Towards Reliable LLM-Assisted Verification Artifact Generation in EDA

Khushboo Qayyum[2], Qurrat Ul Ain[3], Muhammad Hassan[1,2], Asjad Afnan[2], Rolf Drechsler[1,2]
[1]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
[3]Department of Software Engineering, NUCES, Islamabad, Pakistan
{khushboo.qayyum,asas03}@dfki.de {hassan,drechsler}@uni-bremen.de quratul.ain.v@isb.nu.edu.pk

***Abstract*—*Large Language Models* (LLMs) are increasingly being explored for design and verification tasks in *Electronic Design Automation* (EDA), including the generation of assertions, test scenarios, coverage-driven methodologies, and other verification artifacts. While early results are promising, practical adoption remains limited by concerns over reliability, semantic correctness, and the overall usefulness of generated outputs within real verification flows. This paper proposes a reliability-oriented framework for LLM-assisted verification artifact generation in EDA, integrating contextual grounding, structured generation, and verification-aware feedback to improve artifact quality across multiple abstraction levels. The two case studies demonstrate that assertion quality can vary substantially across modules, reinforcing the need for systematic quality assessment of LLM-generated assertions.**

***Index Terms*—Hardware verification, large language models, assertion generation, formal methods, AI in EDA**

I. Introduction

Large Language Models (LLMs) are increasingly being explored as assistants for hardware verification. One of the most active directions is the generation of *SystemVerilog Assertions (SVA)* from *Natural Language (NL)* requirements, *Register-Transfer Level (RTL)* context, interface descriptions, and design documentation. Since LLMs are particularly good at *Natural Language Processing (NLP)*, these tasks become a natural use case. Assertions are valuable in both simulation and formal verification, but writing them requires design knowledge, understanding of temporal behavior, and careful mapping from informal intent to signals and timing. Recent works have therefore studied LLM-based assertion generation from complete specifications, structured design context, knowledge graphs, and signal-mapping flows (cf. Section II-A).

Despite this progress, the generated assertions are not automatically useful. It may refer to the wrong signal, use the wrong clock or reset, encode the wrong timing bound, or check behavior that is only loosely related to the requirement. It may also compile and prove while still providing little confidence. In formal verification, this is the problem of vacuity. A property may pass because its trigger is unreachable, because assumptions exclude the interesting behavior, or because some part of the property has no effect on the result. Earlier works on vacuity detection showed that vacuous passes can compromise the usefulness of formal verification and reported that, in early formal runs of new hardware designs, a significant fraction of properties could pass vacuously [1]–[4]. Recent LLM-based verification work has started to address parts of the quality problem (cf. Section II-A). Some methods improve assertion generation through structured specification extraction, RTL-aware context, signal bridging, or functional-coverage feedback [5], [6]. Others focus on security properties and explicitly aim to produce non-vacuous assertions [7]. However, a more structured approach to provide reliable qualitative assurance of the generated assertion is still missing.

Contribution: This paper proposes a reliability-oriented framework for Assertion Assurance in LLM-assisted verification artifact generation. More concretely, we focus on the post-generation quality assessment problem. Given a NL requirement, RTL context, assumptions, and one or more LLM-generated candidate assertions, the framework evaluates whether each assertion should be accepted, repaired, or rejected. If an assertion proves, the framework applies various vacuity checking techniques to determine whether the proof is meaningful. If an assertion fails, the framework uses causal counterexample analysis and failure localization to distinguish likely assertion defects from possible design bugs. The output is an Assertion Assurance Record, which summarizes the evidence collected for each candidate assertion. The proposed framework fits naturally within the iterative view of LLM-assisted hardware verification, where an LLM proposes or modifies a verification artifact, *Electronic Design Automation (EDA)* tools evaluate it, and the result is used to refine the next step. Prior work has described this style of interaction as a prompt, verify, and repeat loop [8]. Our contribution is to make the verification step more specific for generated assertions. We argue that proof pass or fail is not a sufficient interface between LLMs and verification tools. The interface should include assurance evidence that helps engineers and tools understand why a generated assertion passed, why it failed, and whether it is worth keeping. We use this framework on assertions generated by state-of-the-art LLM based assertion generation tools from two modules from the OpenTitan suite [9] and use them as a case study. Using the outcomes of these two case studies, we substantiate our claim to the need for such reliability-oriented frameworks.

