49] similarly introduces intermediate rules to constrain SVA generation. Collectively, these approaches show that intermediate representations—including structured descriptions, rules, and knowledge graphs—are useful for decomposing the large semantic jump from heterogeneous specifications to temporal logic. However, additional representations also create new points at which information can be omitted or incorrectly transformed.

2) *Multimodal Processing and Explicit Reasoning*: Hardware specifications frequently encode critical behavior in tables, timing diagrams, waveforms, equations, and figures rather than text alone. AssertLLM [23] incorporates waveform information, while AssertCoder [52] more explicitly decomposes multimodal specification processing by converting

diagrams, waveforms, tables, and formulas into structured representations such as pseudocode, formal expressions, $\text{\LaTeX}$, or JSON before assertion generation. SANGAM [24] likewise employs specialized Spec Analyzer, Signal Mapper, and Waveform Analyzer components to extract complementary information from specifications. These approaches reduce dependence on textual descriptions and can recover timing or structural information unavailable in prose. Their effectiveness, however, depends on the accuracy of multimodal interpretation; an incorrectly interpreted waveform or table can introduce semantic errors before SVA generation begins.

Explicit reasoning provides another mechanism for bridging specifications and temporal properties. Spec2Assertion [53] applies Chain-of-Thought (CoT) prompting and progressive regularization in the pre-RTL setting, encouraging the model to reason about antecedents, consequents, temporal relationships, and cycle dependencies before constructing an SVA. AssertCoder [52] extends multi-step reasoning to structured multimodal inputs, whereas AssertGen [54] first extracts verification objectives from the specification and subsequently maps them to RTL signals. The latter approaches therefore combine explicit reasoning with implementation grounding, addressing a major limitation of purely pre-RTL reasoning. Nevertheless, a plausible reasoning trace is not evidence of semantic correctness: incorrect timing assumptions or incomplete interpretation of the requirement can still produce well-formed but incorrect assertions.

3) *Retrieval and Verification-Guided Refinement*: Retrieval-Augmented Generation (RAG) provides external knowledge without modifying the LLM. AssertLLM [23], for example, retrieves SVA-related verification knowledge to improve generation, whereas AssertionForge [25] uses graph-based retrieval over design-specific relationships. The distinction is important: general verification knowledge primarily helps the model understand *how to express* assertions, while design-specific retrieval helps determine *what behavior should be expressed*. The latter is therefore more directly relevant to specification-to-RTL semantic alignment.

Most recent frameworks further couple generation with iterative feedback. NL2SVA [48] introduced model-checking feedback with human-guided refinement, while general self-refinement methods such as Self-Refine [71] demonstrated that LLMs can critique and revise their own outputs. ChIRAG [50] applies verification-tool feedback to SVA correction by interpreting errors and regenerating failed assertions. Such feedback is particularly effective for syntax and tool-detectable errors because compilers and formal engines provide objective diagnostics. Its ability to correct semantic errors is more limited: a property that successfully compiles or is proven against the RTL may still encode the wrong requirement.

SANGAM [24] extends linear self-refinement into candidate search through Monte Carlo Tree Self-Refine (MCTS_r). After multimodal specification processing, it explores multiple reasoning trajectories for individual signals and uses the resulting traces to generate assertions. Compared with one-shot or linear

refinement, this approach reduces dependence on a single stochastic generation and can explore alternatives involving different signal dependencies or temporal relationships. However, its effectiveness depends critically on the reward or selection criterion; optimizing primarily for formal-tool success can favor assertions that are provable but weak, vacuous, or semantically misaligned. SANGAM therefore represents a shift from simply *repairing* a generated assertion toward *searching* the space of candidate interpretations.

AssertCoder [52] and AssertGen [54] further incorporate mutation testing to evaluate whether generated assertions detect injected RTL faults. This provides stronger evidence of verification usefulness than syntax or proof success alone because it evaluates fault-detection capability. Nevertheless, mutation score remains dependent on the representativeness of the injected faults and does not by itself establish that an assertion faithfully represents the specification.

4) *Agentic SVA Generation*: The progression from task decomposition and iterative refinement naturally leads to agentic workflows, in which specialized components coordinate specification analysis, retrieval, assertion generation, tool execution, and repair. AssertLLM [23] and SANGAM [24] already exhibit elements of role-based decomposition, although their primary contributions are structured generation and search rather than autonomous tool orchestration.

More explicit agentic approaches extend this decomposition. ChatSVA [55] formulates SVA generation as an end-to-end multi-agent workflow with specialized stages for specification processing, functional-checkpoint extraction, and assertion generation, while its AgentBridge infrastructure supports construction of task-specific data. SuperSAGA [56] uses a supervisor-subordinate architecture combining RAG, human-reviewed verification plans, formal-tool feedback, and iterative assertion refinement. SVAgent [57] applies agentic decomposition specifically to hardware-security verification by breaking complex security requirements into finer-grained verification tasks before SVA generation.

This direction also overlaps increasingly with broader agentic formal-verification systems. Saarthi [58], ProofLoop [59], and KG-based agentic formal-verification frameworks [60] coordinate assertion generation with mechanisms such as verification rulebooks, knowledge-graph retrieval, formal-tool queries, counterexample-guided refinement, and coverage feedback. These systems expand the role of the LLM from an assertion generator to an orchestrator of the verification process. However, greater autonomy also increases pipeline complexity, inference and tool-execution cost, and opportunities for error propagation. Consequently, agentic decomposition is most useful when intermediate artifacts such as extracted requirements, signal mappings, verification plans, and candidate assertions can themselves be independently validated rather than trusting only the final output.

5) *Summary and Remaining Limitations*: Overall, inference-time SVA generation has evolved along several complementary dimensions. Early methods such as NL2SVA focus on translating explicitly provided requirements,

whereas AssertLLM expands generation to complete specifications through task decomposition. AssertionForge and AssertGen strengthen specification-to-RTL grounding; AssertCoder and SANGAM incorporate multimodal information; Spec2Assertion emphasizes explicit reasoning; SANGAM further introduces search over alternative reasoning trajectories; and recent agentic frameworks coordinate these capabilities with external verification tools. Thus, the field is moving from direct *NL-to-SVA translation* toward *specification understanding, design grounding, candidate generation, verification, and iterative refinement*.

Despite this progression, three limitations remain common. First, stronger RTL grounding improves signal mapping and executability but can compromise independence from implementation errors, whereas specification-only methods preserve verification intent but struggle with implementation-level details. Second, tool feedback provides reliable evidence for syntax, satisfiability, and behavior relative to the RTL, but does not by itself establish semantic agreement with the specification. Third, increasingly elaborate reasoning, search, and multi-agent pipelines improve robustness at the expense of latency, tool calls, computational cost, and additional opportunities for intermediate errors. Future inference-time methods therefore need to combine specification fidelity and implementation grounding while independently validating intermediate representations and evaluating generated assertion sets using complementary measures of semantic correctness, coverage, non-vacuity, and realistic fault-detection capability.

C. Training-Time Methods

Training-time approaches adapt an LLM to SVA generation by updating its parameters using hardware-verification-specific data. Unlike inference-time methods, which rely on prompting, retrieval, structured context, search, or tool feedback without modifying the underlying model, fine-tuning attempts to internalize SVA syntax, temporal semantics, and recurring hardware-property patterns. This can improve generation consistency and enable smaller open-source models to perform specialized verification tasks. However, because SVA generation involves multiple stages—including requirement extraction, signal mapping, temporal reasoning, and assertion construction—most existing training-time methods focus on particular subtasks rather than learning the complete specification-to-SVA process.

1) *Property-Level Translation and Semantic Adaptation:* Aditi and Hsiao [61] provide an early demonstration of fine-tuning for SVA generation by adapting CodeT5 and PLBart to translate natural-language property descriptions into assertions. The generated SVA is additionally translated back into natural language and compared with the original description, providing an interpretable consistency check between the input requirement and generated property. This work demonstrates that task-specific training can teach models the correspondence between natural-language temporal behavior and SVA constructs. However, it assumes that the desired property has already been identified and expressed as a concise natural-

language statement. It therefore addresses *property translation* rather than the broader problem of extracting verification requirements from complete specifications. Moreover, back-translation measures linguistic consistency rather than behavioral correctness; an assertion can appear consistent in natural language while still containing incorrect temporal semantics or failing to represent the RTL behavior correctly.

Hybrid-NL2SVA follows a similar property-translation formulation but combines domain-specific fine-tuning with inference-time augmentation. It fine-tunes Qwen2.5-Coder-7B-Instruct using a synthetic dataset constructed from hardware-design textbooks and Verilog collected from 67 designs, containing 4,070 SVAs paired with one-sentence natural-language descriptions generated using OpenAI o4-mini. The resulting model shows substantial improvement after fine-tuning, while retrieval-based augmentation further improves generation. Compared with the earlier CodeT5/PLBart approach, Hybrid-NL2SVA benefits from a stronger pretrained code model, a larger hardware-specific dataset, and a hybrid training/inference strategy. Nevertheless, both approaches share the same fundamental abstraction: they assume that a relevant property has already been extracted and expressed in natural language. Their results therefore demonstrate improved *NL-to-SVA translation*, but provide limited evidence for full specification-driven assertion generation. In addition, synthetic natural-language labels and automatically collected assertions make performance dependent on the semantic quality of the constructed training pairs.

2) *Specification-Conditioned and Hybrid Fine-Tuning:* AssertCraft [51] moves closer to specification-driven generation by fine-tuning an LLM using assertion-related examples extracted and filtered from open-source repositories together with synthetic data. Rather than relying exclusively on the fine-tuned model, AssertCraft combines task-specific adaptation with an inference-time iterative refinement loop in which candidate assertions can be repaired using a non-fine-tuned LLM. This hybrid architecture is significant because it treats fine-tuning and inference-time reasoning as complementary: fine-tuning provides domain-specific generation capability, while iterative refinement addresses errors that remain after generation.

This design also illustrates an important difficulty in evaluating training-time approaches. Improvements in AssertCraft arise from several interacting components, including dataset construction, task decomposition, the fine-tuned generator, and iterative refinement. Consequently, end-to-end improvements cannot necessarily be attributed to fine-tuning alone without controlled ablation. More generally, fine-tuning can improve the model’s prior knowledge of SVA patterns but does not eliminate the need for specification grounding or external verification. A model trained on high-quality assertion examples may still generate a property that is inappropriate for the particular design being verified.

3) *Domain-Specific Model Adaptation:* VERT [62] further emphasizes the value of domain-specific training data. It constructs a large SVA dataset from open-source hardware designs

and uses it to fine-tune DeepSeek Coder and LLaMA 3.1 models, which are subsequently compared with GPT-4o across multiple hardware designs. The results show that smaller open-source models adapted to hardware-verification data can outperform a substantially larger general-purpose closed-source model on the evaluated SVA-generation tasks. This provides important evidence that, for specialized verification tasks, domain adaptation can compensate for model scale. Fine-tuned open-source models also enable local deployment, which is attractive when proprietary RTL and specifications cannot be exposed to external model providers.

VERT, however, also exposes a recurring limitation of current training-time methods: the granularity and provenance of training data strongly determine what the model learns. Assertions mined from open-source repositories provide realistic SVA syntax and recurring temporal patterns, but snippet-level examples contain limited information about the complete specification and design context from which the assertions originated. Consequently, strong performance on local assertion-generation tasks does not necessarily imply that the model can identify appropriate verification objectives from a full specification, reason across module hierarchies, or construct a comprehensive assertion set for an unseen IP.

4) *Cross-Method Analysis and Limitations*: Taken together, these studies demonstrate a progression from fine-tuning for direct NL-to-SVA translation toward domain-adapted models embedded within larger verification pipelines. Aditi and Hsiao [61] establish the feasibility of learning mappings between natural-language properties and SVA; Hybrid-NL2SVA scales this formulation using a modern code LLM, synthetic data, and retrieval; AssertCraft [51] combines fine-tuning with iterative inference-time refinement; and VERT [62] demonstrates that sufficiently specialized training can make smaller open-source models competitive with or superior to larger general-purpose models. Thus, training-time and inference-time techniques are increasingly complementary rather than competing alternatives: fine-tuning internalizes recurring domain knowledge, while retrieval, structured context, RTL grounding, and verification-tool feedback provide design-specific information that cannot reasonably be encoded in model parameters.

Nevertheless, current training-time methods remain primarily focused on learning *how to express properties as SVAs*, rather than determining *which properties are required to adequately verify a design*. Most training datasets consist of assertion snippets or pairs of natural-language descriptions and SVAs, with relatively limited representation of complete specifications, RTL hierarchy, verification intent, and relationships among multiple properties. Dataset construction also introduces risks from noisy repository assertions, automatically generated descriptions, duplicated examples, and possible overlap between training and evaluation designs. These issues make data quality, provenance, and contamination control particularly important when evaluating fine-tuned models.

Future training-time methods therefore require datasets that preserve richer relationships among *specifications, RTL, verifi-*

cation objectives, assertions, and verification outcomes. Training objectives could similarly move beyond next-token prediction on individual assertions toward learning specification-to-property alignment, signal grounding, temporal reasoning, and assertion-set construction. Importantly, fine-tuning should not replace deterministic verification: compilation, simulation, formal analysis, vacuity checking, and mutation testing remain necessary for independently evaluating generated properties. A promising direction is therefore a hybrid architecture in which domain-specific training provides efficient hardware-verification knowledge, while inference-time retrieval and reasoning supply design-specific context and formal tools provide objective feedback on correctness and verification effectiveness.

D. Evaluation Methods and Benchmarks

The inference- and training-time methods reviewed above employ different mechanisms to improve SVA generation, ranging from structured prompting, retrieval, reasoning, and formal-tool feedback to domain-specific fine-tuning. Evaluating these methods, however, requires determining not only whether an LLM can produce executable SVA code, but also whether the generated properties faithfully capture verification intent and effectively detect incorrect behavior. Existing evaluations can therefore be viewed as providing increasingly strong but complementary forms of evidence: *syntactic validity, formal consistency, coverage, fault-detection capability*, and ultimately *specification alignment*.

1) *Evaluation Methodologies*: **Syntactic validity** is the most basic and widely reported metric. It measures whether generated assertions can be parsed, compiled, and accepted by SystemVerilog verification tools. This metric is inexpensive, deterministic, and particularly useful for measuring whether prompting or fine-tuning improves knowledge of SVA syntax. However, compilation establishes only that an assertion is executable; a syntactically correct property may still check the wrong behavior, use inappropriate timing, or encode an incorrect requirement.

Formal property verification (FPV) provides stronger behavioral evidence by checking generated assertions against an RTL implementation using formal verification tools such as JasperGold or SymbiYosys. If a trusted golden RTL implementation is assumed to correctly realize the specification, successful proof provides evidence that the generated assertion is consistent with the expected design behavior, while counterexamples can identify incorrect properties. FPV is therefore widely used in benchmarks such as AssertLLM [23], VERT [62], AssertionBench [64], and FVEval [65]. However, FPV is not a direct semantic oracle. A weak, vacuous, or incorrectly interpreted assertion may still be provable, and results depend on the RTL, assumptions, constraints, and formal environment. Proofs may also time out or remain inconclusive for complex designs. Thus, FPV primarily establishes *consistency with the analyzed implementation*, not necessarily correctness with respect to the original specification.

Coverage analysis complements FPV by measuring how generated properties relate to the implementation. Existing work reports metrics such as cone of influence (COI), proof coverage, formal coverage, and, depending on the environment, statement, branch, or toggle coverage. AssertLLM [23] introduced COI-based analysis alongside FPV, while later benchmarks such as VERT [62] and AssertLLM2 [67] employ broader coverage measurements. Coverage can expose trivial assertions that constrain or observe little of the design and can assess the collective reach of an assertion set. Nevertheless, high coverage does not imply correct verification intent: a property may involve many RTL signals while still encode behavior inconsistent with the specification.

Mutation testing evaluates a different dimension—whether assertions can detect erroneous implementations. Mutations are introduced into an otherwise correct RTL design, and generated assertions are evaluated according to whether they fail on the faulty versions. AssertCoder [52], VERT [62], and AssertLLM2 [67] use mutation-based evaluation to complement syntax, proof, and coverage metrics. Compared with proving assertions only against golden RTL, mutation testing provides stronger evidence of practical fault-detection capability and assertion strength. However, its validity depends strongly on the realism, diversity, and relevance of the injected faults. Easily detected or artificial mutations may inflate performance, while generating representative semantic bugs at scale remains difficult. Mutation score should therefore be interpreted as a measure of *fault-detection effectiveness*, rather than direct proof of specification correctness.

These metrics evaluate complementary properties rather than interchangeable notions of correctness. Syntax checking establishes *executability*; FPV establishes *consistency with RTL under the verification environment*; coverage measures *interaction with or reach into the design*; and mutation testing measures *ability to detect selected faults*. The strongest objective, **specification alignment**, asks whether an assertion accurately captures the intended requirement. This remains the most difficult dimension to evaluate automatically because neither the RTL nor generated mutations provide an independent semantic oracle when the objective is to verify the implementation against a natural-language specification. Consequently, no single existing metric is sufficient, motivating evaluation frameworks that combine multiple complementary measures.

2) *Benchmarks*: Existing SVA benchmarks differ substantially in task formulation, design scale, input context, and evaluation methodology, reflecting the evolution from property translation toward specification- and design-level assertion generation.

AssertLLM [23] introduced one of the earliest standardized benchmarks for specification-driven SVA generation. It contains 20 samples drawn from open-source IP and system-level designs, pairing specifications with corresponding RTL. This pairing is important because it enables evaluation of both specification-conditioned generation and generated assertions against executable implementations. Its evaluation combines syntax checking, COI analysis, and FPV, providing stronger

evidence than syntax-only evaluation. However, the small number of designs limits behavioral and structural diversity and makes broad conclusions about generalization difficult.

