Agentic AI Meets Formal Verification in the Era of Agentic Hardware Development

Invited Paper

Wenji Fang[†], Yuchao Wu[†], Jing Wang, Zhiyao Xie[*]
Hong Kong University of Science and Technology

ABSTRACT

As AI agents become increasingly capable of generating and optimizing hardware circuits, ensuring the correctness of AI-generated designs becomes more important. Meanwhile, advances in AI lower the barriers to developing and applying formal methods, creating new opportunities to automate and improve hardware verification. This invited paper presents an overview of the methodologies and applications of hardware formal methods in the AI era. First, we examine how agentic AI advances hardware formal methods across three levels: specification, proof construction, and reasoning engines. Second, we discuss how formal methods support reliable AI-driven hardware development through RTL generation, debugging, and correctness-preserving optimization. We review representative research alongside our recent contributions, highlighting key technical advances and insights from experimental results. Finally, we discuss future research opportunities toward proof-native hardware development, where formal reasoning could become an integral part of autonomous hardware design.

1 INTRODUCTION

Recent advances in large language models (LLMs) and AI agents are reshaping hardware development. Designers can increasingly express requirements in natural language and rely on AI agents to generate, debug, and optimize RTL implementations. We refer to this emerging paradigm as *agentic hardware development*. As AI capabilities improve, agents can explore larger design spaces and perform increasingly complex design tasks with less human intervention. This trend creates new opportunities for more automated and efficient hardware development.

Growing AI capabilities increase the demand for formal verification while creating new opportunities to advance formal methods. As shown in Figure 1, AI agents can generate and modify more hardware designs, increasing the number of design decisions that require correctness checking. Formal methods provide rigorous guarantees that implementations satisfy their formal specifications. Meanwhile, developing and applying formal methods traditionally requires substantial expertise in specification, proof construction, and algorithm design. Advances in AI can reduce these barriers by automating verification tasks, guiding formal reasoning, and accelerating the development of verification algorithms.

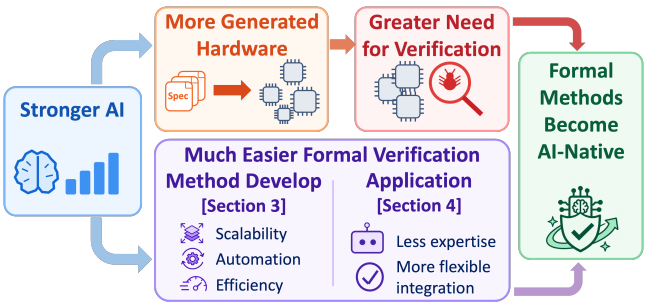


Figure 1: Advances in AI make hardware generation and optimization more accessible, increasing the demand for formal verification. Meanwhile, AI lowers the barriers to developing and applying formal methods, creating opportunities to integrate formal reasoning into agentic hardware development.

This invited paper reviews the methodology and applications of hardware formal methods in the AI era through two perspectives: (1) *Part I (Methodology): Agentic AI for Hardware Formal Methods* examines how AI advances formal verification methodologies, improving their automation, efficiency, and scalability. (2) *Part II (Application): Formal Methods for Agentic Hardware Development* examines how formal verification supports reliable AI-driven hardware generation, debugging, and optimization. Together, these parts provide a systematic view of recent progress and emerging opportunities for hardware formal methods.

Part I (Methodology) organizes recent advances in agentic AI for hardware formal methods into three levels: specification, proof construction, and reasoning engines. (1) *Specification:* AI supports assertion generation, specification quality evaluation, and the development of executable behavioral specifications. (2) *Proof construction:* AI agents guide model checking and automate theorem proving through semantic guidance, lemma generation, and proof reuse. (3) *Reasoning engines:* AI agents discover and optimize reusable algorithms and heuristics for SAT solving and model checking. These levels cover the development of correctness specifications, the construction of proofs, and the improvement of underlying reasoning algorithms.

Part II (Application) examines three applications of formal methods in agentic hardware development: generation, debugging, and optimization. (1) *Generation:* Formal properties, executable models, and proof feedback guide RTL construction and correction. (2) *Debugging:* Formal counterexamples and verification results support root-cause diagnosis and targeted repair. (3)

[†]Equal Contribution
[*]Corresponding Author (eezhiyao@ust.hk)

This work is licensed under a Creative Commons Attribution 4.0 International License.
ASPDAC '27, January 25–28, 2027, Tokyo, Japan
© 2027 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2923-2/2027/01
https://doi.org/10.1145/3840404.3860940

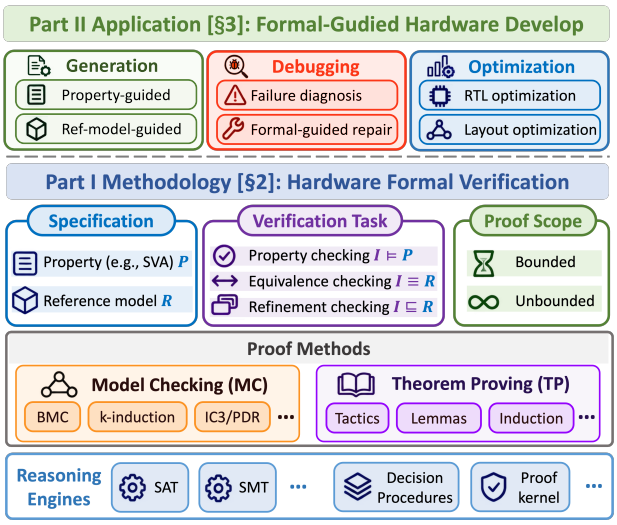


Figure 2: Overview of our taxonomy of hardware formal verification methodology and application.

Optimization: Formal equivalence checking enables correctness-preserving transformations guided by PPA feedback. These applications demonstrate how formal verification can provide continuous correctness feedback throughout AI-driven hardware development.

We present a systematic review of recent advances, supported by representative research, experimental results, and our own contributions. Our main contributions are summarized as follows:

- **Methodology: Agentic AI for Hardware Formal Methods.** We present a taxonomy of AI-driven advances in formal specification, proof construction, and reasoning engines, highlighting their technical progress and challenges.
- **Application: Formal Methods for Agentic Hardware Development.** We organize formal-guided approaches to RTL generation, debugging, and optimization, examining their roles in improving hardware design reliability.
- **Research Progress and Insights.** We review representative studies and our recent contributions, using experimental results and case studies to identify key advances and remaining challenges.
- **Future Research Directions.** We discuss opportunities in intent-aligned specifications, scalable and reusable proving, and proof-native hardware development.