This work was supported in part by the German Ministry for Research, Technology and Space (BMFTR) with project *ExaVerse* (grant number 01IW25003).

II. Preliminaries and Motivation

A. Related Work

Assertion-based verification (ABV) is widely used in hardware verification because assertions improve observability, support simulation and formal verification, and help detect corner-case design errors. However, writing effective SVAs remains difficult because assertions must accurately capture design intent, signal-level behavior, temporal constraints, clocking discipline, reset semantics, and environmental assumptions. Prior surveys show that assertion quality has a direct effect on verification effectiveness, while manual assertion development remains expertise-intensive and error-prone [10]. Consequently, automatic assertion generation from specifications, RTL structure, simulation traces, and design documentation has been an active research area [11]–[17].

LLMs have recently been explored as flexible assistants for generating hardware verification artifacts. [7] studied LLM-generated security assertions and showed that prompt detail strongly affects assertion quality. [18] explored circuit-aware translation from natural language to SVA, emphasizing the need to include design context when mapping specification text to RTL signals. ChIRAAG [19] converts NL specifications into a structured format before generating SVAs and uses simulation log feedback to repair generated assertions. LAAG-RV [20] improves signal synchronization by providing both Verilog design information and specification text to the LLM, followed by simulator-guided refinement. AssertLLM [6] and AssertLLM2 [21] further decompose assertion generation into specification extraction, signal mapping, and SVA generation, showing that full-document processing and explicit signal mapping can improve structural and functional correctness. These methods demonstrate the potential of LLM-assisted SVA generation, but they also show that direct generation is not reliable without context, feedback, and review.

Several works extend the use of LLMs beyond assertion generation to broader verification and debugging tasks. Domain-adapted LLMs have been studied for VLSI design and formal verification, showing that hardware-specific adaptation can improve performance over generic models of similar size [22]. LLMs have also been applied to HDL bug identification and patching using retrieval-augmented generation [23], [24], RTL syntax repair [5], and RTL debug automation [25]. Hardware-security-focused debugging has similarly motivated domain-specific LLMs trained on design defects and fixes [26]. These works are relevant because they show a common trend: LLM outputs in EDA are useful only when combined with domain context, tool feedback, and structured evaluation.

Benchmarking and evaluation efforts have also emerged to assess LLM capability in hardware verification. FVEval provides a benchmark for evaluating LLM performance on formal verification tasks, including NL-to-SVA and RTL-guided assertion generation [27]. OpenLLM-RTL provides infrastructure and datasets for evaluating LLM-aided RTL generation and verification tasks [28]. Other studies examine LLM use in processor verification [29], software/hardware co-design for LLM-based design verification [30], and automated incremental proof generation [31]. These works provide important evaluation infrastructure, but benchmark metrics such as syntax correctness, pass rate, or coverage do not fully answer whether a generated assertion should be accepted into a real verification flow.

Recent work has also explored verification-aware feedback loops. Mutation testing has been used to evaluate the usefulness of LLM-generated invariants and formal properties [32]. LISA combines retrieval-augmented generation, chain-of-thought reasoning, formal verification, and coverage feedback to generate and refine SVAs [33]. AssertionForge constructs a structured representation of specifications and RTL to improve assertion generation through better contextual grounding [34]. AssertGen bridges high-level verification objectives to RTL signals using cross-layer signal chains before generating assertions [35]. Fine-tuned assertion-generation approaches have also been proposed to reduce syntax and functional errors through subtask-specific prompting and iterative refinement [36]. These methods improve the generation process, but their main objective remains producing better assertions rather than producing a structured assurance decision for each generated assertion.