VERT [62] takes a substantially different approach. Its dataset contains approximately 20,000 snippets extracted from open-source hardware repositories, including BOOM, Rocket Chip, XiangShan, and other RTL projects. Rather than emphasizing complete specification-to-design tasks, VERT targets recurring SVA-generation patterns and errors, including temporal and clock-cycle interpretation. Its scale makes it particularly suitable for training and evaluating domain-adapted models, and its combination of syntax, COI, FPV, and mutation testing provides multiple measures of assertion quality. The tradeoff is granularity: snippet-level tasks provide limited evidence that a model can identify verification objectives from a complete specification or reason across large hierarchical designs. Thus, VERT is especially valuable for evaluating the *training-time domain adaptation* discussed in the previous subsection, whereas specification-level benchmarks better exercise the inference-time reasoning and grounding pipelines reviewed earlier.

AssertionBench [64] occupies an intermediate position. It contains 100 samples derived from OpenCores designs, with formally proven reference assertions, English descriptions, and corresponding Verilog designs. This construction provides controlled NL-to-SVA tasks while retaining executable RTL for formal validation, making it useful for evaluating in-context learning and property translation. However, its emphasis on syntax and FPV provides limited measurement of assertion-set coverage, fault-detection capability, or fidelity to complete design specifications.

FVEval [65] provides broader task diversity through three complementary sub-benchmarks. NL2SVA-Human contains 79 expert-written assertions from real-world verification environments paired with natural-language descriptions; NL2SVA-Machine contains 300 machine-generated natural-language/SVA pairs; and Design2SVA contains 192 synthetic RTL instances based on arithmetic pipelines and finite-state machines. This decomposition is useful because it distinguishes human-authored property translation, scalable synthetic translation, and direct RTL-to-SVA generation rather than collapsing them into a single task. It therefore enables analysis of how performance changes with the source and form of the input. However, its primary reliance on syntax checking and FPV again leaves specification coverage, realistic fault detection, and full-document semantic alignment largely unmeasured.

CVDP [66] differs from these assertion-specific benchmarks by evaluating SVA generation as one component of a broader suite of LLM-based EDA tasks. Its assertion subset contains 98 tasks with prompts, relevant context, design files, and test harnesses, using checklist-based evaluation and syntax checking. This task-oriented formulation is valuable for assessing SVA generation within a broader hardware-engineering context rather than an isolated translation setting. However, its evaluation provides less evidence about formal correct-

ness, non-vacuity, coverage, and fault-detection strength than assertion-specific benchmarks.

AssertLLM2 [67] moves toward a more comprehensive design-level evaluation. It contains 83 samples collected from open-source Verilog/SystemVerilog repositories together with corresponding specifications, while filtering out designs below 200 lines of RTL to increase task complexity and normalizing specification structure. More importantly, its evaluation combines syntax checking, FPV, COI, proof coverage, formal coverage, and mutation testing. Compared with earlier benchmarks that emphasize one or two dimensions, AssertLLM2 therefore evaluates both whether assertions are executable and provable and whether they exercise meaningful portions of the design and detect injected faults. Its remaining limitation is shared by the broader literature: these tool-based metrics remain proxies for whether generated assertions faithfully and comprehensively represent the original specification.

3) *Cross-Benchmark Analysis and Open Challenges:* Taken together, these benchmarks reveal a progression in both *task complexity* and *evaluation strength*. Property-level benchmarks such as AssertionBench and portions of FVEval provide controlled evaluation of NL-to-SVA translation, while VERT emphasizes scale and is particularly suited to training-time domain adaptation. AssertLLM and AssertLLM2 move toward complete specification- and design-conditioned generation, better exercising the reasoning, retrieval, signal-grounding, and refinement capabilities of inference-time systems. CVDP instead evaluates assertion generation within a broader EDA task environment. Similarly, evaluation has progressed from syntax checking toward FPV, coverage, and mutation testing, with recent benchmarks increasingly combining these metrics.

However, direct comparison across reported results remains difficult because benchmarks differ in design source, scale, input information, property granularity, formal environment, and evaluation criteria. Training-time evaluations introduce the additional concern of contamination or repository overlap, particularly when both training and evaluation examples originate from public RTL repositories. Inference-time methods face a related concern when specifications, RTL, or reference assertions may have appeared in model pretraining data. Evaluation on held-out designs and, where possible, previously unpublished or carefully contamination-controlled benchmarks is therefore important for establishing generalization.

More fundamentally, existing benchmarks still provide limited measurement of *assertion-set quality*. Evaluating assertions independently does not determine whether the generated set adequately covers the specification, contains redundant or contradictory properties, or omits critical requirements. Future benchmarks should therefore preserve explicit relationships among specifications, RTL, reference verification objectives, assertions, and realistic faults, enabling separate measurement of syntactic validity, semantic fidelity, specification coverage, non-vacuity, consistency, behavioral diversity, and fault-detection capability. For iterative and agentic methods, evaluation should additionally report inference cost, verification-tool runtime, number of refinement iterations or tool calls,

robustness across repeated runs, and human intervention. Such multidimensional evaluation is necessary to compare increasingly sophisticated inference- and training-time approaches and to determine whether improvements in SVA generation translate into more effective hardware verification.

E. Challenges and Future Directions

Despite substantial progress from direct NL-to-SVA translation toward structured, tool-guided, fine-tuned, and agentic approaches, several challenges remain before LLM-generated SVAs can be reliably used in practical verification.

Semantic alignment and evaluation. A fundamental challenge is determining whether generated assertions faithfully represent the specification. Existing methods commonly rely on syntax checking, formal property verification (FPV), coverage, and mutation testing. These provide complementary evidence of executability, consistency with RTL, design interaction, and fault-detection capability, respectively, but none directly establishes specification fidelity. SpecAlign [72] represents an initial step toward explicitly evaluating the semantic alignment between generated SVAs and their corresponding specifications. Nevertheless, specification-level evaluation remains challenging, particularly when golden RTL is used for both generation and validation, since an incorrect property derived from faulty RTL may still be formally proven against that RTL. Future evaluation should therefore combine complementary metrics with independent specification-level assessment. Standardized benchmarks should also include diverse unseen designs, realistic specifications and faults, and contamination controls to enable meaningful comparison across methods.

From individual assertions to assertion sets. Existing evaluations primarily assess individual properties, whereas practical verification depends on whether the complete assertion set adequately captures design intent. Generating many simple and correct assertions may produce favorable aggregate results while missing complex behaviors or interactions. Future evaluation should therefore consider specification coverage, non-vacuity, redundancy, consistency, behavioral diversity, and fault-detection effectiveness at the assertion-set level. Explicit traceability between specification requirements and generated properties would further reveal missing or inadequately represented requirements.

Generalization, scalability, and cost. Training-time approaches show the benefits of domain adaptation, while inference-time methods improve design-specific reasoning through retrieval, structured representations, and tool feedback. However, both remain sensitive to data quality and are largely evaluated on relatively small designs or localized properties. Scaling to complex SoCs requires reasoning across hierarchical RTL and specifications while selectively retrieving relevant information across modules and abstraction levels. Moreover, multimodal processing, candidate search, agentic workflows, and repeated formal-tool calls increase latency and computational cost. Future evaluations should therefore report inference and verification runtime, tool calls, refinement iterations, and human intervention, together with controlled

ablations that isolate the contributions of individual techniques.

Toward complete verification property sets. Another promising direction is joint generation of *assumptions, assertions, and cover properties*. Assumptions characterize valid environmental behavior, assertions specify required design behavior, and cover properties establish reachability of important scenarios. Generating and validating them together can reduce spurious failures, expose overconstraint and vacuity, and provide a more complete representation of verification intent than assertions alone.

Overall, future research should shift from optimizing the generation of individually valid assertions toward constructing *specification-aligned and independently validated verification property sets*. LLMs are well suited for interpreting requirements, connecting information across heterogeneous artifacts, and proposing candidate properties, while deterministic simulation and formal tools provide essential validation and feedback. Combining these complementary capabilities—while maintaining traceability to the specification, controlling cost, and preserving independence from implementation errors—is a promising path toward reducing manual assertion-development effort and enabling reliable, review-centered assertion-based verification.

IV. TESTBENCH GENERATION

This section surveys recent work on automatic testbench and stimulus generation for hardware verification, with emphasis on large language model (LLM) based methods. It situates LLM approaches relative to pre-LLM template and grammar-based tools, then reviews prompt-only generation, EDA-feedback loops, coverage-directed stimulus generation, self-validating testbench construction, UVM/subsystem-level generation, training-time alignment, and benchmark design. The main conclusion is that current LLMs are most useful when embedded in verification-aware workflows that constrain generation with scenarios, templates, simulators, coverage reports, and functional self-checks, rather than when used as unconstrained SystemVerilog authors.

A. Motivation and Scope

Simulation-based functional verification remains one of the most common pre-silicon verification techniques. A useful testbench is not merely code that compiles: it instantiates the design under test (DUT), drives meaningful stimuli, observes outputs, compares behavior against an oracle or reference model, and records structural or functional coverage. In industrial SystemVerilog flows, this often expands into Universal Verification Methodology (UVM) components such as sequence items, sequencers, drivers, monitors, agents, scoreboards, coverage collectors, and tests [73, 74]. Python-based harnesses such as cocotb are also relevant because many LLM workflows exploit Python’s stronger representation in pretraining corpora to implement checkers, reference models, or orchestration code [75].

The literature on LLM-assisted testbench generation separates into three related but distinct tasks. The first is *stimulus generation*: producing input vectors, transactions, or sequences that improve code or functional coverage. The second is *testbench construction*: producing executable drivers, clocks, resets, monitors, and checkers. The third is *verification-environment synthesis*: generating reusable UVM or subsystem-level verification infrastructure. These tasks differ in difficulty. Stimulus generation can be judged by coverage and simulator output, while full testbench generation also requires correct timing, reset handling, protocol compliance, and a faithful oracle. This distinction explains why many recent systems use the LLM as a planner or repair agent, while deterministic code templates, DSLs, or simulators enforce the final executable structure [76, 77, 78, 79].

Figure 2 summarizes this taxonomy and the feedback signals that cut across the three task levels.

B. Pre-LLM Automation: Templates, Grammars, and Constrained Random Testing

Automatic testbench generation predates modern LLMs. Earlier tools demonstrate two lessons that remain important. First, constraining the output space greatly improves syntactic stability. Second, simulation throughput matters because verification requires repeated execution. AVERT, for example, generates Verilog testbenches for grammatical-evolution hardware-design workflows by using DUT port information and standard test-vector files; it supports combinational and sequential circuits and improves simulation efficiency by evaluating multiple DUTs in a single generated testbench [80]. Such approaches do not solve oracle discovery or semantic adequacy, but they show that structured generation can remove repetitive formatting work and reduce avoidable syntax errors.

Other pre-LLM work includes coverage-directed test generation, directed testing, and RTL fuzzing. Machine-learning-assisted coverage-directed generation and directed hardware validation emphasizes the same optimization problem that LLM systems now revisit: selecting tests that close coverage efficiently while preserving bug-finding power [81, 82]. RFUZZ, DirectFuzz, and FuzzWiz show that fuzzing-style feedback can drive RTL exploration using coverage or hardware-specific guidance, making them important non-LLM baselines for stimulus-generation studies [83, 84, 85].

Traditional constrained-random verification similarly provides a useful baseline for LLM work. Randomization and coverage feedback can explore large input spaces, especially in UVM-style environments, but engineers still have to write constraints, monitors, scoreboards, and coverage models. LLM-based methods therefore should not be evaluated only against direct prompting; they should also be compared with random, constrained-random, template-based, fuzzing-based, and coverage-directed baselines where appropriate. This is especially important because a generated testbench that improves line coverage may still miss a functional bug if the checker is weak or absent.

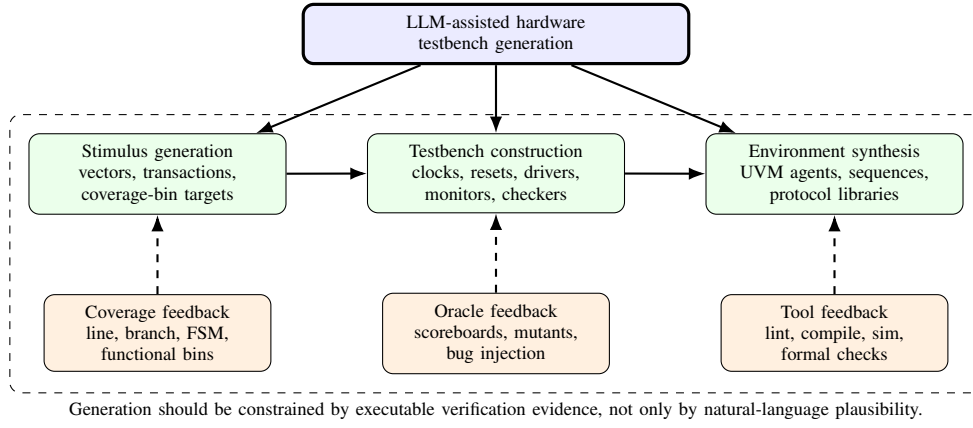


Fig. 2. Taxonomy of LLM-assisted testbench generation tasks and the main verification signals used to constrain them.

C. Prompted LLM Generation with EDA Feedback

The earliest LLM-based testbench studies generally prompt a commercial model with RTL and a natural-language specification, compile or simulate the result, and feed tool output back into subsequent prompts. Bhandari et al. study this pattern for finite-state-machine testbench generation and bug detection. Their workflow asks GPT-3.5 or GPT-4 to generate tests, uses commercial EDA feedback to identify uncovered transitions or errors, and iteratively refines prompts until coverage improves. The work is important because it treats simulator and coverage output as first-class prompt context, not merely as an after-the-fact evaluation step [86].

The FSM study also exposes a recurring limitation: LLMs struggle with long traces, many states, and multi-bit outputs. The authors mitigate this by chunking traces, resetting after groups of transitions, and decomposing multi-bit outputs into smaller subproblems. This suggests a broader design principle for testbench generation: scalable systems should avoid asking the model to reason over an entire trace or full environment in one context window. Instead, they should decompose the task into local coverage gaps, individual protocol obligations, or smaller signal cones.

D. Coverage-Directed Stimulus Generation

A second line of work uses the LLM primarily as a coverage-directed stimulus generator. LLM4DV is an open-source framework that orchestrates LLMs for hardware test-stimuli generation across multiple DUTs. Rather than asking for a complete verification environment, it iteratively samples missed coverage bins, restarts or refines dialogue context, and uses few-shot prompting to guide the generation of input stimuli [87]. This narrower framing is valuable because coverage bins and simulator feedback provide an executable objective: the model is rewarded for reaching untested behavior, not for producing aesthetically plausible HDL.

VerilogReader takes a related but more code-coverage-oriented view. It integrates an LLM into a coverage-directed generation loop where the model acts as a “Verilog reader” that interprets RTL and coverage reports, then proposes stimuli

for uncovered branches [22]. The system’s Coverage Explainer and DUT Explainer modules illustrate a practical prompt-engineering pattern: raw Verilator output is not ideal LLM context, so an intermediate explanation layer translates simulator artifacts into natural-language goals. The method performs best when the design remains within the model’s comprehension scope, and its degradation on larger FSMs reinforces the scalability concerns observed in FSM-focused work.

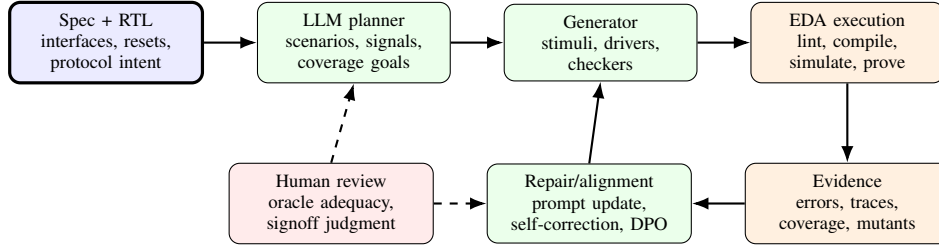
Spec2Cov generalizes this agentic coverage-closure perspective by driving test-stimulus generation directly from design specifications and iteratively coordinating compilation, simulation, coverage-report parsing, and prompt refinement [88]. Its results show that agentic coverage closure can reach full coverage on simpler designs but remains significantly harder on complex designs, which is consistent with the broader literature: coverage feedback is powerful, but it does not eliminate the need for design understanding, good oracles, and scalable decomposition.

Several newer systems frame coverage closure as part of a broader verification-agent loop. PRO-V-R1 combines an LLM with reinforcement learning and simulation feedback to improve automatic RTL verification, while UCAGENT proposes an end-to-end block-level verification agent that traverses states and goals toward coverage closure [89, 90]. These systems are not only testbench generators; they combine planning, test generation, execution, and refinement. They are therefore best viewed as adjacent evidence that testbench generation is becoming an agentic workflow rather than a single prompt-completion task.

Figure 3 abstracts the recurring loop shared by these systems.

E. Full Testbench Generation and Self-Correction

AutoBench is a key transition from stimulus-only work to complete LLM-generated testbenches. It proposes a staged workflow that separates scenario specification, Verilog driver generation, and Python checker generation. This architecture deliberately uses Python for checking because LLMs are



Solid arrows: automated tool-feedback loop. Dashed arrows: human constraints and review.