We envision formal methods becoming a native part of AI-driven hardware development. Advances in requirements engineering can support more complete and intent-aligned formal specifications, while AI-native verification techniques can improve proof automation, efficiency, and scalability. Integrating these capabilities into hardware generation, debugging, and optimization can enable proof-native development, where AI agents make increasingly autonomous design decisions supported by machine-checkable correctness guarantees.

Table 1: Our AssertLLM2 [41] benchmark statistics and evaluation metrics.

Statistic	Value	Eval. Dimension	Metric
Design Scale	83	Syntax Check	Compile Pass
Func. Categories	13	Formal Provability	FPV Proven Rate
Avg. RTL Scale	2.9k LoC	Coverage	COI/Formal/Proof
Avg. Spec Scale	10k tokens	Mutation Testing	Mutant Kill Rate

2 PART I (METHODOLOGY): AGENTIC AI FOR HARDWARE FORMAL METHODS

Hardware formal verification involves three closely related levels: specifying the intended behavior, constructing proofs against the specification, and executing the underlying reasoning algorithms. Each level presents a distinct challenge and opportunity for agentic AI. As illustrated in Figure 3, (1) *Formal specification* (Section 2.1) defines what needs to be verified, with AI helping generate and validate correctness targets aligned with designer intent; (2) *Proof construction* (Section 2.2) determines how to establish correctness for a given verification task, with AI guiding model checking and theorem proving through semantic hints, invariants, and lemmas; and (3) *Reasoning engines* (Section 2.3) provide the underlying computational capabilities, with AI discovering reusable algorithms and heuristics to improve solving efficiency. These three levels connect the correctness target, task-specific proof strategies, and general-purpose reasoning algorithms. Table 2 summarizes representative research in these directions.

2.1 Formal Specification: Capturing Design Intent

AI-assisted hardware specification is evolving from automated assertion generation toward more complete and reliable formal specifications. A central challenge is to establish proof targets that accurately capture designer intent and provide sufficient behavioral coverage. We discuss two complementary directions: property-based and reference model-based specification.

2.1.1 Property-based Specification. Property-based specification expresses functional and temporal requirements through formal assertions. Recent research has progressed from automated assertion generation toward improving the quality, coverage, and bug-detection effectiveness of generated properties.

Traditional Background. Property-based verification expresses functional and temporal requirements through assertions, such as SystemVerilog Assertions (SVA) [15, 17]. As illustrated in Figure 2, formal property verification (FPV) checks whether an implementation I satisfies a property P, written as $I \models P$. Traditionally, engineers manually translate design requirements into assertions, requiring expertise in hardware behavior and temporal logic. The effectiveness of verification depends on the correctness and behavioral coverage of these properties.

AI-based Methods. Early research explored automating assertion generation from natural-language specifications. Our AssertLLM framework was among the early efforts to generate SVA from complete specification documents, extracting functional and

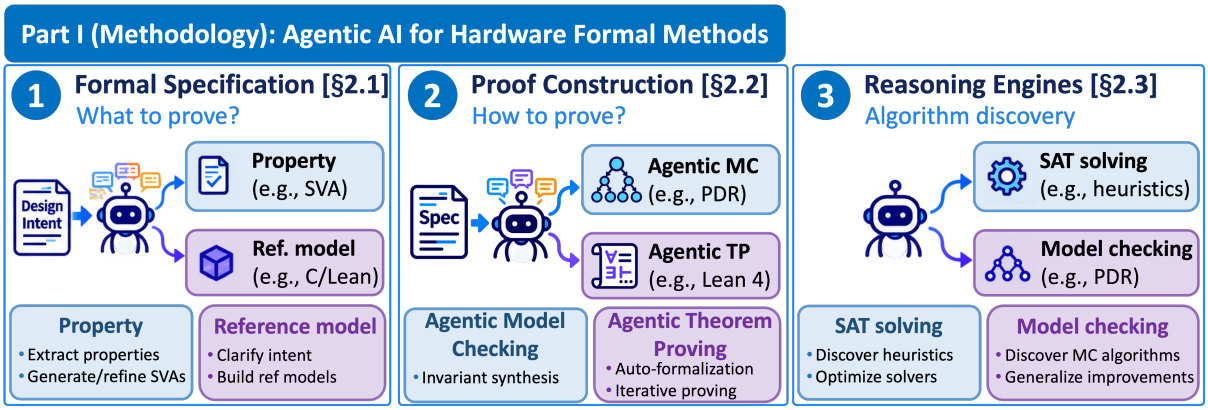


Figure 3: Three levels of agentic AI for hardware formal methods. AI supports specification generation and intent clarification, guides task-specific proof construction through model checking and theorem proving, and improves reusable reasoning algorithms through agentic discovery.

temporal requirements from text and waveform diagrams [45]. Recent approaches further improve generation through learned grammars, retrieval, specialized language models, and RTL-aware specification analysis [3, 14, 17, 31, 42].

As assertion generation becomes increasingly automated, evaluating the quality of generated properties becomes a central challenge. Syntactic correctness and formal provability alone provide limited evidence of verification effectiveness. Coverage and bug-detection capability offer additional measures of assertion quality. Coverage-guided approaches such as LASSO [1] and CoverAssert [40] use verification feedback to expand property coverage. Benchmarks including FVEval [21] and AssertionBench support systematic evaluation of generated assertions [32].

Our Work and Insights. Our AssertLLM2 [41] benchmark evaluates assertion quality across 83 designs through syntax checking, formal verification, coverage analysis, and mutation testing. Table 1 summarizes its statistics and evaluation metrics. Claude Sonnet 4.5 achieves 93.04% syntax success and 65.65% structural reach, but only 23.30% proof coverage and an 11.69% bug-kill ratio. These results reveal a substantial gap between generating valid assertions and achieving effective verification. They motivate coverage-driven generation and evaluation methods that measure how well properties detect incorrect hardware behavior.

2.1.2 Reference Model-based Specification. Reference model-based specification describes expected hardware behavior through executable models. It provides a broader correctness target for equivalence and refinement checking, complementing the selected behaviors expressed by individual assertions.

Traditional Background. Reference model-based verification describes expected hardware behavior through executable models, such as RTL, C/C++, or Lean. As illustrated in Figure 2, formal equivalence verification (FEV), including combinational equivalence checking (CEC) and sequential equivalence checking (SEC), establishes behavioral equivalence between an implementation I and a reference R, written as $I \equiv R$ [23, 28, 38]. Refinement checking establishes that an implementation conforms to an abstract reference model, written as $I \sqsubseteq R$ [6, 11, 43]. These approaches provide broader behavioral correctness targets, but constructing reliable reference models requires accurately defining state transitions, interfaces, timing, and functional semantics.