Security-oriented assertion generation has developed in parallel and is particularly relevant because it emphasizes non-vacuity, coverage, and vulnerability relevance. LASP generates security properties from *System-on-Chip (SoC)* specifications, RTL features, and security assets [37]. LASSO uses LLM-aided property generation for assertion-based SoC security verification [38]. FV-PAL combines structural partitioning with LLM-guided property generation for scalable formal verification [39]. SPELL maps SoC security concerns into LLM-guided secure design and verification artifacts [40]. HADA uses multi-source data to train LLMs for hardware security assertion generation [41], while LAMBDA uses LLM-assisted assertions for malicious bug detection in hardware designs [42]. These works show the importance of non-vacuity and coverage-aware property generation, but their focus is mainly security-specific property generation rather than a general post-generation assurance layer for arbitrary LLM-generated assertions.

The need for such an assurance layer is supported by classic formal verification research. Vacuity detection asks whether a property passed for a meaningful reason or because some part of the property had no effect on the result [2], [3]. This is especially important for LLM-generated assertions because an assertion may look plausible, compile successfully, and even prove while checking little or none of the intended behavior. Counterexample explanation helps engineers understand why a property failed by identifying relevant states, signal values, and transitions in the failing trace [1]. Causality and responsibility provide a basis for reasoning about which components of a model or formula influence a verification outcome [4]. These ideas were originally developed for human-written specifications and formal models, but they are directly applicable to LLM-generated assertions because they provide evidence

about whether a generated assertion is meaningful, sensitive to the intended behavior, and useful for debugging.

Overall, prior work shows a clear progression from rule-based NL assertion generation to LLM-assisted SVA generation, and then to grounded, tool-assisted, and feedback-driven verification flows. Existing methods have made strong progress in generating assertions from specifications, binding them to RTL context, improving them through simulation or formal feedback, and evaluating them through coverage or mutation testing. However, most approaches remain generation-centered as they ask how to produce better assertions. The complementary question remains insufficiently addressed, after an LLM has generated an assertion, what evidence is needed before that assertion should be trusted?

B. Running Example

To motivate the need for post-generation assertion assurance, consider a simple arbiter requirement:

> "If '`req`' is asserted outside reset, then '`grant`' must be asserted within one to three cycles."

A reasonable SVA for this requirement is shown in Listing 1. The assertion uses the active-low reset only in the `disable iff` condition and checks that every request is followed by a grant within the required latency window.

Listing 1: A reasonable assertion for the request-grant requirement.

```
assert property (@(posedge clk) disable iff (!rst_n)
  req |-> ##[1:3] grant
);
```

Now suppose an LLM generates the candidate assertion in Listing 2. At first glance, the assertion appears close to the requirement. It uses the correct clock, the correct reset signal, and the expected request-grant timing window. A formal tool may also prove it.

Listing 2: A plausible but vacuous LLM-generated assertion.

```
assert property (@(posedge clk) disable iff (!rst_n)
  (req && !rst_n) |-> ##[1:3] grant
);
```

However, the proof is not meaningful. The property is disabled whenever `!rst_n` is true, while the antecedent also requires `!rst_n`. As a result, the antecedent can never be evaluated. The assertion passes because the trigger is unreachable, not because the design satisfies the request-grant requirement.

This example shows why proof status alone is not enough. A flow that accepts generated assertions after syntax checking and formal proof would mark Listing 2 as successful. A verification engineer, however, would want to know whether the trigger is reachable, whether the consequent matters, and whether the proof depends on the timing bound. For this candidate assertion, a witness obligation such as Listing 3 would be unreachable.

Listing 3: Witness obligation exposing the unreachable antecedent.

```
cover property (@(posedge clk) disable iff (!rst_n)
  (req && !rst_n)
);
```

The same issue can be exposed through counterfactual edits. For example, replacing the consequent with `1'b0`, as shown in Listing 4, should normally make a request-grant assertion fail if the request trigger is reachable. If the edited assertion still proves, then the consequent was not influencing the proof result.