Fig. 3. Verification-aware feedback loop used by simulator-in-the-loop, self-correcting, coverage-directed, and agentic testbench-generation systems.

generally stronger at Python than HDL, while Verilog remains necessary for driving and integrating with RTL simulation [76]. AutoBench also introduces AutoEval, which evaluates generated testbenches at multiple levels: basic execution, functional behavior, and mutation-based adequacy. The mutation-based step is particularly important because a testbench can compile and even pass the golden design while still failing to reject incorrect RTL mutants.

CorrectBench extends this direction by adding functional self-validation and self-correction. Its central insight is that syntax-level debugging is insufficient: the generated checker or driver may be functionally wrong even after it compiles. CorrectBench uses scenario-based validation with imperfect RTL variants to infer whether a generated testbench can distinguish correct and incorrect behavior, then feeds detected bug information to an LLM-based corrector [77]. This shifts the target from “does the testbench run?” to “does the testbench detect meaningful behavioral differences?”

ConfiBench further extends the AutoBench/CorrectBench family with confidence-based scenario masking and a multi-testbench ensemble. Instead of trusting a single generated testbench in all scenarios, it masks high-risk scenarios and combines multiple generated testbenches to compensate for coverage loss. Reported results show a higher overall pass rate than AutoBench and CorrectBench-style baselines, especially on sequential circuits [91]. The broader lesson is that ensemble and confidence mechanisms can be useful when individual LLM outputs remain unstable.

F. UVM and Subsystem-Level Generation

Moving from module-level Verilog testbenches to UVM environments introduces a substantial jump in complexity. UVM code is class-heavy, convention-driven, and protocol-sensitive; a generated environment must coordinate transaction definitions, sequences, drivers, monitors, scoreboards, configuration databases, coverage groups, and simulator-specific build rules. VerifLLMBench addresses this evaluation gap by benchmarking LLM-generated UVM testbenches using verified DUTs, generated UVM components, iterative syntax repair, linting, compilation, simulation, and coverage analysis [92]. Its importance is methodological: UVM generation must be evaluated as an environment-construction problem, not merely as a single-file code-generation task.

UVM² demonstrates a more automated UVM workflow that generates UVM testbenches and refines stimuli with coverage feedback on RTL designs up to roughly 1.6K lines of code [93]. HAVEN pushes the structured-generation principle further by preventing the LLM from writing raw UVM code directly. Instead, LLM agents extract an architectural plan, protocol-specific Jinja2 templates instantiate UVM components, and a protocol-aware sequence DSL generates bus-correct stimulus. On open-source IPs spanning Direct, Wishbone, and AXI4-Lite interfaces, HAVEN reports complete compilation success and strong average code and functional coverage [78]. This is one of the clearest examples that reducing LLM freedom can improve verification trust.

UVMarvel extends the UVM story to subsystem-level verification. It introduces an intermediate representation, a bus protocol library, signal tracking, and a Verilog patching library to translate heterogeneous specifications into protocol-correct UVM environments and refine stimuli [79]. Compared with module-level benchmarks, this is closer to industrial verification because subsystems involve protocol interactions, generated interconnect, and multiple coordinated blocks. However, it also shows that successful LLM-aided UVM generation currently depends on substantial handcrafted infrastructure: protocol libraries, IRs, templates, and tool-specific repair loops.

Other UVM-oriented and end-to-end verification studies support the same trend. Exploratory APB UVM case studies show that generative models can help scaffold protocol-specific UVM components, but also that human review is still needed for completeness and correctness [94]. Multi-agent verification frameworks and security-verification agents extend the idea beyond plain functional testbenches by combining LLM-generated collateral with simulator execution, assertion/security checks, or tool-guided debug loops [95, 96, 97]. These works are useful related literature even when their primary target is broader verification automation rather than testbench generation alone.

G. Training-Time Alignment and Execution-Aware Learning

Prompting and agentic refinement improve inference-time behavior, but they do not directly teach the model which testbench patterns produce high verification value. TB or Not TB addresses this by fine-tuning models with Coverage-

Driven Direct Preference Optimization (CD-DPO). It constructs PairaNet, a preference dataset of higher- and lower-quality testbenches labeled by simulation-derived coverage metrics, and trains models to prefer stimuli that improve coverage [98]. This is important because it turns coverage from a prompt-time hint into a training signal.

LLM4Cov similarly argues that online reinforcement learning with simulators is often too slow and expensive, so it builds an offline execution-aware agent-learning framework. The framework uses execution-validated data curation, policy-aware synthesis, and worst-state-prioritized sampling to train compact models for high-coverage testbench generation [99]. These training-time approaches indicate a likely future direction: instead of relying only on increasingly large general models, verification-specific generators may be trained on simulator-labeled examples, coverage traces, failed repair attempts, and tool-grounded preferences.

Structured Testbench Generation (STG) takes an even stronger stance against unconstrained LLM generation. Rather than iteratively prompting an LLM to synthesize testbenches, STG exploits hardware structure to generate deterministic testbenches, reduce false passes, and curate verification-oriented data for model training [100]. STG’s results support a recurring conclusion across this section: testbench generation benefits from deterministic structure wherever possible, with LLMs used for planning, classification, repair, or data augmentation rather than unrestricted final-code authorship.

H. Agentic Verification Workflows

The newest papers increasingly describe testbench generation as one component of an agentic verification loop. In these systems, the LLM may decompose a verification goal, generate or repair testbench code, execute tools, interpret compiler or simulator output, update coverage goals, and repeat. This matches the “prompt, verify, repeat” pattern articulated in recent verification-cycle work [97]. The important distinction from earlier self-debugging is that the agent owns more of the verification workflow: it can select which scenario to target, which tool output to inspect, and when to revise code or stimuli.

This agentic framing has two consequences for a survey. First, related work should include papers whose output is not only a testbench file but a verification process that includes testbench generation. PRO-V-R1, UCAgent, MAVF, and SV-LLM are examples because they use generated programs, tests, or verification collateral inside larger automatic RTL verification pipelines [89, 90, 95, 96]. Second, evaluation needs to include cost and convergence, not only final coverage: the number of compile/simulation attempts, failed repairs, prompts, and tool calls becomes part of the method’s practicality.

I. Benchmarks and Evaluation Metrics

A serious testbench-generation benchmark must evaluate several layers of correctness. At minimum, the generated artifact should compile, elaborate, and simulate. However, these

checks are only gateway metrics. A stronger benchmark asks whether the testbench drives meaningful scenarios, reaches structural and functional coverage targets, detects injected bugs or RTL mutants, and avoids false passes. AutoBench and CorrectBench make this explicit through multi-stage evaluation and mutant-based checks [76, 77]. CVDP broadens the benchmark landscape by including hardware design and verification tasks in both non-agentic and agentic forms, revealing that tool-using verification tasks remain difficult for current models [101]. Benchmarking should also include fuzzing and directed-test baselines because classical CDG, RFUZZ, DirectFuzz, and FuzzWiz attack the same coverage-closure objective with non-LLM search strategies [81, 83, 84, 85].

The most useful metrics for this subfield include:

- **Build success:** parsing, elaboration, simulator compatibility, and lint cleanliness;
- **Stimulus effectiveness:** line, branch, toggle, FSM-state, FSM-transition, and user-defined functional coverage;
- **Oracle strength:** ability to reject faulty designs, RTL mutants, or injected bugs;
- **Repair convergence:** number of tool-feedback iterations, token cost, and simulator runs required;
- **Scalability:** performance on sequential circuits, larger FSMs, bus protocols, and subsystem-level designs;
- **Portability:** whether generated collateral works across open-source and commercial simulators.

Figure 4 shows why later-stage metrics provide stronger evidence than syntax or compilation alone. Tables IV–VI summarize representative work across these dimensions.

J. Challenges and Future Directions

LLM-based testbench generation has evolved from generating directed tests for small RTL modules toward self-checking testbenches, coverage-guided refinement, UVM environments, and subsystem-level verification. In parallel, workflows have progressed from one-shot generation toward iterative and agentic approaches that incorporate simulation and coverage feedback. Despite this progress, unconstrained SystemVerilog and UVM generation remains unreliable for complex sequential and protocol-intensive designs. The strongest approaches therefore combine LLM reasoning with deterministic structures such as templates, intermediate representations (IRs), domain-specific languages (DSLs), protocol libraries, and verification-tool feedback.

A central challenge is **oracle quality**. Effective stimuli provide limited value without trustworthy mechanisms for determining correctness. High coverage or successful execution against golden RTL does not necessarily imply strong bug-detection capability. Mutation testing, imperfect RTL variants, and controlled bug injection therefore provide important complementary evaluation [76, 77, 100]. Future work should place greater emphasis on generating and validating scoreboards, reference models, assertions, and other checking mechanisms.

Sequential and temporal reasoning remains difficult for resets, handshakes, long-latency operations, pipelines, FSMs, and subsystem protocols. Existing approaches mitigate this



Evidence strength increases from left to right; later stages are harder to satisfy but more meaningful for signoff.

Fig. 4. Evaluation ladder for generated testbenches. Compilation is necessary, but coverage, oracle strength, mutation resistance, and portability are stronger indicators of verification value.

TABLE IV
REPRESENTATIVE RELATED WORK ON AUTOMATIC AND LLM-ASSISTED HARDWARE TESTBENCH GENERATION: CLASSICAL BASELINES AND STIMULUS-GENERATION METHODS.

Work	Primary target	Generation style	Feedback / evaluation	Main contribution	Key limitation
AVERT [80]	Verilog testbench construction for evolutionary hardware	Grammar/template-based generation from ports and test vectors	Simulation throughput on benchmark circuits	Shows that structured non-LLM generation can reduce manual testbench setup and improve simulation efficiency	Requires externally supplied vectors and does not infer semantic oracles
Bhandari et al. [86]	FSM testbench generation and bug detection	Prompted GPT-3.5/GPT-4 with iterative EDA feedback	Transition coverage and bug-detection traces	Demonstrates compiler/simulator feedback as a practical prompt-refinement signal	Scalability limited by trace length, state count, and context management
RFUZZ [83]	RTL fuzzing and coverage exploration	Coverage-guided fuzzing for RTL simulators	Coverage feedback and bug-finding experiments	Establishes fuzzing as a strong non-LLM coverage-directed baseline	Does not generate human-readable verification environments or semantic checkers
DirectFuzz [84]	Directed RTL test generation	Directed fuzzing with hardware-aware guidance	Targeted coverage and bug exposure	Improves reachability of hard-to-trigger RTL behaviors	Still needs external checking and target selection
FuzzWiz [85]	RTL fuzzing with hardware-specific feedback	Automated hardware fuzzing workflow	Coverage and crash/bug finding	Provides a recent fuzzing baseline for automated RTL stimulus generation	Optimizes exploration more than reusable testbench construction
LLM4DV [87]	Hardware test-stimuli generation	Coverage-directed prompting and missed-bin refinement	Functional coverage across multiple DUTs	Frames LLMs as stimulus generators inside a DV loop	Less focused on complete reusable testbench environments
VerilogReader [22]	Code-coverage-directed stimulus generation	LLM reads RTL plus coverage explanations	Verilator-style coverage and random-testing comparison	Introduces coverage and DUT explainers to translate tool output into LLM-readable goals	Degrades when designs exceed model comprehension scope
Spec2Cov [88]	Specification-driven coverage closure	Agentic prompt refinement with simulator interaction	Compilation, simulation, and code coverage over varied designs	Automates iterative stimulus generation from specifications	Full coverage remains difficult for complex designs

TABLE V
REPRESENTATIVE RELATED WORK ON AUTOMATIC AND LLM-ASSISTED HARDWARE TESTBENCH GENERATION: FULL TESTBENCH, UVM, AND SUBSYSTEM-LEVEL METHODS.

Work	Primary target	Generation style	Feedback / evaluation	Main contribution	Key limitation
AutoBench [76]	Full Verilog/Python hybrid testbench generation	Scenario, driver, and checker decomposition	Eval0-Eval2 metrics with mutant RTLs	Establishes a systematic LLM testbench-generation and evaluation framework	Syntax-focused repair still leaves functional checker errors
CorrectBench [77]	Functionally self-correcting testbench generation	Generator, validator, and corrector agents	Scenario validation and Eval2 pass ratio	Adds functional self-validation and self-correction without golden testbenches	Still relies on imperfect generated variants and benchmark-scale tasks
ConfiBench [91]	Robust testbench generation under uncertainty	Scenario masking and multi-testbench ensemble	Pass rate on AutoBench-style tasks	Uses confidence and ensembles to reduce risky generated behavior	Added complexity and cost; still task-benchmark centered
VerifLLMBench [92]	UVM testbench benchmark	LLM-generated UVM components with iterative repair	Build success, lint, simulation, and coverage	Provides an open methodology for evaluating LLM-generated UVM environments	Demonstration designs are relatively simple compared with industrial subsystems
UVM ² [93]	UVM testbench and stimulus generation	Automated UVM framework with coverage-guided refinement	Code and functional coverage on larger RTL designs	Moves LLM verification from simple benches toward reusable UVM workflows	Requires substantial tool orchestration and domain-specific setup
HAVEN [78]	UVM testbench and sequence synthesis	LLM planning plus protocol templates and sequence DSL	Compilation, code coverage, and functional coverage	Avoids raw UVM generation by using templates and protocol-aware DSLs	Protocol-specific engineering must be built and maintained
UVMarvel [79]	Subsystem-level UVM generation	IR, protocol library, signal tracking, and patching	Subsystem code coverage and automated runtime	Extends LLM-aided UVM generation to subsystem-level protocols	Public evidence remains recent and protocol-library dependent

challenge by decomposing behavior into scenarios, transactions, DSL operations, or targeted coverage gaps [22, 78, 86]. Extending such structured reasoning to longer traces and interacting protocols is important for scaling beyond small designs.

Industrial-scale UVM generation introduces additional challenges because useful environments must correctly connect components, obey protocol timing, implement meaningful scoreboards and coverage models, and integrate with simulator workflows. HAVEN, UVM², UVMarvel, and VerifLLMBench demonstrate progress, but also indicate that UVM generation benefits substantially from templates, IRs, and reusable protocol libraries [78, 79, 92, 93]. This suggests a hybrid approach in which LLMs perform high-level planning and reasoning while deterministic mechanisms generate structurally constrained verification components.

Scalability and evaluation also remain open problems. HDLBits- and VerilogEval-derived tasks provide reproducibil-

ity but are generally too small and public to establish industrial readiness and may introduce contamination concerns. CVDP and UVM-oriented benchmarks improve realism [92, 101], but broader evaluation is needed across hidden designs, large subsystems, mixed-language repositories, multiple simulators, and standardized bug suites. Richer datasets aligning specifications, RTL, testbenches, execution traces, coverage, and bugs would further support both evaluation and verification-aware training.

Finally, **execution-driven refinement can be costly**. Repeated LLM and simulator interactions increase inference cost, simulator usage, and verification time. Promising directions include learning from execution traces, deterministic structural generation, targeted repair, and agentic controllers that obtain more useful feedback from fewer tool runs [89, 90, 98, 99, 100]. Evaluation should therefore report simulator runs, refinement iterations, inference cost, and human intervention in addition to functional results.

TABLE VI
REPRESENTATIVE RELATED WORK ON AUTOMATIC AND LLM-ASSISTED HARDWARE TESTBENCH GENERATION: TRAINING-TIME AND AGENTIC METHODS.

Work	Primary target	Generation style	Feedback / evaluation	Main contribution	Key limitation
TB or Not TB [98]	Coverage-optimized stimulus generation	CD-DPO fine-tuning from coverage-labeled preferences	CVDP CID12 coverage metrics	Uses simulator-derived coverage as a training-time preference signal	Depends on curated paired data and simulator-labeled examples
LLM4Cov [99]	High-coverage agentic testbench generation	Offline execution-aware agent learning	Coverage pass rate under agentic evaluation	Learns from execution-validated trajectories without online RL cost	Benchmark and training protocol are recent; industrial generality remains open
PRO-V-R1 [89]	Automatic RTL verification agent	LLM plus reinforcement learning and simulation feedback	Program/test generation and verification outcomes	Treats verification as an executable feedback loop rather than static generation	Broader RTL-verification scope makes direct TB-only comparison difficult
UCAgent [90]	End-to-end block-level functional verification	Agentic state/goal exploration with tool feedback	Coverage closure and verification progress	Shows how test generation can be embedded in a full verification agent	Very recent preprint; benchmark generality remains evolving
MAVF [95]	Module-level verification automation	Multi-agent generative AI workflow	Verification artifact generation and tool execution	Demonstrates division of labor among agents for module-level verification	Industry-style framework with limited public reproducibility details
SV-LLM [96]	SoC security verification	Agentic LLM workflow for security checks and generated collateral	Security-verification outcomes	Extends LLM-generated verification artifacts to security-oriented SoC flows	Security focus differs from general functional testbench generation
STG [100]	Deterministic testbench generation and data curation	Structure-driven deterministic generation	Runtime, compile success, coverage, and false-pass reduction	Shows deterministic structure can outperform iterative LLM testbench synthesis in speed and reliability	Less flexible when verification intent is not recoverable from structure alone

Overall, existing work suggests that progress in LLM-based testbench generation depends less on unconstrained code-generation capability or model scale than on how effectively the LLM is integrated with the verification process. The field is consequently shifting from prompt-based testbench generation toward *verification-aware automation*, in which simulator outcomes serve as validation signals, refinement feedback, or training preferences, while deterministic structures constrain generation. Future progress will likely depend on stronger oracle generation, improved temporal and protocol reasoning, richer specification–RTL–testbench datasets, simulator-portable and contamination-resistant evaluation, and scalable workflows that combine LLM planning with deterministic verification infrastructure.