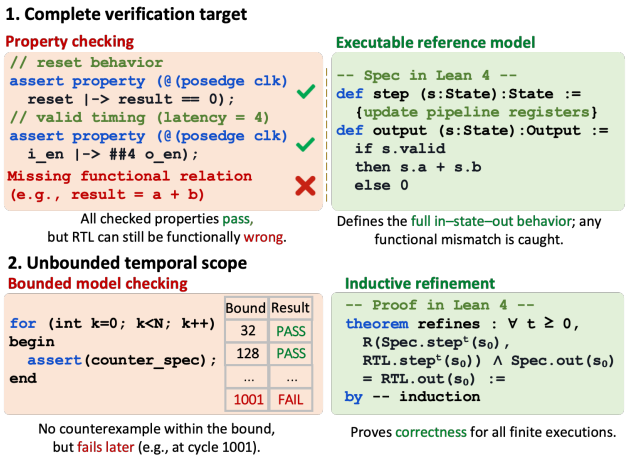


Figure 4: Two key dimensions of hardware formal verification: specification completeness and proof completeness. Executable reference models can capture broader intended behavior, while unbounded proofs establish correctness across all execution depths.

AI-based Methods. Recent advances in LLMs create opportunities to support the construction, validation, and use of executable hardware specifications. FormalRTL uses a supplied C reference model to guide RTL generation and correction through equivalence checking [22]. HINT introduces an executable hardware-intent representation with specification-level checking and correctness obligations for RTL lowering [5]. These approaches demonstrate how executable specifications can provide explicit behavioral targets for AI-driven hardware development.

Results and Insights. As illustrated in Figure 4(1), individual assertions may leave functional relationships unchecked even when all specified properties pass. Executable reference models can describe broader input–state–output behavior and support equivalence or refinement checking. Their effectiveness depends on accurately capturing designer intent, including state transitions, timing, and interface behavior. Automating the construction and validation of reliable formal executable specifications remains an important research opportunity.

Table 2: Representative research in agentic AI for hardware formal methods.

Category	Direction	Representative Work	Key Contribution
Formal Specification	Property-based	AssertLLM [45]	Automates SVA generation from specification documents and waveform.
		AssertLLM2 [41]	Evaluates assertion quality via formal verification, coverage, and mutation testing.
		AssertionForge [3]	Combines specification and RTL context for assertion generation.
		LASSO [1]	Uses formal feedback to improve assertion coverage.
		FVEval [21]	Benchmarks assertion generation using syntax and formal verification.
	Reference model-based	FormalRTL [22]	Uses a supplied C reference model to guide RTL generation and equivalence-based correction.
		HINT [5]	Introduces an executable hardware-intent representation with specification-level checking and explicit RTL-lowering obligations.
Proof Construction	Agentic MC	LLM4PDR [18]	Generates semantic predicates and clauses to guide word-level PDR.
		CIll [34]	Iteratively generates supporting invariants from counterexamples to induction.
		Agentic-IC3 [10]	Uses evolving proof states and solver feedback to guide IC3 proof search.
	Agentic TP	CktFormalizer [44]	Combines typed circuit representations with interactive equivalence proving.
		Rtl2lean [26]	Translates RTL into Lean models and constructs correctness theorems.
		CircuitProver [46]	Automates hardware property proving through compositional reasoning and reusable Lean proofs.
Reasoning Engines	SAT solving	AutoSAT; AutoModSAT [35, 36]	Discovers and optimizes SAT solver heuristics through LLM-driven search.
		AESAT [25]	Combines LLM-guided heuristic optimization with evolutionary search.
		SATLUTION [47]	Extends algorithm discovery to repository-level SAT solver code evolution.
	Model checking	IC3-Evolve [27]	Discovers reusable IC3/PDR heuristics through offline code evolution and correctness validation.

2.2 Proof Construction: Agent-driven Formal Reasoning

Agent-driven formal reasoning aims to improve the ***automation, efficiency, and scalability*** of hardware formal verification. AI agents use design semantics and verification feedback to guide proof search, construct supporting lemmas, and reuse verified knowledge. We discuss two complementary directions: agentic model checking and agentic theorem proving.

2.2.1 Agentic Model Checking. Agentic model checking uses high-level hardware semantics and verification feedback to improve the efficiency of traditional model checking algorithms.

Traditional Background. Model checking (MC) automatically reasons about hardware state spaces to verify correctness properties. As illustrated in Figure 2, verification can cover bounded or unbounded executions. Bounded model checking (BMC) searches for counterexamples within a finite depth using SAT/SMT solving [4, 30]. Unbounded techniques, including k-induction and IC3/PDR, establish correctness over all reachable states through inductive reasoning [9, 16, 33, 48]. These methods provide a high degree of automation, but their efficiency often depends on discovering useful invariants and managing large state spaces.

AI-based Methods. Recent approaches introduce LLM-generated semantic guidance through two strategies. (1) *Pre-search guidance:* LLM4PDR generates predicates, clauses, and helper assertions to guide word-level PDR, with candidate hints validated by the underlying model checker [18]. (2) *Feedback-driven guidance:* CIll iteratively generates supporting invariants from counterexamples to induction (CTIs), while Agentic-IC3 interacts with evolving proof states to propose semantic lemmas and guide clause generalization [10, 34]. These approaches allow model checking to incorporate design-specific semantic knowledge during proof search.

Results and Insights. Recent studies report improved verification performance on challenging arithmetic and processor designs using LLM-generated semantic guidance [10, 18, 34]. High-level RTL semantics can help identify useful invariants and functional relationships that are difficult to discover through conventional search alone. Pre-generated hints introduce limited inference overhead, while interactive agents can adapt to evolving proof states. Balancing semantic guidance quality, proof-search efficiency, and inference cost remains an important challenge for larger designs.

2.2.2 Agentic Theorem Proving. Agentic theorem proving automates machine-checked correctness proofs through proof planning, lemma generation, and compositional reasoning. An emerging direction is to accumulate and reuse verified proof knowledge across hardware verification tasks.

Traditional Background. Theorem proving (TP) constructs machine-checked correctness proofs using proof assistants such as Coq and Lean. As illustrated in Figure 2, theorem proving supports mathematical induction, expressive specifications, and compositional reasoning [6, 19]. For example, Kami uses Coq to establish modular correctness properties for parameterized hardware designs [6]. Compared with automated state-space exploration, theorem proving allows explicit reasoning over abstract state relations, parameterized designs, and reusable lemmas. However, traditional theorem proving requires substantial human expertise in formal modeling, proof decomposition, and interactive proof construction.