Listing 4: Counterfactual edit testing if the consequent matters.

```
assert property (@(posedge clk) disable iff (!rst_n)
  (req && !rst_n) |-> ##[1:3] 1'b0
);
```

For Listing 2, the reset and antecedent conflict would receive high responsibility for the vacuous pass, while `grant` and the delay range `##[1:3]` would receive little or no responsibility because they are never exercised. The proper repair is to remove `!rst_n` from the antecedent and rely on the reset guard, leading back to the assertion in Listing 1.

The opposite problem can also occur. Consider another LLM-generated assertion for the same requirement, shown in Listing 5. This assertion is not vacuous, but it encodes a timing bound that is too strict.

Listing 5: An assertion with an overly strict timing bound.

```
assert property (@(posedge clk) disable iff (!rst_n)
  req |-> ##[1:2] grant
);
```

This assertion may fail on a correct design if `grant` arrives in the third cycle after `req`. A counterexample trace may look like Listing 6.

Listing 6: Counterexample trace for a valid third-cycle grant.

```
cycle 5: req   = 1
cycle 6: grant = 0
cycle 7: grant = 0
cycle 8: grant = 1
```

The trace shows that the design responds within three cycles, which is allowed by the requirement, but not by the generated assertion. The proper action is not to reject the design. The proper action is to repair the assertion by changing the timing range from `##[1:2]` to `##[1:3]`.

These two cases show the central problem addressed in this paper. A passing generated assertion may be vacuous, and a failing generated assertion may be incorrect rather than bug-revealing. Therefore, LLM-generated assertions should be treated as candidate artifacts that require post-generation assurance. The next section presents a framework that checks such assertions using vacuity analysis, witness obligations, counterfactual edits, causal counterexample analysis, responsibility ranking, and structured repair feedback.

III. Reliable LLM-Assisted Verification Artifact Generation

A. Overview

This section presents a post-generation assurance framework for LLM-generated assertions. The framework does not as-

sume a particular assertion generation method. Instead, it starts from the point where one or more candidate assertions have already been produced by an LLM-assisted flow. The goal is to determine whether each candidate assertion should be accepted, repaired, or rejected before it is used in a verification environment. The complete flow is shown in Fig. 1. For each generated assertion, the framework first performs context alignment, then runs formal evaluation. Proved assertions enter the pass-side branch, where vacuity, witness, counterfactual, and responsibility checks determine whether the proof is meaningful. Failed assertions enter the fail-side branch, where causal counterexample analysis and failure localization determine whether the assertion should be repaired or whether the trace should be reviewed as a possible RTL bug. The output of either branch is an accept, repair, or reject decision, together with an Assertion Assurance Record.

Given a NL requirement R, RTL context C, formal assumptions A, and a set of LLM-generated candidate assertions $S = \{s_1, s_2, \ldots, s_n\}$, the framework evaluates each assertion using formal evidence and diagnostic checks. The requirement describes the intended behavior. The RTL context provides nomenclature, hierarchy, mapping, and protocol details. The assumptions describe allowed input behavior and environment constraints used by the formal setup. The candidate assertions may come from any LLM-assisted assertion generation method. The key idea is that the LLM output is treated as a proposal rather than a finished verification artifact. The framework uses formal evaluation as the entry point into a broader assurance process. This process checks for vacuous properties, explains informative failure, ranks likely causes, and produces structured feedback for repair. In the next sections, we discuss each step in detail.

B. Assertion-Context Alignment

The first stage aligns each generated assertion with the available design context. This step checks whether the assertion can be meaningfully evaluated against the RTL and assumptions. In practice, many LLM-generated assertions fail before deeper reasoning is possible because of issues with nomenclature and encoding. The alignment stage checks whether referenced signals exist, whether clock and reset usage match the RTL context, and whether the assertion conflicts with known assumptions. This stage is not intended to prove that the assertion is correct. Its purpose is to decide whether the assertion is grounded enough to enter formal evaluation. For example, if an assertion refers to `grant_valid` but the interface only contains `grant`, the assertion is flagged for repair or rejection.