V. DEBUGGING AND DESIGN REPAIR

Sections III and IV examined how LLMs can assist pre-silicon verification by generating assertions, stimuli, testbenches, and other verification artifacts to expose incorrect RTL behavior. Once a failure is detected, however, compiler diagnostics, simulation mismatches, assertion violations, or formal counterexamples must be translated into an actionable diagnosis and a validated correction. This motivates *debugging and design repair*, which closes the verification loop by identifying the fault, determining its cause, correcting the RTL, and re-verifying the design.

As illustrated in Fig. 5, RTL debugging is an iterative failure-to-repair process. **Bug localization** narrows an observed failure to suspicious RTL modules, blocks, expressions, signals, or assignments using the RTL and failure evidence such as waveforms, assertion reports, or formal counterexamples [26, 102, 103]. **Root cause analysis (RCA)** determines how the localized logic produces the observed deviation, while **RTL repair** generates a corrective modification for re-verification [104]. If the failure persists or new violations emerge, the relevant stages are repeated. These stages should not be conflated: accurate localization does not establish the root cause, a plausible diagnosis does not guarantee a correct repair, and a patch that passes the available checks may not preserve behavior outside those checks. Source-only localization further identifies suspicious RTL without an observed failure, whereas assertion repair corrects the verification

property rather than the RTL implementation [26, 105, 106]. Manual execution of this loop remains a major verification bottleneck [102], increasingly aggravated by design scale and integration complexity [104].

RTL concurrency and hierarchy make this process particularly challenging. A defect may propagate across multiple modules and clock cycles before becoming observable, making the reported failure structurally and temporally distant from its source [102, 103]. Repair must further relate the suspected logic to the specification and timing behavior, remove the failure, and preserve unaffected functionality. Errors in localization or diagnosis can therefore propagate to invalid or incomplete repairs and additional debugging iterations [104].

LLMs can support this workflow by jointly processing RTL, specifications, diagnostics, simulation traces, and formal counterexamples to assist localization, causal diagnosis, and repair [26, 103, 107]. Hardware-specific methods strengthen these capabilities through RAG for external design or repair knowledge [108], RTL-specific fine-tuning [103, 109], explicit dependency representations [107], and multi-agent coordination of localization and repair [110, 111]. These techniques provide complementary mechanisms for implementing the failure-to-repair workflow.

In this section, we survey 25 papers on LLM-assisted hardware debugging, encompassing bug detection and localization, RCA, RTL and assertion repair, and integrated debugging workflows. Section V-A classifies the works by their primary debugging methodology, while Section V-B compares their coverage of the debugging pipeline. Section V-C examines debugging embedded within RTL generation workflows. Section V-D concludes with open challenges and future directions. General-purpose software-repair methods without hardware evaluation are excluded.

A. LLM-Based Debugging Methodologies Taxonomy

LLM-assisted debugging methods span multiple stages of the debugging pipeline but differ in how they acquire and use information to guide localization, diagnosis, and repair. To characterize these methodological differences, we group the reviewed works into four categories according to their principal mechanism: (1) prompting and tool-feedback-based debugging; (2) retrieval-augmented debugging; (3) model adaptation;

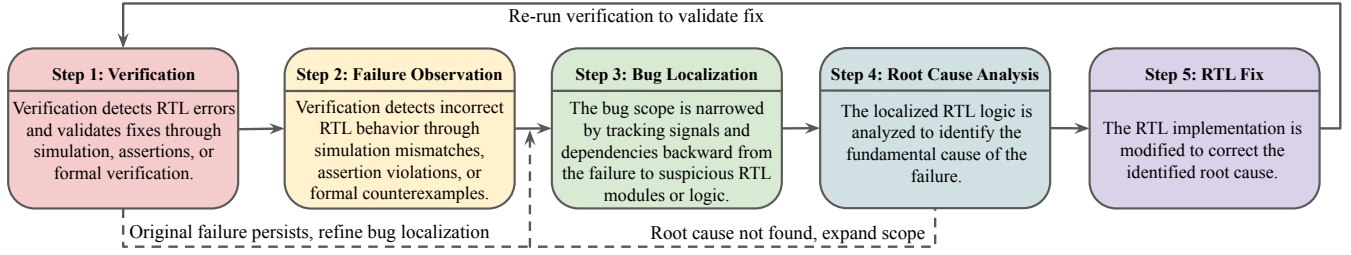


Fig. 5. Conventional RTL debugging process illustrating iterative bug localization, root cause analysis, RTL fixing, and re-verification.

and (4) structured and agentic reasoning. These categories capture complementary sources of knowledge and control, enabling comparison of their capabilities, requirements, and limitations beyond the pipeline stages they address. The categories are not mutually exclusive or indicative of capability. Many frameworks combine multiple mechanisms; each work is categorized by its principal contribution, with additional mechanisms identified in Tables VII and VIII.

1) *Category 1: Prompting and Tool-Feedback-Based Debugging*: This category uses prompting to guide LLMs in identifying defects or generating candidate repairs. Some methods regenerate RTL and analyze differences from the original code to locate potential defects [105]. Others incorporate compiler or simulation feedback to iteratively refine repairs [110]. These approaches primarily rely on knowledge encoded in the LLM and information provided in the prompt, without external retrieval or model parameter updates.

Prompting methods can inject task-specific knowledge without modifying model parameters. Ahmad et al. [112] investigate repair of human-introduced hardware security bugs using CWE-conditioned prompts and model/prompt sampling. Across 15 bugs, every case has at least one successful repair candidate, while GPT-4 produces 3,862 correct repairs in 7,500 attempts. For selected weaknesses, CWEAT is used for bug detection and post-repair checking. *Self-HWDebug* [113] instead derives reusable repair instructions from vulnerable/secure RTL pairs. Its two-shot configuration repairs all 25 test snippets spanning five CWE categories. Both approaches encode vulnerability knowledge through prompts rather than external retrieval or model adaptation. Beyond repair, prompting can support bug detection and localization. *FLAG* [105] prompts an LLM to regenerate RTL and uses differences from the original code, token confidence, and distance from comments as features for defect localization. Because it does not require a failing testbench, it can screen incomplete or non-compiling designs. Its line-level method detects 38 of 120 OpenTitan bugs, demonstrating how prompted RTL regeneration can provide diagnostic features for source-level screening.

Tool-feedback-based methods close the loop by using compiler or simulation results to evaluate generated repairs and guide subsequent revisions. *MEIC* [110] employs debugging and scoring agents, together with candidate history, rollback, and self-planning, to manage iterative repair. On 178 RTLLM-derived error instances, it reports syntax and functional fix

rates of 93% and 78%, respectively. *UVLLM* [114] further combines UVM checking with mismatch-driven data-flow slicing and test-score-based rollback. Time-aware slicing narrows the failure context presented to the LLM, while executable UVM checks guide candidate selection. Across 331 error instances from 27 RTLLM-derived modules, it reports syntax and functional fix rates of 86.99% and 71.92%, respectively.

These methods demonstrate that prompting and verification feedback can support both source-level detection and iterative RTL repair without model adaptation. Their effectiveness, however, depends strongly on the information exposed by the prompt and verification tools. Ambiguous diagnostics or weak functional feedback may not reveal the underlying cause, causing repeated generation to converge slowly or fail when the required hardware knowledge is unavailable. This limitation motivates retrieval-based methods that explicitly augment the LLM with external debugging knowledge.

2) *Category 2: Retrieval-Augmented Debugging*: Retrieval-augmented methods supplement the LLM with external information, such as specifications, expert repair examples, documentation, or design context, at inference time. This provides task-specific knowledge without modifying model parameters and can improve both diagnosis and repair when the required information is not reliably encoded in the LLM.

Qayyum et al. [115] use RAG to retrieve specification context for HDL bug identification and repair. Their method combines specification retrieval with expression decomposition and repairs all 25 nonconstant mutations across three OpenTitan IPs, but none of the five constant mutations. The failed constants are absent from the specifications, illustrating that retrieval effectiveness depends on the completeness of the underlying knowledge source.

Rather than retrieving specifications, *RTLFixer* [108] retrieves validated expert solutions associated with compiler-error patterns and combines them with ReAct-style prompting and compiler feedback. The framework reports a 98.5% compilation-error repair rate, with ablations showing complementary contributions from retrieval, iterative feedback, and diagnostic quality. *GROVE* [116] further makes the retrieval corpus adaptive by organizing previously resolved cases into an evolving knowledge tree with applicability conditions. Candidate knowledge is validated on its originating cases, while budget-aware retrieval selects relevant entries for new failures. On a module-disjoint SVA-Eval split, *GROVE* improves

Llama-3.3 pass@1 from 72.7% to 84.7% and achieves 86.9% pass@5. These results demonstrate the potential of reusable repair knowledge, although validation on an originating case does not guarantee transfer to other designs.

Retrieval can also select relevant design structure rather than textual repair knowledge. *HDLxGraph* [117] combines AST hierarchy with data-flow traversal to retrieve repository-level RTL context. Its debugging evaluation covers five mor1kx commits and uses ROUGE rather than functionally validated repair, so the results demonstrate context retrieval rather than repair correctness. *ViTAD* [118] instead augments signal dependencies with timing information and retrieves repair strategies conditioned on inferred causes of timing violations. Across 54 injected timing cases, *ViTAD* improves the fix rate from 54.38% to 71.05%, a gain of 16.67 percentage points.

Overall, retrieval-augmented debugging depends on the relevance and completeness of external knowledge and the ability of the LLM to apply it correctly.

3) Category 3: Debugging through Model Adaptation:

Model adaptation encodes hardware-specific debugging knowledge in model parameters through continued pretraining, supervised fine-tuning, preference optimization, or specialized training objectives. These methods aim to improve localization and repair, but their effectiveness depends on representative training data. Public datasets pairing faulty RTL with fault locations, diagnostics, and validated repairs remain limited.

HDLdebugger [103] addresses this data limitation by constructing a synthetic debugging corpus through reverse engineering. Controlled mutations of correct RTL are paired with compiler diagnostics and reference implementations to provide repair supervision, while code and document retrieval supplies additional context at inference time. The resulting framework reports an 81.93% execution pass rate on its test split drawn from 92,143 industrial HDL instances. *LLM4SecHW* [119] instead mines open-source hardware version histories for defect/fix pairs and descriptions and applies lightweight adapter-based fine-tuning. Its evaluation uses per-case ROUGE scores and qualitative examples rather than an aggregate functionally validated repair rate. These approaches illustrate two complementary sources of adaptation data: controlled synthetic defects and naturally occurring development histories.

Model adaptation can also specialize LLMs for fault localization. *LiK* [26] combines continual pretraining, supervised localization training, and SimPO preference optimization, achieving 93.38% localization pass@1 compared with 77.9% for GPT-4-Turbo on the same benchmark. Supplying its predicted locations to GPT-3.5 further improves downstream repair from 40.39% to 58.92%, demonstrating the value of localization as an intermediate debugging step. *VeriDebug* [120] instead integrates localization and repair within a shared model using contrastive representations of fault locations and bug types. It reports 67.2% top-1 localization accuracy and 64.7% top-1 repair accuracy, where repair requires an exact match to the reference corrected line. Thus, *LiK* and *VeriDebug* represent different interfaces between learned localization and repair: explicit localization for a downstream repair model

versus joint localization and correction.

Adaptation can further target repair generation or constrain its scope. *AssertSolver* [109] combines pretraining, supervised fine-tuning, and direct preference optimization over correct and incorrect responses for assertion-triggering RTL faults. It reports repair pass@1 and pass@5 of 88.54% and 90.00%, respectively, and generates explanations alongside candidate corrections. *ARSP* [121] instead trains semantic partitioning and fragment-repair models to restrict modification to a selected RTL region while retaining the specification and global code as context. On injected faults in 33 designs, it achieves 77.92% pass@1 and 83.88% pass@5. These methods address complementary aspects of repair: response-level supervision improves candidate generation, whereas learned partitioning constrains where modifications are applied.

Overall, model adaptation enables hardware-specific localization and repair knowledge to be internalized rather than repeatedly retrieved or supplied through prompts. The reviewed methods show that adaptation can improve localization, guide downstream repair, and specialize models for particular defect classes, while synthetic data, repository histories, and preference signals provide different forms of supervision. However, performance depends strongly on the quality and representativeness of the training data, and metrics such as exact patch match or textual similarity do not necessarily establish functional correctness. Moreover, parameter adaptation alone does not resolve failures whose causes span large RTL hierarchies, multiple clock cycles, or long verification traces. These limitations motivate the structured representations, explicit search, and agent coordination examined in the next category.

4) Category 4: Structured and Agentic Debugging: Prompting, retrieval, and model adaptation improve the knowledge available to an LLM, but complex RTL failures may still require reasoning across multiple signals, modules, clock cycles, and candidate repairs. Structured and agentic methods address this challenge by explicitly organizing failure evidence, intermediate hypotheses, repair search, or tool interactions. This category therefore includes graph-based reasoning, structured search, and agent coordination, with classification determined by the principal methodological contribution rather than the use of multiple agents alone.

FVDebug [107] structures formal counterexamples as a directed acyclic graph that captures signal dependencies across timesteps. A Graph Scanner uses batched LLM inference with paired for-and-against prompting to identify suspicious nodes, after which a reasoning agent develops causal hypotheses and generates patches. On 38 SVA-Eval-Human cases, *FVDebug* reports a Quality@Best score of 0.956 under an LLM judge and repair pass@1/pass@5 of 71.1%/86.8%. Its additional CVA6 and proprietary industrial examples primarily demonstrate RCA on larger designs and should not be interpreted as achieving the same repair rates. *VeriPilot* [122] instead aligns RTL variables with internal variables of a supplied golden model and traces control- and data-flow dependencies to guide iterative repair. On adapted CVD tasks, this approach improves GPT-4o repair success from 54.3% to 85.71%,

demonstrating the value of explicit reference alignment while requiring an executable golden model.

VeriTrace [123] introduces *Agentic Temporal Exploration*, in which an Inspector agent iteratively selects internal signals and time windows for hypothesis-driven waveform analysis before a separate RTL Debugger performs targeted repair. The Inspector reasons over the specification, RTL, and waveform evidence without direct access to the testbench, while candidate repairs are re-simulated against the supplied golden testbench and retained when they reduce mismatches. On VerilogEval-V2, *VeriTrace* achieves 100% with Claude Sonnet 4.5, with notable gains on sequential FSM tasks. However, repair acceptance is determined by the supplied testbench, so passing establishes consistency with the exercised behaviors rather than independent conformance to the complete specification; behaviors, corner cases, or temporal conditions omitted by the testbench remain unverified.

Structured reasoning can also organize the search over candidate repairs. *R3A* [111] separates multi-agent fault localization from patch exploration, using ranked candidate locations to initialize a stochastic tree-of-thoughts search. A heuristic balances exploration and exploitation under a fixed time budget, enabling *R3A* to solve 29 of 32 RTL-Repair cases, with mean pass@1 and pass@5 of 75.0% and 86.7%, respectively. *Clover* [124] extends this approach by dispatching repair tasks between LLM-based editing and SMT-based repair within a dialogue-state tree. It solves 31 of 32 RTL-Repair cases and achieves 87.5% mean pass@1, compared with 29 of 32 cases and 75.0% for *R3A*. These results suggest the potential of combining generative search with constrained symbolic repair, although the cross-study comparison does not isolate the contribution of SMT reasoning and both methods depend on the adequacy of the waveform checks.

Agentic workflows can also target debugging outputs other than RTL repair. *MARVEL* [125] coordinates a security-policy supervisor with tool-specialized executors to identify SoC vulnerabilities rather than generate patches. Its 51 reported findings comprise 19 vulnerabilities, 14 warnings, and 18 hallucinated findings, highlighting both the coverage and false-positive challenges of agentic security analysis. *Assert-Fix* [106] targets assertion repair by retrieving relevant RTL and distinguishing timing from logic errors before correction. It achieves formal assertion-fix rates of 73.7% – 83.8% across four designs, but assumes golden RTL and therefore addresses a different repair target from faulty-RTL correction.

LintLLM [126] illustrates structured diagnostic reasoning without an explicitly established multi-agent architecture. Its Logic-Tree prompt decomposes diagnosis into ordered steps, while a Defect Tracker tentatively fixes detected defects and reruns LLM-based detection to identify primary defects. Because this feedback comes from repeated LLM detection rather than independent EDA-tool validation, the resulting diagnoses still require external confirmation. Across 90 designs spanning 11 defect categories and three difficulty levels, it reports an 83.33% correct rate and a 12.22% false-positive rate.

Overall, structured and agentic methods complement the

knowledge-oriented mechanisms of the preceding categories by explicitly controlling how debugging evidence is represented, analyzed, and searched. Dependency graphs can expose spatial and temporal fault propagation, structured search can explore alternative repairs, and specialized agents or symbolic tools can decompose and constrain complex debugging tasks. These mechanisms are particularly useful when failures cannot be resolved from a single prompt or local code context, but their effectiveness remains bounded by the quality of the available verification evidence, reference models, and tool feedback. In practice, the four categories are complementary rather than exclusive: hybrid frameworks increasingly combine prompting, retrieval, model adaptation, structured reasoning, and verification tools to address different information and control requirements throughout the debugging process.

B. LLM-Based Debugging Works Across Pipeline Stages

The preceding taxonomy classifies LLM-based debugging methods by their primary mechanisms. We next compare the reviewed works by their coverage of the debugging pipeline in Fig. 5, focusing on bug localization, RCA, repair, and integrated re-verification. Tables VII and VIII summarize these capabilities and the corresponding evaluation evidence.