Agentic Methods. Recent approaches use LLM agents to automate hardware formalization and proof construction. CktFormalizer combines typed circuit representations with interactive theorem proving, while Rtl2lean translates RTL into Lean models and constructs hierarchical correctness theorems [26, 44]. These approaches use proof-assistant feedback to guide lemma generation, proof decomposition, and iterative repair. They also create opportunities for compositional proving and reusable proof libraries.

Our Work. CircuitProver [46] combines hardware-aware proof planning with a reusable Lean library for circuit property verification. The agent generates supporting lemmas, repairs failed proofs using Lean feedback, and reuses verified theorems and proof strategies. Compositional reasoning enables the construction of higher-level correctness proofs from verified component properties.

Results and Insights. CircuitProver [46] completes all 63 evaluated tasks, compared with 58 for its baseline, while reducing average proof rounds from 9.2 to 4.6. These results demonstrate the benefits of structured proof construction and reusable proof knowledge. As hardware complexity increases, automatically discovering inductive invariants, intermediate lemmas, and compositional proof structures becomes increasingly important. A promising direction is to develop agentic proving systems that continuously build and reuse machine-checked hardware proof libraries.

2.3 Reasoning Engines: Agentic Algorithm Discovery

Developing efficient reasoning engines traditionally requires substantial expertise in algorithm design, implementation, and optimization. Recent advances in LLM coding agents enable automated algorithm discovery through iterative code modification, benchmarking, and correctness validation. We discuss two directions: SAT solving and model checking, highlighting the trend from heuristic optimization toward autonomous algorithm development.

2.3.1 SAT Solving. Agentic SAT solving is evolving from automated heuristic tuning toward broader algorithm discovery and repository-level code evolution.

Traditional Background. SAT solving provides a fundamental reasoning capability for hardware formal verification, including bounded model checking, equivalence checking, and inductive proof procedures. As illustrated in Figure 2, SAT/SMT solvers discharge logical constraints generated by verification algorithms [29, 30]. Modern SAT solvers commonly use conflict-driven clause learning (CDCL), supported by heuristics for variable branching, restarts, and clause management. These heuristics strongly affect solving performance and traditionally require extensive expert knowledge and manual optimization.

Agentic Methods. Recent approaches use LLMs to discover and optimize SAT solver heuristics through iterative code generation and evaluation. AutoSAT explores heuristic generation within a modular CDCL solver, while AutoModSAT and AESAT combine LLM-guided optimization with evolutionary search [25, 35, 36]. Beyond individual heuristic functions, SATLUTION uses planning and coding agents to evolve larger solver codebases, with distributed benchmarking and accumulated optimization knowledge guiding subsequent iterations [47]. These methods expand the search space from predefined parameters to algorithmic implementations and interactions among solver components.

Results and Insights. LLM-driven algorithm discovery has produced competitive SAT solvers, including AESAT's AE-Kissat-MAB, which won the 2025 SAT Competition main track [25]. AutoSAT, AutoModSAT, and SATLUTION also report improvements over their evaluated baselines [35, 36, 47]. These results suggest that *agentic code evolution can explore algorithmic choices beyond manually designed search spaces*. Repository-level evolution further enables the joint optimization of interacting solver components. An important research challenge is to discover improvements that generalize across diverse SAT instances while maintaining solver correctness.

2.3.2 Model Checking. Agentic algorithm discovery is extending from SAT solving to model checking, where verification performance depends on more complex state-space reasoning and search strategies.

Traditional Background. Model checking engines combine symbolic state-space algorithms with SAT/SMT solving to discharge verification obligations. As illustrated in Figure 2, their reasoning capabilities support automated correctness checking over bounded or unbounded executions. Algorithms such as IC3/PDR rely on several interacting heuristic decisions, including inductive generalization, proof-obligation scheduling, and clause propagation [9, 16]. These choices strongly influence verification efficiency and traditionally require substantial expertise in algorithm design and optimization.

Agentic Methods. Recent research explores offline LLM-driven evolution of model checking algorithms. IC3-Evolve [27] generates and evaluates code modifications to IC3/PDR heuristics through iterative experimentation. Candidate implementations are evaluated on hardware benchmarks, with independently checked proof certificates and replayable counterexamples validating their results. The discovered heuristics are integrated into a standalone model checker without runtime LLM inference.

Results and Insights. IC3-Evolve [27] demonstrates the potential of automated algorithm discovery for hardware model checking. Compared with task-specific proof guidance, offline algorithm discovery produces reusable improvements that can benefit future verification tasks. Current exploration mainly focuses on selected heuristic components, leaving opportunities for broader algorithm discovery in state-space abstraction, invariant generalization, and interactions with SAT/SMT solvers. *Correctness validation, search efficiency, and cross-design generalization* are key challenges for developing reliable self-improving model checkers.

3 PART II: FORMAL METHODS FOR AGENTIC HARDWARE DEVELOPMENT

Agentic hardware development involves three closely related activities: generating hardware implementations, debugging functional errors, and optimizing designs for better power, performance, and area (PPA). Each activity requires formal verification in a different role. As illustrated in Figure 5, (1) *Generation* (Section 3.1) uses formal specifications and verification feedback to guide RTL construction and correction; (2) *Debugging* (Section 3.2) uses counterexamples and formal reasoning to identify root causes and validate repairs; and (3) *Optimization* (Section 3.3) uses equivalence checking to explore alternative implementations while preserving

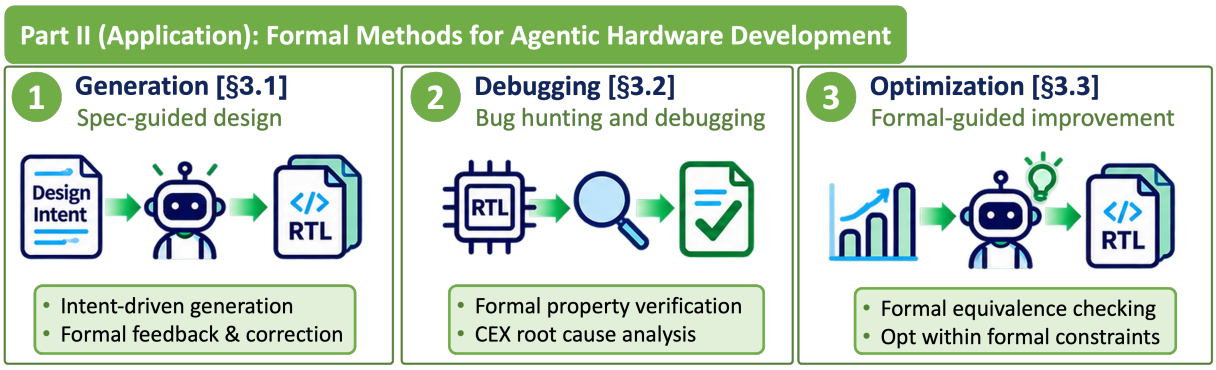


Figure 5: Three applications of formal methods in agentic hardware development. Formal verification guides RTL generation through correctness feedback, supports debugging through counterexample-based diagnosis and repair, and enables correctness-preserving design optimization.

functional correctness. Together, these activities form an iterative design process in which formal verification provides continuous correctness feedback and constraints for agentic decision-making. Table 3 summarizes representative research in these directions.