C. Formal Evaluation

After alignment, each candidate assertion is evaluated using a formal property checker or assertion checking engine. The formal result is divided into two main branches. If the assertion proves, the framework performs pass-side assurance to determine whether the proof is meaningful. If the assertion fails, the framework performs fail-side diagnosis to determine the cause of failure. This distinction is important because a pass and a fail are both ambiguous for LLM-generated assertions as each can arise from multiple underlying causes (see Section II-B).

D. Pass-Side Assurance

When a generated assertion proves, the framework first checks whether the proof is vacuous. Consider the request-grant requirement from Section II-B. If the generated assertion is written as Listing 2, then the assertion may prove, but the proof is not meaningful. The antecedent requires `!rst_n`, while the property is disabled whenever `!rst_n` is true. The framework classifies this as reset-dominance vacuity because the reset condition prevents the intended trigger from being evaluated. To support a vacuity result, the framework generates witness obligations for the proof. The framework then applies counterfactual assertion edits. These are small diagnostic changes to the candidate assertion, used to test whether the proof depends on the right components. The edits may replace the consequent with `1'b0`, replace it with `1'b1`, change the delay range, remove or invert a reset condition, or weaken and strengthen parts of the antecedent. These edits are not intended to create replacement assertions for use in the verification environment. They are probes that help determine which components matter to the verification result. If both the original and edited assertions prove, then the consequent may not be influencing the result. This suggests that the trigger may be unreachable or that the assumptions exclude the interesting case. In this way, counterfactual edits provide a practical test of causal relevance. A component is treated as meaningful only if changing it changes the proof, vacuity, witness, or counterexample outcome. The results of these checks are summarized through a responsibility ranking. Responsibility is used here as a lightweight diagnostic score rather than as a complete formal computation over the entire RTL state space. A component is assigned higher responsibility if a small change to that component changes the observed verification outcome. A component receives low or zero responsibility if changing it has no effect. In the reset-dominance example from Section II-B, the conflict between the reset guard and the antecedent receives high responsibility, while `grant` and the delay range receive little or no responsibility because they are never exercised. This ranking gives the repair stage a concrete target.

E. Fail-Side Diagnosis

When a generated assertion fails, the framework analyzes the counterexample before deciding whether to repair the assertion or report a possible RTL bug. This is necessary because generated assertions can fail for reasons unrelated to the design. The framework performs causal counterexample analysis by identifying the signal values and time steps that are relevant to the failure. The analysis focuses on the signals used in the assertion and the portions of the trace that determine whether the property holds. The failure diagnosis is then converted into repair feedback.

F. Decision: Repair, Rejection, and Acceptance

After pass-side or fail-side analysis, the framework decides whether the assertion should be accepted, repaired, or rejected.

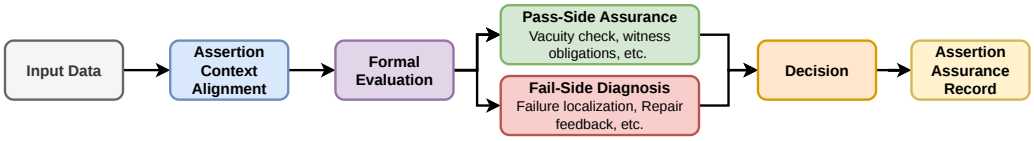

Fig. 1: Overview of reliable LLM-assisted verification artifact generation framework

An assertion is accepted only when the collected evidence supports its usefulness. In the pass case, this means that the assertion is aligned with the RTL context, proves non-vacuously, has reachable witness obligations, and contains requirement-critical components that influence the result. In the fail case, this means that the failure is consistent with the requirement and should be reviewed as a possible design issue. An assertion is repaired when the framework can localize the defect and generate feedback that is likely to preserve the original requirement. Typical repairs fix shallow assertion errors such as wrong polarity, loose timing, or near-miss signal names. In such cases, the structured diagnosis is sent back to the LLM or to the verification engineer. The revised assertion is then evaluated again using the same assurance flow. An assertion is rejected if it is irrelevant, unrepairable, vacuous, RTL-inconsistent, or undiagnosable. Rejection is also appropriate when the original requirement is too ambiguous to support a meaningful assertion without human clarification.