Bug localization. Localization methods differ in both the granularity of their outputs and how directly those outputs identify faulty RTL. *LiK* [26] predicts fault locations that guide downstream correction, while *VeriDebug* [120] jointly learns localization and repair. In contrast, *ARSP* [121] identifies a candidate repair region, and *HDLxGraph* [117] retrieves repository context relevant to diagnosis. These outputs serve related purposes but provide different levels of localization evidence. Notably, *LiK* shows that its predicted locations improve downstream GPT-3.5 repair from 40.39% to 58.92%, demonstrating practical value beyond localization accuracy. By comparison, candidate-region extraction or context retrieval alone does not establish line-level localization accuracy or improved repair. Localization should therefore be evaluated by both the specificity of the identified region and its demonstrated contribution to subsequent debugging.

Root cause analysis. RCA goes beyond localization by explaining how the suspected RTL produces the observed failure. Its evaluation should therefore distinguish a causally supported diagnosis from reasoning generated only as an intermediate step toward repair. *FVDebug* [107] separately evaluates generated causal hypotheses and checked repair outcomes, reporting a Quality@Best score of 0.956 for RCA in addition to its repair results. *AssertSolver* [109] also generates explanations with proposed corrections, but explanation generation alone does not establish causal accuracy. Similarly, a successful patch validates the correction only under the supplied checks and does not necessarily validate every step of its accompanying explanation. Evaluation scale must also be associated with the pipeline stage: *FVDebug*’s larger CVA6 and industrial demonstrations establish RCA capability at that scale, but not equivalent automated repair performance. RCA evaluation should therefore report whether causal hypotheses

TABLE VII

SUMMARY OF LLM-BASED HARDWARE DEBUGGING WORKS (PART I). CAT.: C1 = PROMPTING/MODEL GENERATION/TOOL FEEDBACK, C2 = RETRIEVAL, C3 = PARAMETER ADAPTATION, C4 = STRUCTURED/AGENTIC REASONING. BL = BUG LOCALIZATION, RCA = ROOT CAUSE ANALYSIS, FIX = RTL REPAIR, E2E = INTEGRATED FAILURE-TO-PATCH WORKFLOW WITH TOOL RECHECKING.

Paper	Cat.	BL	RCA	Fix	E2E	Core Technique	Bug Types	Key Result	Benchmark
<i>AutoVeriFix</i> [127]	C1	–	–	✓	✓	Python-assisted two-stage generation, iterative simulation feedback	Syntax, functional	GPT-4: 77.2% pass@1 (Human), 83.7% pass@1 (Machine), 83.5% pass@5 (RTLLM v2.0)	VerilogEval-Human/Machine, RTLLM v2.0, generation outcomes
<i>VerilogCoder</i> [128]	C1,4	(✓)	(✓)	✓	✓	Graph-based planning, debug agent, AST-based waveform tracing	Syntax, functional	GPT-4 Turbo: 94.2% workflow pass rate with supplied testbench feedback	VerilogEval-Human v2, 156 tasks, supplied golden testbenches
<i>On hardware security bug code fixes</i> [112]	C1	–	–	✓	(✓)	CWE-conditioned prompts, model/prompt sampling, selected CWEAT integration	Hardware security	15/15 cases have a successful candidate, GPT-4: 3,862/7,500 correct attempts	15 security bugs, separate functional-repair comparison
<i>MEIC</i> [110]	C1,4	(✓)	(✓)	✓	✓	Prompt-configured debug/scorer agents, candidate history and rollback	Syntax, functional	93% syntax and 78% functional fix rates	178 RTLLM-derived error instances
<i>UVLLM</i> [114]	C1,4	✓	(✓)	✓	✓	UVM checking, mismatch-driven DFG slicing, test-score rollback	Syntax, functional	86.99% syntax and 71.92% functional fix rates	331 error instances across 27 RTLLM-derived modules
<i>RTLFixer</i> [108]	C1,2,4	–	(✓)	✓	✓	ReAct and expert-guidance retrieval with compiler feedback	Compilation errors	98.5% compilation-error repair, not a functional repair rate	VerilogEval-derived syntax errors, additional RTLLM evaluation
<i>From Bugs to Fixes</i> [115]	C2	✓	(✓)	✓	–	Specification retrieval, expression decomposition and patching closure	Functional mutations	25/25 nonconstant mutations patched, 0/5 constant mutations	AON Timer, UART, and EDN, 30 injected mutations
<i>GROVE</i> [116]	C2,4	(✓)	(✓)	✓	–	Tool-validated knowledge tree, applicability conditions, snapshot-and-zoom retrieval	Assertion-triggering RTL faults	Llama-3.3: 84.7% pass@1, 86.9% pass@5, Jasper-checked	SVA-Eval: 732 training/183 test cases, module-disjoint split
<i>HDLdebugger</i> [103]	C2,3	(✓)	(✓)	✓	–	Reverse-engineered error pairs, code/document retrieval, supervised fine-tuning	Compiler-oriented HDL errors	81.93% execution pass rate on the reported test split	92,143 industrial HDL instances, 8:2 train/test split
<i>HDLxGraph</i> [117]	C2	(✓)	–	(✓)	–	AST hierarchy and DFG signal traversal for repository retrieval	Repository defects	Debugging assessed by ROUGE overlap, no functional repair pass rate	Five <code>mor1kx</code> commits, HDLSearch evaluates retrieval separately
<i>VTAD</i> [118]	C2,4	✓	✓	✓	–	Timing dependency graph, violation-path analysis, cause-conditioned repair knowledge	Setup/hold, CDC	GPT-4o: 71.05% fix rate versus 54.38% without <i>VTAD</i>	54 reported injected timing cases, Vivado-based evaluation
<i>LLM4SecHW</i> [119]	C3	✓	(✓)	✓	–	Repository-history defect/fix pairs and descriptions, adapter-based fine-tuning	Hardware defects, security	Per-case ROUGE and qualitative examples, no aggregate validated repair rate	Open-source hardware version histories, including OpenTitan

✓: explicitly implemented target/output, (✓): supporting operation or restricted demonstration, –: not separately established. E2E denotes automated tool rechecking within the declared workflow, not full-design verification, signoff, or independent proof of all unaffected behavior. An E2E dash does not mean that a paper omitted experimental validation. Category labels identify mechanisms, not publication order. Metrics retain each paper’s evaluation definition and are not directly comparable across rows.

are independently assessed and distinguish such evidence from intermediate reasoning used to guide repair.

Repair and closed-loop debugging. Repair methods differ in both the artifact being corrected and the criterion used to accept a candidate. For RTL repair, *RTLFixer* [108] evaluates compilation success, whereas *VeriDebug* [120] measures exact agreement with a reference corrected line. These criteria differ from behavioral validation through simulation, assertions, or formal checking, as used by other repair frameworks [107, 114]. *AssertFix* [106] further targets incorrect assertions while assuming golden RTL, and therefore addresses a different repair problem from correcting a faulty implementation. Closed-loop integration adds another dimension by automatically rechecking candidate repairs and using the resulting feedback for further correction. For example, *UVLLM* [114] combines candidate generation with UVM-based rechecking to support iterative repair. Such re-verification strengthens the debugging loop but establishes correctness only with respect to the checks

exercised by the workflow.

Overall, the reviewed methods provide increasingly broad coverage of the debugging pipeline, ranging from source-level localization to integrated localization, diagnosis, repair, and re-verification. However, pipeline coverage alone does not determine debugging capability or correctness. Localization outputs vary in granularity, RCA requires evidence that generated explanations reflect the observed failure, and repair results depend strongly on the acceptance criterion and behavioral oracle. Accordingly, comparisons should distinguish *which pipeline stages are automated* from *what evidence validates the output of each stage*. This distinction is particularly important for closed-loop workflows, where repeated tool feedback can improve repair success without guaranteeing correctness beyond the supplied verification checks.

TABLE VIII

SUMMARY OF LLM-BASED HARDWARE DEBUGGING WORKS (PART II). THE COLUMN DEFINITIONS FOLLOW TABLE VII. SO DENOTES SOURCE-ONLY SCREENING AS AN INPUT CONDITION, NOT AN ADDITIONAL CATEGORY. A SUPERScript A IDENTIFIES ASSERTION REPAIR RATHER THAN RTL REPAIR.

Paper	Cat.	BL	RCA	Fix	E2E	Core Technique	Bug Types	Key Result	Benchmark
<i>LiK</i> [26]	C3	✓	–	–	–	Continual pretraining, line-localization supervision, SimPO	Functional, RTL faults	93.38% localization pass@1, downstream GPT-3.5 repair: 40.39% to 58.92%	102 RTLLM-derived cases spanning five injected bug types
<i>VeriDebug</i> [120]	C3	✓	–	✓	–	Contrastive localization/type embeddings and guided repair with shared parameters	Functional, RTL faults	67.2% location Acc@1, 64.7% exact-match repair Acc@1	Synthetic functional defects, compilable RTL, held-out evaluation split
<i>AssertSolver</i> [109]	C3	✓	(✓)	✓	–	Pretraining, supervised fine-tuning, and DPO on correct/incorrect repair responses	Assertion-triggering, RTL faults	88.54% pass@1, 90.00% pass@5, assertion-failure resolution	SVA-Eval: 877 generated and 38 human-curated cases
<i>Self-HWDebug</i> [113]	C1	–	–	✓	–	Instructions synthesized from vulnerable/secure pairs, no weight update	Hardware security	Two-shot setting: 25/25 test snippets mitigated	Five CWE categories, five test snippets per category
<i>FVDebug</i> [107]	C2,4	✓	✓	✓	(✓)	Signal-event causal graph, balanced scanning, hypothesis exploration, patch generation	Formal-verification failures	Quality@Best 0.956 (LLM judge), repair pass@1/pass@5: 71.1%/86.8%	38 SVA-Eval-Human cases, additional CVA6 and industrial RCA cases
<i>R3A</i> [111]	C4	✓	(✓)	✓	✓	AST-segmented localization agents, stochastic tree-of-thoughts patch search	Structural, functional	29/32 cases solved, mean pass@1 75.0%, pass@5 86.7%	RTL-Repair, acceptance against reference waveforms
<i>LintLLM</i> [126]	C4, SO	✓	(✓)	(✓)	–	Logic-Tree prompting, tentative fixes and LLM redetection to isolate primary defects	Syntax, lint/semantic defects	o1-mini: 83.33% correct rate, 12.22% reported false-positive rate	90 designs, 11 defect categories, three difficulty levels
<i>FLAG</i> [105]	C1, SO	✓	–	–	–	Static targeting, regeneration differences and token probabilities, line classifier	Security, functional	Line-level detection: 51/201, token top-five: 47/201	OpenTitan (120), HackAtDAC (42), CirFix (30), CWE examples (9)
<i>ARSP</i> [121]	C3,4	(✓)	–	✓	–	Trained semantic partitioning and fragment repair, controlled recombination	Functional, RTL faults	77.92% pass@1, 83.88% pass@5, testbench-checked	Injected faults in 33 RTLLM/GenBen designs, mean 219 RTL lines
<i>VeriPilot</i> [122]	C4	✓	(✓)	✓	✓	Golden-model variable alignment, CDFG tracing, reference-guided iterative repair	Functional, RTL faults	GPT-4o on adapted CVDP: 54.3% to 85.71% repair success	Strider and adapted CVDP tasks, executable golden models supplied
<i>Clover</i> [124]	C4	(✓)	(✓)	✓	✓	Agentic dispatch to LLM/SMT repair, dialogue-state tree search, RTL toolbox	Structural, functional	31/32 RTL-Repair cases solved, mean pass@1 87.5%	RTL-Repair, additional random-logic synthetic evaluation
<i>MARVEL</i> [125]	C2,4	✓	(✓)	–	–	Security-policy supervisor and tool-specialized detection executors	SoC security	51 reports: 19 vulnerabilities, 14 warnings, 18 hallucinations	OpenTitan-based Hack@DATE SoC
<i>AssertFix</i> [106]	C2,4	(✓)	✓	✓ ^A	(✓) ^A	Related-RTL retrieval, timing/logic classification, assertion-specific correction	Incorrect SVA timing/logic	Formal assertion fix rates: 73.7%–83.8% across four designs	I2C, ECG, Pairing, SHA3, golden RTL assumed

BL includes candidate-region extraction. It does not imply measured line-level accuracy. RCA marked as supporting denotes explanatory reasoning or defect classification without an independently evaluated causal-diagnosis task. *LintLLM*'s edits support internal diagnosis. *FVDebug*'s large-design demonstrations concern RCA and should not inherit its small-benchmark repair rates. Assertion repair and source-only detection are included for their debugging role but remain distinct from RTL functional repair.

C. Debugging as a Component of LLM-Based RTL Generation Framework

The preceding subsections consider debugging primarily as the diagnosis and repair of faulty RTL. Debugging also serves as an integral component of LLM-based RTL generation, where generated candidates are iteratively corrected before being accepted. In this setting, correction occurs at two levels: compilation errors can be addressed using compiler diagnostics, whereas functional errors require behavioral feedback from a testbench or reference model.

RTLFixer [108] illustrates compilation-oriented correction of LLM-generated Verilog. It combines compiler feedback with retrieved expert repair guidance to iteratively revise erroneous candidates. This improves RTL executability, but compilation alone does not establish functional correctness.

Functional correction requires stronger behavioral feedback. *VerilogCoder* [128] integrates graph-based task planning with

coding and debugging agents. Its debugging agent uses simulation mismatches and AST-based waveform tracing to revise generated RTL against a supplied testbench. With GPT-4 Turbo, the complete workflow achieves a 94.2% pass rate on 156 VerilogEval-Human v2. *VeriTrace* [123] extends this paradigm by allowing a dedicated Inspector agent to iteratively select internal signals and temporal windows for hypothesis-driven waveform analysis before targeted RTL repair.

AutoVeriFix [127] instead constructs its own behavioral reference from the natural-language specification. Its first stage generates a Python reference model and tests, which are refined using Python coverage feedback. The second stage generates Verilog and iteratively corrects compilation errors and behavioral mismatches against this reference. With GPT-4, *AutoVeriFix* achieves 77.2% pass@1 on VerilogEval-Human, 83.7% pass@1 on VerilogEval-Machine, and 83.5% pass@5 on RTLLM v2.0. These results characterize the complete

generation-and-correction workflow rather than repair of a fixed set of pre-existing RTL defects.

Another recent work, *VeriRefine* [129], also integrates verification and debugging directly into RTL generation through a structured intermediate representation. Unlike *VerilogCoder*, which uses a supplied benchmark testbench to guide debugging, *VeriRefine* generates its verification environment from the specification, keeping both generation and correction grounded in the same source of intended behavior.

Overall, debugging turns RTL generation into an iterative process guided by verification feedback. Its reliability depends on both the coverage of the checks and the correctness of the expected behavior. Verification artifacts generated from the specification may share interpretation errors with the RTL. Evaluation should therefore report feedback access and iteration budgets and use benchmark simulation or properties withheld during repair to assess functional correctness.

D. Future Research Direction: Self-Evolving Hardware Debugging Systems

The reviewed methods demonstrate a progression from prompt-based correction toward retrieval, model adaptation, structured reasoning, agentic search, and verification-guided repair. However, most systems learn little from completed debugging tasks and restart subsequent tasks with the same knowledge and strategies. A promising direction is *self-evolving hardware debugging*, in which agents accumulate validated experience across designs and revisions and use it to improve bug localization, diagnosis, repair, and verification.

A self-evolving system should retain more than successful patches. Each debugging episode should associate the failure and repair with its fault location, causal evidence, relevant signal dependencies, specification assumptions, applicability conditions, and validation results. *GROVE* [116] provides an initial example of organizing validated repair knowledge for reuse, while *MEIC*, *R3A*, and *Clover* [110, 111, 124] maintain candidate states and feedback within individual debugging sessions. Extending such mechanisms across tasks could allow previous experience to guide retrieval, fault localization, repair selection, and tool invocation rather than merely provide another repair example.

A central challenge is *reliable transfer across designs and revisions*. A repair that succeeds on one design may become invalid when interfaces, timing behavior, implementation structure, or specifications change. Persistent knowledge must therefore retain the conditions under which it was learned and be revalidated before reuse rather than treating every successful repair as universally applicable. Evaluation should correspondingly use sequences of unseen designs or design revisions and compare systems with and without accumulated experience under equivalent resource budgets. Such experiments should measure whether experience reduces localization effort and improves independently validated repair rates without increasing regressions or false positives.

A second challenge is *evolving debugging strategies rather than only accumulating repair knowledge*. The structured and

agentic methods reviewed above show that debugging performance depends on decisions about which fault hypotheses to explore, which signal or temporal dependencies to trace, which context to retrieve, and when to invoke simulation, formal, or symbolic tools. Future agents could adapt these policies from previous validated outcomes, progressively learning which reasoning and tool-use strategies are effective for different failure classes. Such adaptation must be separated experimentally from gains due simply to additional retrieval, repair attempts, or computation. Runtime, token consumption, tool calls, and solver budgets should therefore accompany repair accuracy when evaluating self-evolving strategies.

A third challenge is maintaining *independent and trustworthy verification*. The preceding studies show that compiler success, exact patch matching, simulation tests, assertion checking, and formal verification provide different levels of evidence, and no single metric establishes correctness in all settings. Moreover, when the same testbench or reference model guides repair and determines acceptance, the system can overfit to the available verification environment. A self-evolving agent could amplify this problem by retaining such locally successful but incomplete repairs as future knowledge. New experience should therefore be admitted to persistent memory only after independent validation, preferably using held-out tests, properties, or formal checks that are not available during repair. The agent should also distinguish faults in the RTL from errors in assertions, reference models, specifications, or environmental assumptions rather than modifying the artifact that defines its own acceptance criterion.