3.1 Generation: Formal-guided RTL Construction

Formal-guided RTL generation is evolving from one-shot code generation toward iterative construction and correction using verification feedback. Assertion properties, executable reference models, and verified intermediate representations provide complementary ways to establish correctness targets.

Traditional Background. Traditional RTL development relies on engineers to translate design requirements into implementations and identify functional errors through simulation and verification. LLMs automate RTL construction, increasing the need for reliable correctness feedback and automated correction.

Formal-guided Methods. Recent approaches integrate formal verification into RTL generation through two directions. (1) *Property-guided generation:* Formal properties provide correctness feedback for training data selection, error detection, and RTL correction. OpenLLM-RTL uses formal verification to select training examples [24]. (2) *Formal-model-guided generation:* Executable reference models and formally defined intermediate representations provide behavioral constraints and proof obligations for hardware generation. FormalRTL uses a supplied C reference model to guide RTL generation and correction through equivalence checking [22]. CktFormalizer uses typed circuit representations and interactive theorem proving to support hardware generation [44]. HINT introduces an executable hardware-intent representation with pre-RTL checking and correctness obligations for RTL lowering [5]. Proof2Silicon uses formally verified Dafny programs to guide hardware generation [20].

Our Work. CorrectRTL [39] introduces an SVA-based self-correction framework for LLM-generated RTL. As illustrated in Figure 6, it uses automatically generated assertions to provide formal feedback for RTL correction. The verification results identify violated requirements, and an LLM agent iteratively repairs the implementation. This reduces dependence on manually constructed testbenches during the correction process.

Results and Insights. CorrectRTL improves Pass@1 from 20.5% to 49.0% on the 200-design ScaleBench benchmark, with correctness

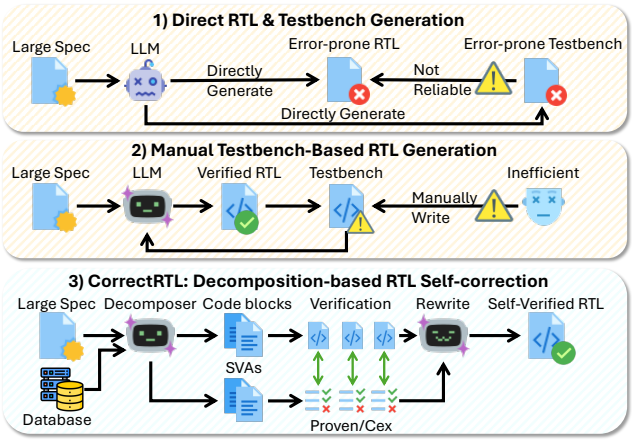


Figure 6: Overview of our CorrectRTL [39] and its assertion-guided RTL self-correction workflow.

independently evaluated through sequential equivalence checking [39]. These results demonstrate how formal feedback can improve RTL generation through iterative correction. Property-guided approaches provide targeted feedback on specified behaviors, while executable models and formal intermediate representations support more structured hardware generation. The effectiveness of these methods depends on the reliability of their formal specifications and the correctness of the transformations connecting them to RTL implementations.

3.2 Debugging: Formal-guided Diagnosis and Repair

Formal-guided debugging is progressing from interpreting verification failures toward automated root-cause diagnosis and formally validated repair. Counterexamples and assertion failures provide concrete evidence for guiding agent decisions.

Traditional Background. Hardware debugging traditionally requires engineers to analyze waveforms, simulation traces, and formal counterexamples to identify functional errors. Root-cause localization and repair are particularly challenging for sequential designs, where failures may depend on interactions across multiple cycles.

Formal-guided Methods. Recent approaches explore two complementary capabilities. (1) *Failure diagnosis:* AI agents analyze

Table 3: Representative research in formal methods for agentic hardware development.

Category	Direction	Representative Work	Key Contribution
Generation	Property-guided generation	OpenLLM-RTL [24]	Uses formal verification to select training examples for RTL generation.
		CorrectRTL [39]	Uses generated assertions and formal feedback for iterative RTL correction.
	Formal-model-guided generation	FormalRTL [22]	Uses C reference models for equivalence-guided RTL generation and correction.
		CktFormalizer [44]	Uses typed circuit representations and theorem proving for hardware generation.
		HINT [5]	Introduces an executable hardware-intent representation with pre-RTL checking and RTL-lowering obligations.
		Proof2Silicon [20]	Uses verifier-guided learning and formally verified intermediate programs for hardware generation.
Debugging	Failure diagnosis	FVDebug [2]	Uses formal counterexamples and causal reasoning for root-cause analysis.
	Formal-guided repair	AssertSolver [50]	Uses assertion failures and repair feedback to guide RTL correction.
		Stitch [51]	Uses counterexample-guided RTL repair with formal validation.
		Tran et al. [37]	Combines open-source model checking with counterexample-guided RTL repair.
		Das et al. [8]	Uses assertion-guided repair and JasperGold to validate multi-line RTL fixes.
Optimization	RTL-level optimization	ROVER [7]	Explores equivalent logic transformations through verified e-graph rewriting.
		ASPEN [49]	Combines LLM rewrite rules with equivalence checking and PPA feedback.
		Dr. RTL [13]	Uses timing-guided agentic RTL optimization with sequential equivalence checking.
	Layout-level optimization	E-Layout [12]	Enables equivalence-preserving layout logic restructuring using physical feedback.

formal counterexamples, design dependencies, and specifications to identify likely root causes. FVDebug combines counterexample analysis with causal reasoning and RTL context to explain formal verification failures and suggest fixes [2]. (2) *Formal-guided repair:* Agents use failing properties and counterexamples to propose RTL modifications, followed by formal rechecking. AssertSolver and Stitch explore assertion- and counterexample-guided correction [50, 51]. An open-source multi-agent workflow [37] further combines SymbiYosys and Z3 with counterexample-driven RTL repair. Recent work [8] on RISC-V processors uses assertion-guided LLM repair and JasperGold verification to address multi-line RTL bugs.