G. Assertion Assurance Record

The final output of the framework is an *Assertion Assurance Record*. This record summarizes the evidence collected for a candidate assertion. It is not a proof certificate and does not claim that the assertion is sufficient for signoff. Its role is to make the acceptance, repair, or rejection decision traceable. The record includes the requirement identifier and text, the candidate assertion, the RTL signals used, the clock and reset context, the assumptions, the proof result, the vacuity result, the witness results, the counterfactual edit results, the responsibility ranking, the counterexample diagnosis when applicable, the repair history, the final decision, and the human-review priority. An example record for the repaired request-grant assertion is shown in Listing 7.

Listing 7: Example Assertion Assurance Record for an accepted assertion.

```
Assertion Assurance Record

Requirement:
  If req is asserted outside reset, grant must arrive
     within 1 to 3 cycles.

Candidate assertion:
  @(posedge clk) disable iff (!rst_n)
    req |-> ##[1:3] grant

Context:
  clock: clk
  reset: rst_n, active low
  signals: req, grant

Formal result:
  PASS

Vacuity:
  non-vacuous

Witness obligations:
  antecedent reachable: yes
  full req-grant sequence reachable: yes

Counterfactual edits:
  consequent replaced by 0: proof fails
  timing changed to ##[4:6]: proof fails
  reset inverted: trigger unreachable

Responsibility ranking:
  req trigger: high
  grant consequent: high
  timing bound ##[1:3]: high
  reset guard: high

Repair history:
  one repair iteration

Decision:
  accept

Review priority:
  low
```

For a vacuous assertion, the same record format captures the diagnosis and repair feedback, as shown in Listing 8.

Listing 8: Example Assertion Assurance Record fields for a repaired vacuous assertion.

```
Decision:
  repair

Diagnosis:
  reset-dominance vacuity

Repair feedback:
  The antecedent requires reset to be active while
  the assertion is disabled during reset. Remove
  !rst_n from the antecedent and rely on
  disable iff (!rst_n).
```

H. Preliminary Case Study

This section presents two case studies to reinforce the need for quality assessment of the assertion generated through the use of LLMs. We apply our framework to the assertions generated using the AssertLLM2 [21] for the two OpenTitan modules [9] i.e., CSRNG and AONTimer. We selected 3 files from the CSRNG module and 4 files from the AONTimer module. The total Lines of Code were similar for both modules ($\approx$ 2000 *LoC*). We used Qwen3-coder:30b for these experiments with temp 0 and 16k token limit.

Table I shows the number of generated assertions for the given module against the number of vacuous assertions within the generated assertions. The LLM produced reliable assertions that passed the quality check for the CSRNG module, but this outcome did not carry over to the AONTimer. Of the 32 assertions generated, only 12 passed the framework's context alignment and formal evaluation check. JasperGold

then proved 7 of these, while the remaining 5 failed. Of the 7 proven assertions, 5 were usable and 2 were vacuous. The key finding is that, despite the two modules being of comparable size, one produced an overwhelming share of unusable assertions (27 of 32) underscoring the need for quality measures on LLM-generated assertions.

TABLE I: Vacuous Assertion Assessment

Case Study	AssertLLM2	Post-generation Assurance Framework			
	Generated	Accepted	Proved	Counter Examples	Vacuous
CSRNG	14	8	8	0	0
AONTimer	32	12	7	5	2

IV. Conclusion

LLMs are increasingly being explored for design and verification tasks in EDA, including the generation of assertions, test scenarios, coverage-driven methodologies, and other verification artifacts. While early results are promising, practical adoption remains limited by concerns over reliability, semantic correctness, and the overall usefulness of generated outputs within real verification flows. This paper proposes a reliability-oriented framework for LLM-assisted verification artifact generation in EDA, integrating contextual grounding, structured generation, and verification-aware feedback to improve artifact quality across multiple abstraction levels.