Finally, scalability remains an open challenge because realistic failures can span modules, clock cycles, and long execution traces. Current approaches demonstrate the benefits of dependency graphs, slicing, retrieval, fine-tuning, and agent decomposition, but evidence for large-scale end-to-end repair remains limited. Future systems will require compact representations of spatial and temporal failure evidence, hierarchical localization and reasoning, selective retrieval, and adaptive tool invocation to control context and computational cost. Human oversight will also remain important for ambiguous specifications, high-impact repairs, and final acceptance criteria.

Overall, existing work establishes that LLMs can support individual stages of hardware debugging and that combining prompting, retrieval, adaptation, structured reasoning, and verification feedback can substantially strengthen closed-loop repair. The next step is to move from isolated debugging episodes toward systems that *learn from validated experience while preserving independent verification*. Achieving this goal requires reliable cross-design transfer, persistent knowledge with explicit applicability conditions, adaptive yet resource-efficient debugging policies, scalable reasoning over temporal and hierarchical RTL behavior, and evaluation against verification evidence withheld from the learning and repair process. Addressing these challenges is essential for transforming LLM-based debugging from task-specific repair automation into dependable hardware debugging systems that improve across designs without accumulating incorrect or obsolete

knowledge.

VI. LLMs FOR FORMAL VERIFICATION

Simulation-based validation remains indispensable in hardware verification because of its flexibility and scalability; however, it explores only a finite set of input sequences and execution scenarios and therefore cannot guarantee the absence of bugs. Formal verification (FV), in contrast, provides exhaustive mathematical guarantees by proving that a design satisfies specified properties over all states and inputs admitted by a formal model [35, 130]. Despite these guarantees, practical adoption of FV is hindered by two fundamental bottlenecks. The first is the specification bottleneck, which requires engineers to translate design intent into precise properties, environmental assumptions, invariants, and verification models. The second is the computational bottleneck arising from state-space explosion and the complexity of SAT/SMT-based reasoning for large designs. Industrial FV further requires engineers to select appropriate proof engines and strategies, analyze counterexamples, diagnose root causes, and iteratively refine verification models. These challenges are compounded by incomplete or ambiguous specifications, bounded-verification limitations, vacuous properties, multi-property interactions, and solver scalability, making formal verification both labor intensive and computationally demanding.

Recent advances in large language models (LLMs) have created opportunities to address both the specification and computational bottlenecks of formal verification. By processing natural-language specifications, RTL, assertions, source code, and verification traces, LLMs can generate candidate properties, invariants, verification models, proof strategies, and solver optimization heuristics, while also assisting with counterexample analysis and verification workflow orchestration. Rather than relying directly on LLM outputs, recent research increasingly adopts a neuro-symbolic paradigm in which LLMs perform semantic reasoning and heuristic generation, while model checkers, SAT/SMT solvers, and other formal engines validate the generated artifacts and provide corrective feedback. This separation combines the flexibility of generative models with the soundness of formal verification and has enabled applications spanning property generation, model checking, logic equivalence checking, solver optimization, and end-to-end verification workflow automation.

SystemVerilog assertion generation using LLMs has been treated in Section III. This section reviews the emerging landscape of LLM-driven formal verification and examines how LLMs are being integrated with formal engines to address proof construction, equivalence checking, solver optimization, and workflow automation. It also analyzes the limitations that currently constrain reliability and scalability, including hallucination, limited context and reasoning depth, data scarcity, benchmark overfitting and contamination, verification latency, and solver scalability. By organizing existing approaches according to their verification objectives, LLM roles, formal back ends, and evaluation methodologies, this review identifies common design patterns and limitations and highlights

research directions toward reliable, scalable, and increasingly autonomous LLM-assisted formal verification for production-grade hardware.

A. LLM-Assisted Proof Generation and Model Checking

Beyond formal property generation, recent research has increasingly employed LLMs to assist the proof process itself. These approaches use LLMs to construct intermediate proof artifacts, guide proof search, optimize model-checking algorithms, and coordinate verification workflows, thereby moving LLMs deeper into the formal verification pipeline while retaining conventional formal engines as the source of correctness guarantees.

One important direction is incremental and hierarchical proof generation, which decomposes large verification problems into smaller proof obligations. Qayyum *et al.* [131] proposed an LLM-assisted framework that partitions a design into smaller modules and generates module-level invariants and interface properties. Each generated property is independently validated using SAT/SMT-based formal verification before being composed into higher-level proofs. This decomposition reduces the reasoning burden on the LLM and improves proof scalability by avoiding direct reasoning over the complete flattened design.

A closely related direction is LLM-assisted invariant synthesis, where the LLM generates candidate inductive invariants that are subsequently validated by formal engines. Hassan *et al.* [132] combined GPT-4-generated invariants with formal verification and mutation testing. Mutation testing provides a stronger assessment of invariant quality than syntactic validity or proof success alone because it evaluates whether generated invariants can distinguish the original design from intentionally modified implementations, thereby exposing semantically weak or vacuous properties. CIII [133] further introduced a counterexample-to-induction (CTI)-guided approach in which candidate invariants are checked using bounded model checking and inductiveness checking, and counterexamples are returned to the LLM to guide iterative refinement. Integration with IC3 and k -induction enabled verification of multiple RISC-V processor designs. Together, these studies demonstrate the potential of closed-loop LLM-formal-engine interactions for synthesizing nontrivial proof artifacts, while also showing that effectiveness depends strongly on the quality of counterexample interpretation and the efficiency of formal feedback.

A complementary direction applies LLMs not to individual proof artifacts but to the model-checking algorithms themselves. IC3-Evolve [134], for example, uses an LLM offline to evolve IC3/PDR heuristics through auditable code modifications. Candidate changes are accepted only when SAFE executions produce independently verifiable proof certificates and UNSAFE executions generate replayable counterexamples. The resulting optimized model checker operates without an LLM at runtime, separating heuristic discovery from trusted proof reasoning. This approach illustrates an important distinction between LLM-assisted verification, where the LLM

participates in an individual verification instance, and LLM-driven verification optimization, where the LLM searches for improvements to the underlying formal engine.

Recent work has further expanded the role of LLMs from assisting individual verification tasks to orchestrating complete formal verification workflows. Saarthi [20], for example, organizes specification analysis, verification planning, assertion generation, property proving, counterexample analysis, and coverage closure into an integrated multi-agent workflow coupled with commercial formal verification tools. Although its reported end-to-end automation rate remains approximately 40%, the framework demonstrates a transition toward agentic formal verification in which multiple reasoning and verification activities are coordinated within a unified workflow.

Collectively, these studies indicate an evolution from LLM-based generation of individual proof artifacts toward closed-loop proof refinement, model-checker optimization, and multi-agent verification orchestration. The current state of practice remains fundamentally neuro-symbolic: LLMs are effective as generators, heuristic search mechanisms, decomposition agents, and counterexample interpreters, whereas SAT/SMT solvers, bounded model checkers, IC3/PDR, and commercial formal engines remain responsible for establishing correctness. This division of responsibility is essential because linguistic plausibility cannot establish the soundness or completeness required for formal hardware verification.

B. LLM-Assisted Logic Equivalence Checking

Compared with LLM-assisted property generation and model checking, the use of LLMs for logic equivalence checking (LEC) remains relatively immature. LEC determines whether two hardware implementations exhibit equivalent behavior and is routinely used for RTL-to-RTL comparison, RTL-to-gate verification, and validation of synthesis and engineering-change-order (ECO) transformations. Because conventional LEC provides a proof or counterexample rather than a statistical confidence estimate, current LLM-based approaches generally use LLMs in conjunction with formal equivalence engines rather than as replacements for them. Two complementary directions have emerged: learning-based prediction of equivalence relationships and the use of formal equivalence checking as a correctness oracle for LLM-based RTL generation and refinement.

DeepRTL2 [135] explores RTL equivalence as a representation learning problem. It constructs equivalent and non-equivalent RTL pairs through a feedback-driven process in which GPT-4o generates functionality-preserving RTL transformations and a conventional Yosys LEC flow determines whether the resulting designs remain equivalent. The generated pairs are subsequently used to train representations for equivalence classification. DeepRTL2 reports 68.1% accuracy and 72.3% F1 score, outperforming the evaluated general-purpose embedding baselines. However, the learned predictor serves primarily as a filtering mechanism rather than a replacement for formal equivalence checking because statistical classification cannot provide exhaustive correctness guarantees.

A more prevalent approach treats formal equivalence checking as a correctness oracle within LLM-driven RTL generation. FormalEval [136] uses sequential equivalence checking to compare generated RTL with a reference implementation and returns either a proof result or a counterexample. The formal result therefore provides an objective functional-correctness signal for evaluating and refining LLM-generated RTL. FormalRTL [137] extends this paradigm by coupling LLM-based planning and RTL synthesis with formal equivalence checking and software reference models as executable specifications. Generated RTL is accepted or revised according to formal feedback, transforming RTL generation from a one-shot synthesis task into an iterative generate–verify–repair process.

These studies reveal a common pattern between LLM-assisted model checking and LEC: in both settings, LLMs primarily perform semantic reasoning and generate or refine verification artifacts, while formal engines provide objective feedback that guides subsequent iterations. The key distinction lies in the verification objective: model-checking approaches use LLMs to construct invariants, decompose proof obligations, interpret counterexamples, or guide proof search for properties of a design, whereas LEC approaches use LLMs to reason about correspondences between implementations and to synthesize or repair RTL under equivalence constraints. On the other hand, this common *generate–formally validate–refine* paradigm suggests that techniques developed for LLM-guided proof refinement, counterexample interpretation, and hierarchical decomposition could potentially be transferred to LEC, particularly for correspondence identification and equivalence-problem partitioning. It is also interesting to extend the role of LLMs toward broader proof assistance, including identification of correspondence points, abstraction and partitioning strategies, and interpretation of equivalence counterexamples.

C. LLM-Driven SAT/SMT Solver Optimization

Modern formal verification engines, including IC3/PDR-based model checkers, rely heavily on SAT/SMT solvers as their core reasoning engines. Their performance is therefore strongly influenced by solver heuristics and implementation choices, including branching, clause learning and deletion, pre-processing and inprocessing, restarts, and theory propagation. This dependency has motivated research on using LLMs to automate the discovery and optimization of solver heuristics and implementations.

Non-LLM solver specialization provides an important baseline for this research direction. Su *et al.* developed GipSAT [138], a lightweight CDCL solver specialized for IC3 workloads by exploiting structural properties of IC3-generated SAT queries. GipSAT reduces unnecessary decisions, replaces the conventional VSIDS binary heap with a bucket-based data structure, and optimizes temporary-clause management, achieving an average $3.61\times$ speedup over a MiniSat-based IC3 implementation on HWMCC’20 and HWMCC’24 benchmarks. The subsequent rIC3 framework further combines SAT-level optimizations with IC3 improvements such as dynamic generalization and internal-signal-based reason-

ing [139]. These results establish that substantial gains can arise from tailoring both solver and model-checking strategies to verification-specific workloads and motivate LLMs as mechanisms for automating such optimization.

Based on these observations, AutoSAT [140] and SolSearch [141] use LLMs to generate and iteratively refine CDCL solver heuristics or source-code modifications based on benchmark performance. AutoModSAT [142] further combines prompt optimization with evolutionary search to explore modular SAT solver configurations. These approaches formulate solver optimization as an empirical search loop in which the LLM proposes a modification, the resulting solver is compiled and evaluated on representative instances, and performance feedback guides subsequent iterations. The results demonstrate that LLMs can explore a broad design space of solver heuristics and implementation strategies that would otherwise require substantial manual expertise.

LLM-based optimization has also been explored for SMT reasoning. LLMs can generate and refine SMT constraints or solver API code from high-level objectives, translating informal requirements into SMT-LIB formulations or APIs such as Z3 Optimize [143]. For solver optimization, however, syntactic validity is insufficient: generated formulations and solver modifications must preserve semantic correctness while improving solving efficiency and avoiding unsupported or computationally inefficient theory combinations. Consequently, the central challenge is to jointly evaluate correctness and performance rather than optimize either criterion in isolation.

Overall, existing work have demonstrated the search capability of LLMs for SAT/SMT solver optimization, particularly when candidate generation is combined with automated compilation, correctness checking, and empirical performance evaluation. However, current results remain vulnerable to benchmark overfitting, unstable optimization trajectories, compilation and execution failures, and loss of generality when optimizations are tuned to particular workloads. Future progress therefore requires evaluation across diverse formal-verification workloads and explicit measurement of both solver performance and generalization, rather than reporting improvements on a single benchmark distribution.

D. Cross-Cutting Challenges and Future Directions

Across these research directions, the state of the art is moving beyond one-shot LLM generation toward closed-loop, tool-grounded, and increasingly agentic formal verification. Current systems can generate assertions and invariants, decompose verification problems, interpret counterexamples, guide proof search, optimize solver implementations, and coordinate multi-stage verification workflows. The most reliable architectures share a common pattern: the LLM performs semantic reasoning, candidate generation, heuristic exploration, or workflow coordination, while deterministic formal tools provide correctness feedback and ultimately establish the validity of the generated artifacts. Nevertheless, current systems remain far from autonomous production-grade verification, and the primary research challenge is no longer simply whether LLMs

can generate useful verification artifacts, but whether they can do so reliably, efficiently, scalably, and with sufficient generalization across designs and verification tasks.

First, correctness remains fundamentally different from linguistic plausibility. LLM-generated assertions, invariants, proof strategies, or solver modifications may appear reasonable while being incomplete, over-constrained, vacuous, or semantically incorrect. Formal validation is therefore essential, but validation alone does not guarantee useful verification if the generated properties do not adequately capture design intent. Future systems should jointly address property quality and proof validity through mechanisms that detect vacuity, measure specification coverage, identify missing behaviors, and use counterexamples to refine both the generated artifact and its underlying specification.

Second, scalability remains a central limitation at both the LLM and formal-engine levels. Large industrial designs exceed practical context limits and contain long-range dependencies that are difficult for LLMs to preserve across iterative interactions. At the same time, formal engines face state-space explosion and expensive SAT/SMT reasoning. Hierarchical verification, modular context management, retrieval of design-relevant information, and multi-agent decomposition are promising approaches, but their effectiveness must be evaluated against the cost of maintaining consistency across agents and abstraction boundaries. In particular, future systems should avoid merely shifting the scalability problem from the formal engine to an increasingly large LLM context.

Third, the interaction between LLMs and formal tools introduces a significant latency bottleneck. Proof attempts and SAT/SMT evaluations may require minutes or hours, making naive generate–verify–repair loops prohibitively expensive. Research should therefore investigate adaptive feedback mechanisms that determine when formal evaluation is necessary, surrogate models that prioritize promising candidates, and compact representations of counterexamples, proof failures, and solver traces that preserve the information most useful for subsequent LLM reasoning.

Fourth, generalization and evaluation remain insufficiently understood. Many existing studies rely on small benchmark suites or task-specific datasets, making it difficult to determine whether reported gains reflect genuine reasoning improvements or overfitting to particular designs, coding styles, solver configurations, or benchmark distributions. Future evaluations should include larger and more diverse hardware designs, unseen architectures, cross-benchmark testing, multiple LLMs, and controlled ablation studies that isolate the contributions of prompting, retrieval, agentic decomposition, formal feedback, and solver optimization. Evaluation should also report not only correctness and proof success, but verification time, tool-call cost, coverage, robustness across repeated runs, and the degree of human intervention required.

Finally, a major opportunity lies in developing hardware-aware agentic systems that jointly reason about specifications, RTL structure, formal properties, proof strategies, and the underlying SAT/SMT infrastructure. Rather than treating these

tasks as independent stages, future frameworks could establish feedback loops in which failures at one level inform decisions at another—for example, counterexamples could guide property refinement, failed proofs could trigger abstraction or decomposition, and solver-performance feedback could influence model-checking strategies. Such systems should remain grounded by independently checkable proof artifacts and formal correctness oracles, enabling increasing levels of automation without sacrificing the soundness guarantees that distinguish formal verification from statistical software generation.

VII. CONCLUSION AND FUTURE DIRECTIONS

This survey examined the emerging role of LLMs across hardware functional verification, including assertion generation, stimulus and testbench generation, bug localization and design repair, formal verification, equivalence checking, solver optimization, and agentic verification workflows. Across these areas, a common pattern emerges: LLMs are most effective not as replacements for established verification algorithms, but as semantic reasoning and orchestration layers that connect specifications, design artifacts, verification evidence, and deterministic EDA tools. Prompting, retrieval, model adaptation, structured representations, and agentic reasoning improve different aspects of this interaction, while simulators, formal engines, coverage tools, and SAT/SMT solvers remain essential for evaluating generated artifacts and establishing correctness.

A second cross-cutting finding is the growing importance of closed-loop, verification-grounded workflows. Compilation diagnostics, simulation traces, coverage reports, counterexamples, proof outcomes, and solver statistics increasingly serve not only as acceptance signals but also as feedback for localization, refinement, search, and tool selection. This feedback-driven paradigm appears across assertion and testbench generation, RTL debugging and repair, model checking, equivalence checking, and solver optimization. However, successful interaction with a verification tool should not be confused with complete verification: an artifact may satisfy the available checks while still failing to capture behaviors or requirements that those checks do not represent. The central challenge is therefore not merely generating executable or tool-accepted artifacts, but preserving alignment between *design intent* and the *verification evidence* used to justify correctness.