Results and Insights. These studies show how formal verification can support both fault localization and automated correction. Counterexamples reveal failing behavior, while repeated model checking evaluates whether proposed repairs satisfy the specified properties. Recent work also extends formal-guided repair toward more complex, multi-line bugs. The main challenges include accurate root-cause diagnosis, efficient repair search, and sufficient specification coverage to prevent incomplete or misleading fixes.

3.3 Optimization: Correctness-preserving Design Exploration

Formal equivalence checking enables AI agents to explore alternative implementations under explicit correctness constraints. Recent approaches combine formal checking with EDA feedback to optimize hardware at both RTL and physical-design stages.

Traditional Background. Traditional synthesis and RTL optimization use Boolean rewriting, restructuring, and technology mapping to improve power, performance, and area (PPA). Formal equivalence checking validates correctness-preserving transformations. At later design stages, placement and timing information provide additional guidance for selecting effective logic transformations.

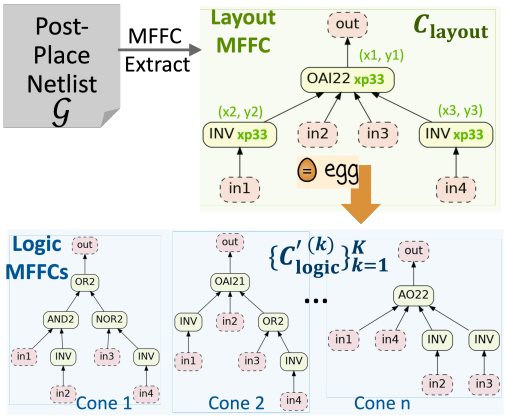


Figure 7: Our E-Layout [12] uses e-graph equality saturation to explore formal equivalent logic structures for timing optimization.

Formal-guided Methods. Recent approaches explore two directions. (1) *Verified logic rewriting:* ROVER uses verified e-graph rewriting, while ASPEN combines LLM-generated rewrite rules with equivalence checking and PPA feedback [7, 49]. (2) *Agentic design exploration:* AI agents iteratively generate design alternatives, evaluate their implementation quality using EDA tools, and validate functional equivalence. This supports design-specific optimization across a broader range of transformations.

Our Work. Our research explores correctness-preserving optimization at two design stages. (1) *RTL-level optimization:* Dr. RTL [13] uses timing reports and EDA feedback to guide agentic RTL transformations, with sequential equivalence checking (SEC) validating functional correctness. (2) *Layout-level optimization:* E-Layout [12]

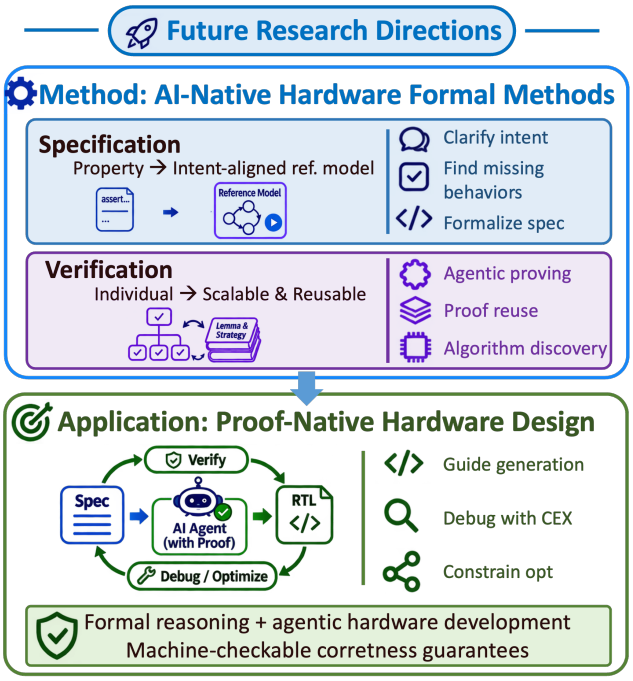


Figure 8: Future research directions toward scalable AI-native formal methods and proof-native hardware development.

explores post-placement logic restructuring through equality saturation and layout-aware structure selection. As illustrated in Figure 7, e-graph rewriting generates multiple equivalent implementations of a logic cone, expanding the structural optimization space. Learning-based selection uses placement and timing information to identify promising alternatives. Local SMT checking and full-design equivalence verification validate the transformations.

Results and Insights. Across 20 designs, Dr. RTL achieves average improvements of 21.3% in worst negative slack (WNS) and 16.9% in total negative slack (TNS), together with a 5.8% area reduction and an 86% SEC pass rate [13]. E-Layout reports average improvements of 6.07% in TNS and 2.29% in WNS, together with a 1.07% area reduction across nine designs [12]. These results highlight how formal equivalence enables broader structural exploration, while physical feedback guides the selection of implementations with better PPA.

4 FUTURE RESEARCH DIRECTIONS

As AI becomes more capable, formal methods are becoming both more important and more accessible. As illustrated in Figure 8, we envision three research directions spanning formal verification methodology and its applications toward proof-native hardware development.

4.1 Specification: From Isolated Properties to Intent-Aligned Reference Models

As AI increasingly automates RTL generation, accurately capturing designer intent becomes more important. A promising direction is to move from individual assertion properties toward more complete, executable formal reference models that support equivalence and refinement checking. This shift highlights the growing importance of requirements engineering in hardware development. AI agents can help designers clarify ambiguous requirements, identify missing behaviors, and translate design intent into formal specifications. Ensuring that these specifications accurately reflect designer intent will be essential for reliable AI-native hardware development.

4.2 Verification: From Individual Proofs to Scalable and Reusable Proving

AI-native verification techniques offer opportunities to improve proof automation, efficiency, and scalability. Agentic theorem proving can automate complex hardware proofs through invariant generation, proof decomposition, and interactive reasoning. Compositional verification and proof reuse can further improve scalability by allowing agents to accumulate and reuse machine-checked lemmas and proof strategies across related designs. Agentic algorithm discovery can also improve SAT/SMT solvers and model checking engines, making formal reasoning more efficient.

4.3 Application: From Formal Verification Feedback to Proof-Native Design

Formal verification is becoming an important part of AI-driven hardware generation, debugging, and optimization. The next step is to integrate formal reasoning throughout the design process, maintaining correctness guarantees as AI agents continuously modify implementations. Formal specifications can guide generation, counterexamples can support debugging, and equivalence checking can enable broader design exploration. **Proof-native hardware development** connects these capabilities into a unified workflow, where AI agents generate, verify, and optimize hardware while maintaining machine-checkable correctness guarantees.