References

[1] A. Groce *et al.*, "What went wrong: Explaining counterexamples," in *Proceedings of the 10th International SPIN Workshop on Model Checking of Software*. Springer, 2003, pp. 121–135.

[2] I. Beer *et al.*, "Efficient detection of vacuity in temporal model checking," *Formal Methods in System Design*, vol. 18, no. 2, pp. 141–163, 2001.

[3] O. Kupferman *et al.*, "Vacuity detection in temporal model checking," *International Journal on Software Tools for Technology Transfer*, vol. 4, no. 2, pp. 224–233, 2003.

[4] H. Chockler *et al.*, "What causes a system to satisfy a specification?" *ACM Transactions on Computational Logic*, vol. 9, no. 3, pp. 1–26, 2008.

[5] Y. Tsai *et al.*, "RTLFixer: Automatically fixing RTL syntax errors with large language model," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, 2024, pp. 1–6.

[6] Z. Yan *et al.*, "AssertLLM: Generating hardware verification assertions from design specifications via multi-LLMs," in *Proceedings of the 30th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2025, pp. 414–421.

[7] R. Kande *et al.*, "(security) assertions by large language models," *IEEE Transactions on Information Forensics and Security*, 2024.

[8] M. Hassan *et al.*, "Prompt. verify. repeat. llms in the hardware verification cycle," in *2025 IEEE International Conference on Omni-layer Intelligent Systems (COINS)*. IEEE, 2025, pp. 1–6.

[9] lowRISC CIC, "OpenTitan: Open source silicon root of trust," 2024, accessed: 2024-04-05. [Online]. Available: https://github.com/lowRISC/opentitan

[10] H. Witharana *et al.*, "A survey on assertion-based hardware verification," *ACM Computing Surveys*, vol. 54, no. 11s, pp. 1–33, 2022.

[11] R. Krishnamurthy *et al.*, "Controlled natural language framework for generating assertions from hardware specifications," in *2019 IEEE 13th International Conference on Semantic Computing (ICSC)*, 2019, pp. 367–370.

[12] J. Zhao *et al.*, "Automatic assertion generation from natural language specifications using subtree analysis," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2019, pp. 598–601.

[13] S. J. Frederiksen *et al.*, "Automated assertion generation from natural language specifications," in *2020 IEEE International Test Conference (ITC)*, 2020, pp. 1–5.

[14] F. Aditi *et al.*, "Hybrid rule-based and machine learning system for assertion generation from natural language specifications," in *2022 IEEE 31st Asian Test Symposium (ATS)*, 2022, pp. 1–6.

[15] M. Orenes-Vera *et al.*, "AutoSVA: Democratizing formal verification of RTL module interactions," in *Proceedings of the 58th ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 1–6.

[16] F. Yagci *et al.*, "GoldMine: Automatic assertion generation using data mining and static analysis," in *Proceedings of the Design, Automation & Test in Europe Conference (DATE)*, 2010, pp. 626–629.

[17] S. Germiniani *et al.*, "HARM: A hint-based assertion miner," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2022.

[18] C. Sun *et al.*, "Towards improving verification productivity with circuit-aware translation of natural language to systemverilog assertions," in *First International Workshop on Deep Learning-Aided Verification*, 2023.

[19] B. Mali *et al.*, "Chiraag: Chatgpt informed rapid and automated assertion generation," in *2024 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 2024, pp. 680–683.

[20] K. Maddala *et al.*, "LAAG-RV: LLM assisted assertion generation for rtl design verification," in *2024 IEEE 8th International Test Conference India (ITC India)*. IEEE, 2024, pp. 1–6.

[21] Y. Wu *et al.*, "AssertLLM2: A comprehensive llm benchmark for assertion generation from design specifications," *arXiv preprint arXiv:2605.27472*, 2026.