This observation motivates a broader research direction toward *specification-centered verification intelligence*. Future systems should maintain explicit relationships among requirements, RTL, assertions, stimuli, reference models, coverage objectives, counterexamples, and repair decisions rather than treating these artifacts independently. Such traceability would enable an agent to determine whether a verification failure originates from the implementation, a generated verification artifact, an incorrect environmental assumption, or an incomplete specification. It would also provide a stronger basis for measuring verification completeness and for preventing implementation-specific behavior from silently redefining the intended specification. Structured intermediate representations,

retrieval, multimodal reasoning, and hierarchical hardware knowledge are promising mechanisms for maintaining these relationships across large designs.

Scalability represents a second high-level challenge. Current results are strongest on relatively localized tasks, whereas industrial verification requires reasoning across large RTL hierarchies, long temporal behaviors, heterogeneous verification artifacts, and expensive EDA-tool interactions. Simply increasing context length, agent count, or refinement iterations is unlikely to provide a sustainable solution because it can shift rather than eliminate the underlying complexity. Future systems should instead selectively retrieve and compress relevant information, reason hierarchically across abstraction levels, decompose verification objectives, and invoke expensive tools only when their expected information value justifies the cost. The longer-term goal is an adaptive neuro-symbolic workflow in which LLM reasoning determines *what* should be analyzed next while deterministic verification engines determine *whether* the resulting claims are valid.

A third challenge is establishing generalization and trustworthy evaluation. Reported results remain difficult to compare because studies differ substantially in benchmark scale, available design information, verification environments, model configurations, iteration budgets, and acceptance criteria. Public hardware repositories further introduce potential overlap between model training data and evaluation artifacts. Future benchmarks should therefore emphasize unseen and contamination-controlled designs, diverse architectures and abstraction levels, realistic specifications and defects, and evaluation across multiple models and verification tools. Evaluation should measure not only final artifact success but also verification effectiveness, robustness across repeated runs, computational and tool cost, and required human intervention. For iterative and agentic systems, controlled ablations are particularly important for distinguishing genuine improvements in reasoning from gains obtained through additional search, tool calls, or computation.

These considerations also change how autonomy should be viewed in hardware verification. The emerging trajectory is from one-shot artifact generation toward persistent agents that plan verification activities, interpret heterogeneous evidence, coordinate tools, and reuse validated experience across designs and revisions. Such systems could progressively improve retrieval, decomposition, debugging, and verification strategies, but persistent knowledge must retain its applicability conditions and validation evidence so that obsolete or design-specific conclusions are not propagated. Human engineers will remain essential for resolving ambiguous intent, reviewing high-impact decisions, defining verification objectives, and establishing signoff criteria. Accordingly, the objective should be dependable human-AI collaboration rather than unrestricted autonomous verification.

Overall, the reviewed literature shows that LLMs can substantially reduce the semantic and orchestration burden of hardware verification when coupled with deterministic verification infrastructure. The field has progressed from isolated

generation of assertions, tests, and repairs toward integrated workflows that reason over specifications, verification artifacts, failure evidence, and formal engines. The next major step is to make these workflows *specification-aligned, scalable, independently verifiable, and demonstrably generalizable*. Progress along these dimensions could transform LLM-assisted verification from a collection of promising task-level demonstrations into a dependable verification methodology in which LLMs provide flexible reasoning and automation while established EDA tools and engineering judgment preserve the correctness guarantees required for hardware design.

ACKNOWLEDGMENT

This work is supported by the National Science Foundation under Grant No. 2434247. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies.

REFERENCES

- [1] Apple, “Apple Introduces M2 Ultra.” <https://www.apple.com/newsroom/2023/06/apple-introduces-m2-ultra/>, June 2023. Accessed: Oct. 3, 2026.
- [2] NVIDIA, “NVIDIA Blackwell Platform Arrives to Power a New Era of Computing.” <https://nvidianews.nvidia.com/news/nvidia-blackwell-platform-arrives-to-power-a-new-era-of-computing>, Mar. 2024. Accessed: Oct. 3, 2026.
- [3] H. Witharana, Y. Lyu, S. Charles, and P. Mishra, “A survey on assertion-based hardware verification,” *ACM Comput. Surv.*, vol. 54, Sept. 2022.
- [4] B. Moyer, “Legacy process nodes going strong: The critical, and growing, significance of mature node chips and processes,” *Semiconductor Engineering*, July 2024.
- [5] H. Foster, “2024 Wilson Research Group IC/ASIC Functional Verification Trend Report,” 2024.
- [6] J. Pan, G. Zhou, C. Chang, I. Jacobson, J. Hu, and Y. Chen, “A Survey of Research in Large Language Models for Electronic Design Automation,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 30, no. 3, pp. 1–21, 2025.
- [7] D. Yu, H. Foster, and T. Fitzpatrick, “A survey of machine learning applications in functional verification,” in *Proc. DVCon*, 2024.
- [8] OpenAI, “Gpt-5.” <https://openai.com/gpt-5/>, 2025. Accessed: 2025-10-13.
- [9] Anthropic, “Claude.” <https://claude.com/product/overview>, 2025. Accessed: 2025-10-13.
- [10] Google DeepMind, “Gemini.” <https://gemini.google.com/>, 2025. Accessed: 2025-10-13.
- [11] 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 Transactions on Design Automation of Electronic Systems*, vol. 30, no. 3, 2025.
- [12] K. Xu, D. Schwachhofer, J. Blocklove, I. Polian, P. Domanski, D. Pfluger, S. Garg, R. Karri, O. Sinanoglu, J. Knechtel, Z. Zhao, U. Schlichtmann, and B. Li, “Large language models for electronic design automation,” 2025.
- [13] J. Wang, X. Chen, L. Zhang, and Z. Xie, “Real-bench: Benchmarking llms on real-world hardware design specifications,” in *The 62nd ACM/IEEE Design Automation Conference (DAC)*, ACM/IEEE, 2025.
- [14] S. Kumar, S. Mukhopadhyay, and S. Kundu, “Enhancing llm performance on hardware design tasks via structural alignment,” in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, IEEE, 2025.
- [15] S. Liu, W. Fang, Y. Lu, J. Wang, Z. Qijun, H. Zhang, and Z. Xie, “Rtlcoder: Fully open-source and efficient llm-assisted rtl code generation technique,” in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, IEEE, 2024.

- [16] C.-T. Ho, H. Ren, and B. Khailany, “Verilogcoder: Autonomous verilog coding agents with graph-based planning and abstract syntax tree (ast)-based waveform tracing tool,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 39, pp. 300–307, Association for the Advancement of Artificial Intelligence (AAAI), February 2025.
- [17] Y. Zhao, H. Zhang, H. Huang, Z. Yu, and J. Zhao, “Mage: A multi-agent engine for automated rtl code generation,” in *Proceedings of the 62nd ACM/IEEE Design Automation Conference (DAC)*, IEEE/ACM, 2025.
- [18] B. Nadimi and H. Zheng, “A multi-expert large language model architecture for verilog code generation,” in *Proceedings of the IEEE International Workshop on LLM-Aided Design (LAD)*, April 2024.
- [19] W. Sun, S. Zhou, X. Liao, and H. Yao, “Eda-copilot: A rag-powered intelligent assistant for eda tools,” *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2026.
- [20] A. Kumar *et al.*, “Saarthi: The first ai formal verification engineer,” in *Proceedings of DVCon U.S.*, 2025.
- [21] 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 *Design, Automation and Test in Europe (DATE)*, 2025.
- [22] R. Ma, Y. Yang, Z. Liu, J. Zhang, M. Li, J. Huang, and G. Luo, “Verilogreader: LLM-aided hardware test generation,” *arXiv preprint arXiv:2406.04373*, 2024.
- [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 *Proceedings of the 30th Asia and South Pacific Design Automation Conference, ASPDAC ’25*, (New York, NY, USA), p. 614–621, Association for Computing Machinery, 2025.
- [24] A. Gupta, B. Mali, and C. Karfa, “Sangam: Systemverilog assertion generation via monte carlo tree self-refine,” in *2025 IEEE International Conference on LLM-Aided Design (ICLAD)*, pp. 180–187, 2025.
- [25] Y. Bai, G. B. Hamad, S. Suhaib, and H. Ren, “Assertionforge: Enhancing formal verification assertion generation with structured representation of specifications and rtl,” in *2025 IEEE International Conference on LLM-Aided Design (ICLAD)*, pp. 85–92, 2025.
- [26] 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,” *arXiv preprint arXiv:2409.15186*, 2024. Version 2.
- [27] M.-H. Tsai *et al.*, “RTLFixer: Automatically Fixing RTL Syntax Errors with Large Language Models,” in *International Conference on VLSI Design*, 2024.
- [28] Z. Zang, Y. Song, A. Wang, B. W.-K. Ling, Q. Sun, Z. Lei, F. Yang, C. Zhuo, and J. Luo, “The dawn of agentic eda: A survey of autonomous digital chip design,” 2026.
- [29] M. Liu, T.-D. Ene, R. Kirby, C. Cheng, N. Pinckney, R. Liang, J. Alben, H. Anand, S. Banerjee, I. Bayraktaroglu, B. Bhaskaran, B. Catanzaro, A. Chaudhuri, S. Clay, B. Dally, L. Dang, P. Deshpande, S. Dhodhi, S. Halepete, E. Hill, J. Hu, S. Jain, A. Jindal, B. Khailany, G. Kokai, K. Kunal, X. Li, C. Lind, H. Liu, S. Oberman, S. Omar, G. Pasandi, S. Pratty, J. Raiman, A. Sarkar, Z. Shao, H. Sun, P. P. Suthar, V. Tej, W. Turner, K. Xu, and H. Ren, “Chipnemo: Domain-adapted llms for chip design,” 2024.
- [30] G. Yang, W. Zheng, D. Liang, P. Hu, Y. Yang, S. Peng, Z. Li, J. Feng, X. Wei, K. Sun, D. Ma, H. Cheng, Y. Shen, X. Chen, X. Hu, T. Y. Zhuo, and D. Lo, “Large language model for verilog code generation: Literature review and the road ahead,” 2025.
- [31] Anonymous, “AI/LLM-Based Security Verification: A Survey and AI Accelerator Case Study,” *arXiv preprint arXiv:2604.01572*, 2026.
- [32] H. Liu, Y. Lu, M. Wang, X. Yao, and B. Yu, “Llm-assisted circuit verification: A comprehensive survey,” in *Proceedings of the 31st Asia and South Pacific Design Automation Conference*, 2026.
- [33] Accellera Systems Initiative, *Universal Verification Methodology (UVM) User’s Guide*, 2015.
- [34] A. Gupta, “Formal hardware verification methods: A survey,” *Formal Methods in System Design*, vol. 1, no. 2–3, pp. 151–238, 1992.
- [35] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, MA, USA: MIT Press, 1999.
- [36] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, “Bounded Model Checking Using Satisfiability Solving,” *Formal Methods in System Design*, vol. 19, no. 1, pp. 7–34, 2001.
- [37] N. Een and N. Sörensson, “Minisat: A sat solver with conflict-clause minimization,” in *Theory and Applications of Satisfiability Testing*, vol. 3569 of *Lecture Notes in Computer Science*, pp. 502–518, Springer, 2005.
- [38] A. Biere, “Cadical at the sat race 2019,” in *Proceedings of SAT Race 2019: Solver and Benchmark Descriptions*, vol. B-2019-1 of *Department of Computer Science Series of Publications B*, pp. 14–15, University of Helsinki, 2019.
- [39] L. de Moura and N. Bjørner, “Z3: An efficient smt solver,” in *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 4963 of *Lecture Notes in Computer Science*, pp. 337–340, Springer, 2008.
- [40] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar, “cvc5: A versatile and industrial-strength smt solver,” in *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 13243 of *Lecture Notes in Computer Science*, pp. 415–442, Springer, 2022.
- [41] IEEE Technology Navigator, “Formal verification—Hardware verification.” <https://technav.ieee.org/topic/>

formal-verification/.

- [42] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is all you need,” in *Advances in Neural Information Processing Systems*, vol. 30, 2017.
- [43] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. V. Le, and D. Zhou, “Chain-of-thought prompting elicits reasoning in large language models,” in *Advances in Neural Information Processing Systems*, vol. 35, pp. 24824–24837, 2022.
- [44] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, “Retrieval-augmented generation for knowledge-intensive nlp tasks,” in *Advances in Neural Information Processing Systems*, vol. 33, pp. 9459–9474, 2020.
- [45] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, “React: Synergizing reasoning and acting in language models,” in *International Conference on Learning Representations*, 2023.
- [46] T. Schick, J. Dwivedi-Yu, R. Dessi, R. Raileanu, M. Lomeli, L. Zettlemoyer, N. Cancedda, and T. Scialom, “Toolformer: Language models can teach themselves to use tools,” in *Advances in Neural Information Processing Systems*, vol. 36, pp. 68539–68551, 2023.
- [47] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Zhu, L. Jiang, S. Zhang, A. H. Awadallah, R. W. White, D. Burger, and C. Wang, “Autogen: Enabling next-gen llm applications via multi-agent conversation,” *arXiv preprint arXiv:2308.08155*, 2023.
- [48] C. B. Hahn, “Towards improving verification productivity with circuit-aware translation of natural language to systemverilog assertions,” 2023.
- [49] F. Siddiqui *et al.*, “AutoSVA2: An Automated Framework for Formal Property Verification Using LLMs,” *arXiv preprint arXiv:2403.XXXXX*, 2024.
- [50] B. Mali, K. Maddala, V. Gupta, S. Reddy, C. Karfa, and R. Karri, “Chiraag: Chatgpt informed rapid and automated assertion generation,” 2024.
- [51] M. Shahidzadeh, B. Ghavami, S. Wilton, and L. Shannon, “Automatic high-quality verilog assertion generation through subtask-focused fine-tuned llms and iterative prompting,” 2024.
- [52] E. Tian, Y. Ci, Q. Yang, Y. Li, and Z. Lyu, “Assert-coder: Llm-based assertion generation via multimodal specification extraction,” 2025.
- [53] F. Wu, E. Pan, R. Kande, M. Quinn, A. Tyagi, D. K. Houngrinou, J. Rajendran, and J. Hu, “Spec2assertion: Automatic pre-rtl assertion generation using large language models with progressive regularization,” 2025.
- [54] H. Lyu, Y. Wang, Y. Du, M. Shi, Z. Chao, W. Li, T. Wang, and H. Li, “Assertgen: Enhancement of llm-aided assertion generation through cross-layer signal bridging,” 2025.
- [55] L. T. Fu, J. Zhou, S. Ren, M. Zhang, J. Xiong, H. Jiang, N. Guan, X. Wang, and J. Yang, “Chatsva: Bridging sva generation for hardware verification via task-specific llms,” *ArXiv*, vol. abs/2604.02811, 2026.
- [56] S. Paul, A. Banerjee, S. Ghosh, S. Surendran, and R. K. Gajavelly, “Supersaga: A supervisor-subordinate agentic workflow for the generation of assertions,” in *2026 31st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 420–425, 2026.
- [57] R. Guo, A. Ayalasomayajula, H. Li, J. Zhou, S. K. Saha, and F. Farahmandi, “Svagent: Ai agent for hardware security verification assertion,” 2025.
- [58] A. Kumar, D. N. Gadde, K. K. Radhakrishna, and D. Lettnin, “Saarthi: The first ai formal verification engineer,” 2025.
- [59] N. Mashnoor, H. Kamali, and K. Azar, “From language to logic: Bridging llms and formal representations for rtl assertion generation,” in *2026 IEEE 44th VLSI Test Symposium (VTS)*, pp. 1–5, 2026.
- [60] V. N. Viswambharan, K. K. Radhakrishna, D. N. Gadde, and A. Kumar, “Knowledge graphs, the missing link in agentic ai-based formal verification,” 2026.
- [61] F. Aditi and M. S. Hsiao, “Validatable generation of system verilog assertions from natural language specifications,” in *2023 Fifth International Conference on Transdisciplinary AI (TransAI)*, pp. 102–109, 2023.
- [62] A. Menon, S. Miftah, S. Kundu, S. Kundu, A. Srivastava, A. Raha, G. Sonnenschien, S. Banerjee, D. Mathaikutty, and K. Basu, “Enhancing large language models for hardware verification: A novel systemverilog assertion dataset,” *ACM Trans. Des. Autom. Electron. Syst.*, Aug. 2025. Just Accepted.
- [63] W. Xiao, D. Ekberg, S. Garg, and R. Karri, “Hybrid-rl2sva: Integrating rag and finetuning for llm-based rl2sva,” 2025.
- [64] V. Pulavarthi, D. Nandal, S. Dan, and D. Pal, “Are llms ready for practical adoption for assertion generation?,” 2025.
- [65] M. Kang, M. Liu, G. B. Hamad, S. Suhaib, and H. Ren, “Fveval: Understanding language model capabilities in formal verification of digital hardware,” 2024.
- [66] N. Pinckney, C. Deng, C.-T. Ho, Y.-D. Tsai, M. Liu, W. Zhou, B. Khailany, and H. Ren, “Comprehensive verilog design problems: A next-generation benchmark dataset for evaluating large language models and agents on rtl design and verification,” 2025.
- [67] Y. Wu, W. Fang, J. Wang, W. Li, Z. Guo, and Z. Xie, “Assertllm2: A comprehensive llm benchmark for assertion generation from design specifications,” 2026.
- [68] S. Vasudevan, D. Sheridan, S. Patel, D. Tcheng, B. Tuohy, and D. Johnson, “Goldmine: Automatic assertion generation using data mining and static analysis,” in *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, pp. 626–629, 2010.
- [69] A. Danese, N. D. Riva, and G. Pravadelli, “A-team: Automatic template-based assertion miner,” in *2017 54th ACM/EDAC/IEEE Design Automation Conference*