5 CONCLUSION

This paper presented a unified view of the growing relationship between AI and formal methods in the era of agentic hardware development. We reviewed how AI advances formal specification, proof construction, and reasoning engines, and how formal methods support reliable RTL generation, debugging, and optimization. Through representative research and our recent contributions, we highlighted the opportunities for improving verification automation, efficiency, and scalability. As AI becomes more capable, the demand for correctness guarantees continues to grow, while the barriers to developing and applying formal methods are decreasing. We envision formal methods becoming a native part of AI-driven hardware development, enabling increasingly autonomous design with machine-checkable correctness guarantees.

ACKNOWLEDGMENTS

This work is supported by Hong Kong Research Grants Council (RGC) CRF-YCRG C6003-24Y, 16216825, T46-415/25-R, and JLFS-YSF/E-605/26. It was partially conducted by ACCESS – AI Chip Center for Emerging Smart Systems, supported by the InnoHK initiative of the Innovation and Technology Commission of the Hong Kong Special Administrative Region Government.

REFERENCES

[1] Dinesh Reddy Ankireddy, Sudipta Paria, Aritra Dasgupta, Sandip Ray, and Swarup Bhunia. 2025. LASSO: LLM-Aided Security Property Generation for Assertion-based SoC Verification. In *MLCAD*.

[2] Yunsheng Bai, Ghaith Bany Hamad, Chia-Tung Ho, Syed Suhaib, and Haoxing Ren. 2025. FVDebug: An LLM-Driven Debugging Assistant for Automated Root Cause Analysis of Formal Verification Failures. arXiv:2510.15906.

[3] Yunsheng Bai, Ghaith Bany Hamad, Syed Suhaib, and Haoxing Ren. 2025. AssertionForge: Enhancing Formal Verification Assertion Generation with Structured Representation of Specifications and RTL. In *LAD*.

[4] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. 1999. Symbolic model checking without BDDs. In *International conference on tools and algorithms for the construction and analysis of systems*. Springer, 193–207.

[5] Tairan Cheng, Yi Liu, Dongsheng Zuo, Zhengyuan Shi, Hongji Zhang, Xiangfei Hu, Maoshuo He, Hao Yan, and Qiang Xu. 2026. HINT: Toward an Executable Hardware-Intent Representation Layer for LLM-Driven RTL Generation. *arXiv preprint arXiv:2608.07625* (2026).

[6] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. 2017. Kami: a platform for high-level parametric hardware specification and its modular verification. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 1–30.

[7] Samuel Coward, Theo Drane, and George A. Constantinides. 2024. ROVER: RTL Optimization via Verified E-Graph Rewriting. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* 43, 12 (2024), 4687–4700.

[8] Debleena Das, Chandan Kumar Jha, Muhammad Hassan, Rolf Drechsler, Ansuman Banerjee, Subhra K. Das, and Amlan Chakrabarti. 2026. Automated Multi-Line Bug Repair Using Model Checker Aided LLMs: A Case Study on RISC-V Processors. *arXiv preprint arXiv:2610.32597* (2026).

[9] Niklas Eén, Alan Mishchenko, and Robert Brayton. 2011. Efficient implementation of property directed reachability. In *2011 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 125–134.

[10] Yu-Wei Fan, SooHyuk Cho, Aarti Gupta, and Sharad Malik. 2026. Agentic-IC3: Enabling Semantic Proof Search in IC3 Model Checking. *arXiv preprint arXiv:2609.27162* (2026).

[11] Wenji Fang, Guangyu Hu, and Hongce Zhang. 2023. r-map: Relating implementation and specification in hardware refinement checking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2023).

[12] Wenji Fang, Rongjian Liang, Yi-Chen Lu, Cunxi Yu, Chen Chen, Shang Liu, Chenhui Deng, Wenkai Li, Haoxing Ren, and Zhiyao Xie. 2026. E-Layout: Layout-Stage Logic Restructuring via RL-Optimized Equality-Saturated Graph Retrieval-Augmented Generation. In *International Conference on Computer-Aided Design (ICCAD)*.

[13] Wenji Fang, Yao Lu, Shang Liu, Jing Wang, Ziyan Guo, Junxian He, Fengbin Tu, and Zhiyao Xie. 2026. Dr. RTL: Autonomous Agentic RTL Optimization through Tool-Grounded Self-Improvement. In *International Conference on Computer-Aided Design (ICCAD)*.

[14] Lik Tung Fu, Jie Zhou, et al. 2026. ChatSVA: Bridging SVA generation for hardware verification via task-specific LLMs. *arXiv preprint arXiv:2604.02811* (2026).

[15] Samuele Germiniani and Graziano Pravadelli. 2022. Harm: a hint-based assertion miner. *TCAD* (2022).

[16] Aman Goel and Karem Sakallah. 2019. Model checking of verilog rtl using ic3 with syntax-guided abstraction. In *NASA Formal Methods Symposium*. Springer.

[17] Christopher B Harris and Ian G Harris. 2016. Glast: Learning formal grammars to translate natural language specifications into hardware assertions. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 966–971.

[18] Guangyu Hu, Mingkai Miao, Zhiyuan Yan, Xiaofeng Zhou, Wei Zhang, and Hongce Zhang. 2026. Enhancing Word-Level Property Directed Reachability with LLM-Driven Semantic Guidance. *arXiv preprint arXiv:2609.30131* (2026).

[19] Warren A Hunt Jr, Matt Kaufmann, J Strother Moore, and Anna Slobodova. 2017. Industrial hardware and software verification with ACL2. *Philosophical transactions. Series A, Mathematical, physical, and engineering sciences* 375, 2104 (2017), 20150399.

[20] Manvi Jha, Jiaxin Wan, and Deming Chen. 2025. Proof2Silicon: Prompt Repair for Verified Code and Hardware Generation via Reinforcement Learning. *arXiv preprint arXiv:2509.06239* (2025).

[21] Minwoo Kang, Mingjie Liu, Ghaith Bany Hamad, Syed M Suhaib, and Haoxing Ren. 2025. Fveval: Understanding language model capabilities in formal verification of digital hardware. In *DATE*.

[22] Kezhi Li, Min Li, Xiangyu Wen, Shibo Zhao, Jieying Wu, Junhua Huang, and Qiang Xu. 2026. FormalRTL: Verified RTL Synthesis at Scale. arXiv:2603.08738.

[23] You Li, Guannan Zhao, Yunqi He, and Hai Zhou. 2023. Se3: Sequential equivalence checking for non-cycle-accurate design transformations. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[24] Shang Liu, Yao Lu, Wenji Fang, Mengming Li, and Zhiyao Xie. 2024. OpenLLM-RTL: Open Dataset and Benchmark for LLM-Aided Design RTL Generation. In *ICCAD*.