[22] M. Liu *et al.*, "Domain-adapted LLMs for VLSI design and verification: A case study on formal verification," in *2024 IEEE 42nd VLSI Test Symposium (VTS)*, 2024, pp. 1–4.

[23] K. Qayyum *et al.*, "From bugs to fixes: HDL bug identification and patching using LLMs and RAG," in *2024 IEEE LLM Aided Design Workshop (LAD)*, 2024, pp. 1–5.

[24] K. Qayyum *et al.*, "LLM-assisted bug identification and correction for verilog HDL," *ACM Transactions on Design Automation of Electronic Systems*, 2025.

[25] K. Xu *et al.*, "Meic: Re-thinking RTL debug automation using LLMs," *arXiv preprint arXiv:2405.06840*, 2024.

[26] W. Fu *et al.*, "LLM4SecHW: Leveraging domain-specific large language model for hardware debugging," in *2023 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*. IEEE, 2023, pp. 1–6.

[27] M. Kang *et al.*, "FVEval: Understanding language model capabilities in formal verification of digital hardware," in *2025 Design, Automation & Test in Europe Conference (DATE)*, 2025, pp. 1–6.

[28] S. Liu *et al.*, "OpenLLM-RTL: Open dataset and benchmark for LLM-aided design RTL generation," in *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2024, pp. 1–9.

[29] C. Xiao *et al.*, "LLM-based processor verification: A case study for neuromorphic processor," in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2024, pp. 1–6.

[30] L. J. Wan *et al.*, "Software/hardware co-design for LLM and its application for design verification," in *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2024, pp. 435–441.

[31] K. Qayyum *et al.*, "Late breaking results: LLM-assisted automated incremental proof generation for hardware verification," in *Proceedings of the 61st ACM/IEEE Design Automation Conference (DAC)*, 2024, pp. 1–2.

[32] M. Hassan *et al.*, "LLM-guided formal verification coupled with mutation testing," in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2024, pp. 1–2.

[33] S. Paul *et al.*, "LISA: LLM informed systemverilog assertion generation with RAG and chain-of-thought," in *2025 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2025, pp. 1–6.

[34] Y. Bai *et al.*, "AssertionForge: Enhancing formal verification assertion generation with structured representation of specifications and rtl," in *2025 IEEE International Conference on LLM-Aided Design (ICLAD)*, 2025, pp. 85–92.

[35] H. Lyu *et al.*, "AssertGen: Enhancement of LLM-aided assertion generation through cross-layer signal bridging," in *2025 IEEE 34th Asian Test Symposium (ATS)*, 2025, pp. 25–30.

[36] M. Shahidzadeh *et al.*, "Automated verilog assertion generation using fine-tuned LLMs with subtask-specific iterative prompting," in *2025 26th International Symposium on Quality Electronic Design (ISQED)*, 2025, pp. 1–8.

[37] A. Ayalasomayajula *et al.*, "Lasp: LLM assisted security property generation for soc verification," in *Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD*, 2024, pp. 1–7.

[38] D. R. Ankireddy *et al.*, "LASSO: LLM-aided security property generation for assertion-based SoC verification," in *2025 ACM/IEEE 7th Symposium on Machine Learning for CAD (MLCAD)*, 2025.

[39] S. Paria *et al.*, "FV-PAL: Scalable formal verification through partitioning and LLM-guided property generation," in *2025 IEEE 43rd International Conference on Computer Design (ICCD)*, 2025.

[40] S. Paria *et al.*, "SPELL: An end-to-end tool flow for LLM-guided secure soc design for embedded systems," *IEEE Embedded Systems Letters*, 2024.

[41] W. Fu *et al.*, "HADA: Leveraging multi-source data to train large language models for hardware security assertion generation," in *2025 ACM/IEEE 7th Symposium on Machine Learning for CAD (MLCAD)*, 2025.

[42] S. Paria *et al.*, "LAMBDA: LLM-assisted malicious bug detection and analysis in hardware designs," in *2025 IEEE International Test Conference (ITC)*, 2025.