- (DAC), pp. 1–6, 2017.
- [70] S. Germiniani and G. Pravadelli, “Harm: A hint-based assertion miner,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 11, pp. 4277–4288, 2022.
 - [71] A. Madaan, N. Tandon, P. Gupta, S. Hallinan, L. Gao, S. Wiegrefe, U. Alon, N. Dziri, S. Prabhunoye, Y. Yang, S. Gupta, B. P. Majumder, K. Hermann, S. Welleck, A. Yazdanbakhsh, and P. Clark, “Self-refine: Iterative refinement with self-feedback,” 2023.
 - [72] J. R. Imperial and H. Zheng, “Specalign: A semantic alignment framework for systemverilog assertion generation,” 2026.
 - [73] IEEE, “IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language.” IEEE Std 1800-2023, 2023. Official standard landing page.
 - [74] Accellera Systems Initiative, “UVM 1.2 User’s Guide,” 2015.
 - [75] cocotb contributors, “cocotb Documentation,” 2026. Accessed 2026-06-16.
 - [76] R. Qiu, G. L. Zhang, R. Drechsler, U. Schlichtmann, and B. Li, “Autobench: Automatic testbench generation and evaluation using llms for hdl design,” in *Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD*, 2024.
 - [77] 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 *Design, Automation and Test in Europe Conference*, 2025.
 - [78] C.-C. Meng, Y.-R. Lu, G.-Y. Lin, T. T. Yeh, K.-C. Wu, and I.-C. Wu, “Haven: Hybrid automated verification engine for uvm testbench synthesis with llms,” *arXiv preprint arXiv:2604.27643*, 2026.
 - [79] J. Ye, D. Pan, H. Liu, Y. Hu, J. Zhou, K. Xu, X. Fang, X. Wang, N. Guan, and Z. Jiang, “Uvmarvel: an automated llm-aided uvm machine for subsystem-level rtl verification,” *arXiv preprint arXiv:2605.04704*, 2026.
 - [80] J. McEllin, R. Conway, and C. Ryan, “AVERT: An automatic verilog testbench generation tool for grammatical evolution,” in *2022 33rd Irish Signals and Systems Conference (ISSC)*, 2022.
 - [81] I. Ioannides and K. Eder, “Coverage-directed test generation automated by machine learning: A review,” *ACM Transactions on Design Automation of Electronic Systems*, vol. 17, no. 1, 2012.
 - [82] A. A. Jayasena and P. Mishra, “Directed test generation for hardware validation: A survey,” *ACM Computing Surveys*, vol. 56, no. 5, pp. 1–36, 2024.
 - [83] K. Laeuffer, J. Koenig, D. Kim, J. Bachrach, and K. Sen, “RFUZZ: Coverage-directed fuzz testing of RTL on FPGAs,” in *Proceedings of the 2018 IEEE/ACM International Conference on Computer-Aided Design*, 2018.
 - [84] S. Canakci, L. Delshadtehrani, F. Eris, M. B. Taylor, M. Egele, and A. Joshi, “DirectFuzz: Automated test generation for RTL designs using directed graybox fuzzing,” in *Proceedings of the 58th ACM/IEEE Design Automation Conference*, pp. 529–534, 2021.
 - [85] D. N. Gadde, A. Kumar, D. Lettnin, and S. Simon, “FuzzWiz: Fuzzing framework for efficient hardware coverage,” *arXiv preprint arXiv:2410.17732*, 2024.
 - [86] J. Bhandari, J. Knechtel, R. Narayanaswamy, S. Garg, and R. Karri, “LLM-aided testbench generation and bug detection for finite-state machines,” *arXiv preprint arXiv:2406.17132*, 2024.
 - [87] Z. Zhang, B. Szekely, P. Gimenes, G. Chadwick, H. McNally, J. Cheng, R. Mullins, and Y. Zhao, “LLM4DV: Using large language models for hardware test stimuli generation,” *arXiv preprint arXiv:2310.04535*, 2023.
 - [88] S. Lowe, E. Hilaneh, A. Babbitt, N. Gopalan, V. Chhabria, and A. Arora, “Spec2Cov: An agentic framework for code coverage closure of digital hardware designs,” *arXiv preprint arXiv:2604.15606*, 2026.
 - [89] Y. Zhao, Z. Wu, B. Yuan, Z. Yu, H. Zhang, W. Ni, C.-T. Ho, H. Ren, and J. Zhao, “PRO-V-R1: Reasoning enhanced programming agent for RTL verification,” *arXiv preprint arXiv:2506.12200*, 2025.
 - [90] J. Wang, Z. Yao, Y. Pi, X. Li, F. Song, J. Wang, Y. Xie, S. Wang, and Y. Bao, “UCAgent: An end-to-end agent for block-level functional verification,” *arXiv preprint arXiv:2603.25768*, 2026.
 - [91] R. Qiu, G. L. Zhang, R. Drechsler, T. Ho, U. Schlichtmann, and B. Li, “ConfiBench: Automatic testbench generation with confidence-based scenario mask and testbench ensemble using LLMs for HDL design,” *ACM Transactions on Design Automation of Electronic Systems*, 2025.
 - [92] N. S. Murthy, E. Nelson, S. S. Sapatnekar, and J. Sartori, “VerifLLMBench: An open-source benchmark for testbenches generated with large language models,” in *Design and Verification Conference (DVCon) U.S.*, 2025.
 - [93] J. Ye, Y. Hu, K. Xu, D. Pan, Q. Chen, J. Zhou, S. Zhao, X. Fang, X. Wang, N. Guan, and Z. Jiang, “From concept to practice: an automated llm-aided uvm machine for rtl verification,” *arXiv preprint arXiv:2504.19959*, 2025.
 - [94] D. Dranga, “Trials of using generative AI for APB UVM testbench generation,” *Romanian Journal of Information Technology and Automatic Control*, vol. 34, no. 2, pp. 75–84, 2024.
 - [95] W. Liu, F. Hou, J. Zhang, H. Liu, and A. Lei, “A multi-agent generative AI framework for IC module-level verification automation,” *arXiv preprint arXiv:2507.21694*, 2025.
 - [96] 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, and F. Farahmandi, “SV-LLM: An agentic approach for SoC security verification using large language models,” *arXiv preprint arXiv:2506.20415*, 2025.

- [97] M. Hassan, M. Nadeem, K. Qayyum, C. K. Jha, and R. Drechsler, "Prompt. verify. repeat. LLMs in the hardware verification cycle," in *2025 IEEE International Conference on Omni-layer Intelligent Systems*, 2025.
- [98] B. Nadimi, K. Filom, D. Chen, and H. Zheng, "TB or Not TB: Coverage-driven direct preference optimization for verilog stimulus generation," *arXiv preprint arXiv:2511.15767*, 2025.
- [99] H. Zhang, Z. Yu, C.-T. Ho, H. Ren, B. Khailany, and J. Zhao, "LLM4Cov: Execution-aware agentic learning for high-coverage testbench generation," *arXiv preprint arXiv:2602.16953*, 2026.
- [100] E.-M. Huang, Y.-H. Kao, R.-H. Deng, W.-P. Hsin, Y.-T. Hsieh, C. Liang, H.-Y. Tsou, M.-C. Chen, Y.-K. Hung, S.-C. Ho, P.-H. Huang, S.-H. Hung, and H. T. Kung, "Structured testbench generation for llm-driven hdl design and verification-oriented data curation," *arXiv preprint arXiv:2606.12983*, 2026.
- [101] N. Pinckney, C. Deng, C.-T. Ho, Y.-D. Tsai, M. Liu, W. Zhou, B. Khailany, and H. Ren, "Comprehensive verilog design problems: A next-generation benchmark dataset for evaluating large language models and agents on RTL design and verification," *arXiv preprint arXiv:2506.14074*, 2025.
- [102] D. Pal and S. Vasudevan, "Symptomatic bug localization for functional debug of hardware designs," in *Proc. International Conference on VLSI Design and International Conference on Embedded Systems (VLSID)*, pp. 517–522, 2016.
- [103] 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 Transactions on Design Automation of Electronic Systems*, vol. 30, no. 6, pp. 102:1–102:26, 2025.
- [104] H. Witharana, Y. Lyu, S. Charles, and P. Mishra, "A survey on assertion-based hardware verification," *ACM Computing Surveys*, vol. 54, no. 11s, pp. 1–33, 2022.
- [105] B. Ahmad, J. Ah-Kiow, B. Tan, R. Karri, and H. Pearce, "FLAG: Finding line anomalies (in RTL code) with generative AI," *ACM Transactions on Design Automation of Electronic Systems*, vol. 30, no. 6, pp. 103:1–103:30, 2025.
- [106] H. Lyu, Y. Du, Y. Wang, Z. Chao, T. Wang, and H. Li, "AssertFix: Empowering automated assertion fix via large language models," *arXiv preprint arXiv:2509.23972*, 2025. Version 1.
- [107] 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. Version 1.
- [108] Y. Tsai, M. Liu, and H. Ren, "RTLFixer: Automatically fixing RTL syntax errors with large language models," in *Proc. ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2024.
- [109] J. Zhou, Y. Ji, N. Wang, Y. Hu, X. Jiao, B. Yao, X. Fang, S. Zhao, N. Guan, and Z. Jiang, "Insights from rights and wrongs: A large language model for solving assertion failures in RTL design," in *Proc. ACM/IEEE Design Automation Conference (DAC)*, pp. 1–7, 2025.
- [110] 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. IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–9, 2024.
- [111] Z. Luo, F. Cui, K. Zhou, R. Guo, M. Xia, H. Hou, and Y. Liang, "R3A: Reliable RTL repair framework with multi-agent fault localization and stochastic tree-of-thoughts patch generation," *arXiv preprint arXiv:2511.20090*, 2025. Version 2.
- [112] B. Ahmad, S. Thakur, B. Tan, R. Karri, and H. Pearce, "On hardware security bug code fixes by prompting large language models," *IEEE Transactions on Information Forensics and Security*, vol. 19, pp. 4043–4057, 2024.
- [113] M. Akyash and H. M. Kamali, "Self-HWDebug: Automation of LLM self-instructing for hardware security verification," in *Proc. IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp. 391–396, 2024.
- [114] Y. Hu, J. Ye, K. Xu, J. Sun, S. Zhang, X. Jiao, D. Pan, J. Zhou, N. Wang, W. Shan, X. Fang, X. Wang, N. Guan, and Z. Jiang, "UVLLM: An automated universal RTL verification framework using LLMs," in *Proc. ACM/IEEE Design Automation Conference (DAC)*, pp. 1–7, 2025.
- [115] K. Qayyum, M. Hassan, S. Ahmadi-Pour, C. K. Jha, and R. Drechsler, "From bugs to fixes: HDL bug identification and patching using LLMs and RAG," in *Proc. IEEE LLM Aided Design Workshop (LAD)*, pp. 1–5, 2024.
- [116] Y. Bai and H. Ren, "Learning to debug: LLM-organized knowledge trees for solving RTL assertion failures," *arXiv preprint arXiv:2511.17833*, 2025. Version 2.
- [117] P. Zheng, J. Qin, F. Zhang, N. Chitla, Z. Wan, S. Wu, Y. Cao, C. Ding, and Y. Zhao, "HDLxGraph: Bridging large language models and HDL repositories via HDL graph databases," in *Proc. Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 533–540, 2026.
- [118] W. Lv, Y. Xia, X. Chen, and L. Kuang, "ViTAD: Timing violation-aware debugging of RTL code using large language models," *arXiv preprint arXiv:2508.13257*, 2025. Version 1.
- [119] W. Fu, K. Yang, R. G. Dutta, X. Guo, and G. Qu, "LLM4SecHW: Leveraging domain-specific large language model for hardware debugging," in *Proc. Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, pp. 1–6, 2023.
- [120] N. Wang, B. Yao, J. Zhou, Y. Hu, X. Wang, Z. Jiang, and N. Guan, "VeriDebug: A unified LLM for verilog debugging via contrastive embedding and guided correction," in *Proc. IEEE International Conference on*

- LLM-Aided Design (ICLAD)*, pp. 61–67, 2025.
- [121] B. Yao, N. Wang, X. Liu, Y. Du, Y. Hu, H. Gao, Z. Jiang, and N. Guan, “ARSP: Automated repair of verilog designs via semantic partitioning,” *arXiv preprint arXiv:2508.16517*, 2025. Version 1.
 - [122] Y. Wang, C. Liu, J. Zhang, L. Zhang, L. Cheng, X. Li, and H. Li, “VeriPilot: An LLM-powered verilog debugging framework,” *arXiv preprint arXiv:2606.23759*, 2026. Version 1.
 - [123] Y.-T. Liu and C. Yu, “VeriTrace: Human-like temporal exploration completes agentic action space,” in *Proc. IEEE International Conference on LLM-Aided Design (ICLAD)*, 2026.
 - [124] Z. Luo, Y. Xu, R. Guo, F. Cui, K. Zhou, M. Xia, H. Hou, Y. Luo, and Y. Liang, “Clover: A neural-symbolic agentic harness with stochastic tree-of-thoughts for verified RTL repair,” *arXiv preprint arXiv:2604.17288*, 2026. Version 1.
 - [125] L. Collini, B. Ahmad, J. Ah-Kiow, and R. Karri, “MARVEL: Multi-agent RTL vulnerability extraction using large language models,” *ACM Transactions on Design Automation of Electronic Systems*, 2026.
 - [126] 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,” in *Proc. Great Lakes Symposium on VLSI (GLSVLSI)*, pp. 673–680, 2025.
 - [127] Y. Tan, X. Meng, Z. Jiang, and Y. Lyu, “AutoVeriFix: Automatically correcting errors and enhancing functional correctness in LLM-generated verilog code,” in *Proc. Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 526–532, 2026.
 - [128] C.-T. Ho, H. Ren, and B. Khailany, “VerilogCoder: Autonomous verilog coding agents with graph-based planning and abstract syntax tree (AST)-based waveform tracing tool,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 39, pp. 300–307, 2025.
 - [129] X. Kong, T. Tabassum, M. Abdelwahab, and H. Zheng, “A progressive approach to synthesizable rtl design generation using llms,” *arXiv preprint arXiv:2607.18519*, 2026.
 - [130] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, “Bounded model checking,” *Advances in Computers*, vol. 58, pp. 117–148, 2003.
 - [131] K. Qayyum, M. Hassan, S. Ahmadi-Pour, C. K. Jha, and R. Drechsler, “Late-breaking results: Llm-assisted automated incremental proof generation for hardware verification,” in *Proceedings of the 63rd ACM/IEEE Design Automation Conference (DAC)*, (New York, NY, USA), pp. 1–2, Association for Computing Machinery, 2026.
 - [132] M. Hassan *et al.*, “Llm-guided formal verification coupled with mutation testing,” in *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*, 2024.
 - [133] Y. Su *et al.*, “Cill: Cti-guided invariant generation via llms for model checking,” *arXiv preprint arXiv:2602.23389*, 2026.
 - [134] M. Miao, G. Hu, Z. Yang, and H. Zhang, “Ic3-evolve: Proof-/witness-gated offline llm-driven heuristic evolution for ic3 hardware model checking,” *arXiv preprint arXiv:2604.03232*, 2026. arXiv.
 - [135] Y. Liu *et al.*, “Deeprtl2: A versatile model for rtl-related tasks,” in *Findings of the Association for Computational Linguistics (ACL)*, pp. 6485–6500, 2025.
 - [136] J. Jiang, J. Wang, Y. Yan, Y. Liu, and J. Zhu, “Formaleval,” *ACM Trans. Design Autom. Electron. Syst.*, vol. 30, no. 1, pp. 1–25, 2025.
 - [137] K. Li *et al.*, “Formalrtl: Verified rtl synthesis at scale,” *arXiv preprint arXiv:2603.08738*, 2026.
 - [138] Y. Su, Q. Yang, Y. Ci, Y. Li, T. Bu, and Z. Huang, “Deeply optimizing the sat solver for the ic3 algorithm,” in *Computer Aided Verification – 37th International Conference, CAV 2025, Proceedings, Part I*, vol. 15931 of *Lecture Notes in Computer Science*, pp. 237–257, Springer, 2025.
 - [139] Y. Su, Q. Yang, Y. Ci, T. Bu, and Z. Huang, “The ric3 hardware model checker,” in *Computer Aided Verification – 37th International Conference, CAV 2025, Proceedings, Part I*, vol. 15931 of *Lecture Notes in Computer Science*, pp. 167–177, Springer, 2025.
 - [140] Y. Sun, F. Ye, X. Zhang, S. Huang, B.-Z. Zhang, K. Wei, and S. Cai, “Autosat: Automatically optimize sat solvers via large language models,” *Journal of Artificial Intelligence Research*, vol. 76, 2026.
 - [141] J. Sheng, Y. Lin, J. Wu, Y. Huang, J. Shi, M. Zhang, and X. Wang, “Solsearch: An llm-driven framework for efficient sat-solving code generation,” in *Proceedings of the 47th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pp. 31–35, IEEE/ACM, 2025.
 - [142] Y. Sun, F. Ye, Z. Chen, K. Wei, and S. Cai, “Discovering heuristics in a complex sat solver with large language models,” *arXiv preprint arXiv:2507.22876*, 2025.
 - [143] T.-C. Hwang, F. Yu, and J.-H. R. Jiang, “Agentic taxation optimization via llm smt-constraint reasoning,” in *Proceedings of the IEEE/ACM 48th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP ’26)*, pp. 797–807, IEEE/ACM, 2026.