[25] Mao Luo, Hang Ding, Chu-Min Li, Junjie Zhang, Zhiwei Ye, Zhipeng Lü, Caiquan Xiong, and Xinyun Wu. 2026. Bridging LLMs and SAT Solving: Automated Evolution of High-Performance Heuristics. In *IJCAI*.

[26] Hongqin Lyu, Junxing Dong, Yonghao Wang, Zhiteng Chao, Tiancheng Wang, and Huawei Li. 2026. Rtl2lean: Automated RTL-to-Lean Translation with Hierarchical Theorem Generation and Lemma Reuse. arXiv:2607.16855.

[27] Mingkai Miao, Guangyu Hu, Ziyi Yang, and Hongce Zhang. 2026. IC3-Evolve: Proof-/Witness-Gated Offline LLM-Driven Heuristic Evolution for IC3 Hardware Model Checking. In *IJCAI*. 271–278.

[28] Alan Mishchenko, Satrajit Chatterjee, Robert Brayton, and Niklas Een. 2006. Improvements to combinational equivalence checking. In *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*. 836–843.

[29] Aina Niemetz and Mathias Preiner. 2023. Bitwuzla. In *International Conference on Computer Aided Verification*. Springer, 3–17.

[30] Aina Niemetz, Mathias Preiner, Clifford Wolf, and Armin Biere. 2018. Btor2, btormc and boolector 3.0. In *CAV*. Springer.

[31] Marcelo Orenes-Vera, Margaret Martonosi, and David Wentzlaff. 2023. Using LLMs to Facilitate Formal Verification of RTL. arXiv:2309.09437.

[32] Vaishnavi Pulavarthi, Deeksha Nandal, Soham Dan, and Debjit Pal. 2025. Assertionbench: A benchmark to evaluate large-language models for assertion generation. In *Findings of the Association for Computational Linguistics (NAACL)*.

[33] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. 2000. Checking safety properties using induction and a SAT-solver. In *FMCAD*.

[34] Yuheng Su, Tianjun Bu, Qiusong Yang, Yiwei Ci, and Enyuan Tian. 2026. CIll: CTI-Guided Invariant Generation via LLMs for Model Checking. *arXiv preprint arXiv:2602.23389* (2026).

[35] Yiwen Sun, Furong Ye, Zhihan Chen, Ke Wei, and Shaowei Cai. 2026. Discovering heuristics in a complex SAT solver with large language models. *Nature Communications* 17 (2026), 9008.

[36] Yiwen Sun, Furong Ye, Xianyin Zhang, Shiyu Huang, Bingzhen Zhang, Ke Wei, and Shaowei Cai. 2026. AutoSAT: Automatically Optimize SAT Solvers via Large Language Models. *Journal of Artificial Intelligence Research* 86, Article 29 (2026).

[37] Ha Trung Tran. 2026. Open-Source LLM-Driven Formal Verification: A Multi-Agent Pipeline for RTL Repair. *arXiv preprint arXiv:2607.28877* (2026).

[38] CAJ Van Eijk. 2000. Sequential equivalence checking based on structural similarities. *TCAD* (2000).

[39] Jing Wang, Shang Liu, Hangan Zhou, Wenji Fang, Yuchao Wu, Yugao Zhu, and Zhiyao Xie. 2026. CorrectRTL: Boost RTL Generation Quality Through Automated Self-Correction with Instruction Decomposition. In *ICCAD*.

[40] Yonghao Wang, Yang Yin, et al. 2026. CoverAssert: Iterative LLM Assertion Generation Driven by Functional Coverage via Syntax-Semantic Representations. In *Design, Automation & Test in Europe Conference (DATE)*.

[41] Yuchao Wu, Wenji Fang, Jing Wang, Wenkai Li, Ziyan Guo, and Zhiyao Xie. 2026. AssertLLM2: A Comprehensive LLM Benchmark for Assertion Generation from Design Specifications. In *International Conference on Computer-Aided Design (ICCAD)*.

[42] Weihua Xiao, Derek Ekberg, Siddharth Garg, and Ramesh Karri. 2025. Hybrid-NL2SVA: Integrating RAG and Finetuning for LLM-based NL2SVA. In *MLCAD*.

[43] Yue Xing, Huaixi Lu, Aarti Gupta, and Sharad Malik. 2022. Compositional verification using a formal component and interface specification. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*. 1–9.

[44] Jing Xiong, Qi Han, Chenchen Ding, He Xiao, Zunhai Su, Xiachong Feng, Chaofan Tao, and Ngai Wong. 2026. CktFormalizer: Autoformalization of Natural Language into Circuit Representations. arXiv:2605.07782v4.

[45] Zhiyuan Yan, Wenji Fang, Mengming Li, Min Li, Shang Liu, Zhiyao Xie, and Hongce Zhang. 2025. AssertLLM: Generating Hardware Verification Assertions from Design Specifications via Multi-LLMs. In *ASP-DAC*.

[46] Ziyi Yang, Wenji Fang, Chen Chen, Zhiyao Xie, and Hongce Zhang. 2026. CircuitProver: Agentic Lean 4 Theorem Proving with Reusable Circuit Proof Library for Hardware Verification. arXiv:2607.27259.

[47] Cunxi Yu, Rongjian Liang, Chia-Tung Ho, and Haoxing Ren. 2025. Autonomous Code Evolution Meets NP-Completeness. arXiv:2509.07367

[48] Emily Yu, Nils Froleyks, Armin Biere, and Keijo Heljanko. 2022. Stratified certification for k-induction. In *The 22nd Conference on Formal Methods in Computer-Aided Design–FMCAD 2022*. TU Wien Academic Press, 11.

[49] Niansong Zhang, Chenhui Deng, Johannes Maximilian Kuehn, Chia-Tung Ho, Cunxi Yu, Zhiru Zhang, and Haoxing Ren. 2025. ASPEN: LLM-Guided E-Graph Rewriting for RTL Datapath Optimization. In *MLCAD*.

[50] Jie Zhou, Youshu Ji, Ning Wang, Yuchen Hu, Xinyao Jiao, Bingkun Yao, Xinwei Fang, Shuai Zhao, Nan Guan, and Zhe Jiang. 2025. Insights from Rights and Wrongs: A Large Language Model for Solving Assertion Failures in RTL Design. In *2025 62nd ACM/IEEE Design Automation Conference (DAC)*. IEEE.

[51] Melisande Zonta-Roudes, Nora Hinderling, and Shweta Shinde. 2026. Stitch: Assertion-Guided Patching of On-Chip Protocol Implementations using LLMs. In *63rd ACM/IEEE Design Automation Conference (DAC)*. ACM.