# Automation of Polynomial Formal Verification using Large Language Models

Luca Müller[1], Khushboo Qayyum[1], Nele Hugo[2], Muhammad Hassan[1,2], Rolf Drechsler[1,2]
[1]Cyber-Physical Systems, DFKI, Bremen, Germany
[2]Institute of Computer Science, University of Bremen, Bremen, Germany
{luca.mueller, khushboo.qayyum}@dfki.de {hugone, hassan, drechsler}@uni-bremen.de

***Abstract*—Current advancements in technology have not only made digital systems ubiquitous, but also made them functionally more complex. Since they are in charge of various critical tasks, it is imperative that they are error-free and perform according to their specifications, requiring extensive testing before fabrication. Formal verification methods can guarantee that a circuit is bug-free, but are prone to time and space explosion, making their adoption challenging. *Polynomial Formal Verification (PFV)* aims to make these formal methods viable for practical use by establishing polynomial time and space complexity bounds on the verification of hardware designs. However, this requires a lot of manual effort in the calculation and proof of these bounds.**

**In this paper, we automate PFV using *Large Language Models (LLMs)* and aim to reduce the manual work associated with proof generation. We implement a framework that automatically calculates polynomial complexity bounds for the verification of a given circuit. Based on these bounds, an LLM generates an executable proof for the Z3 theorem prover. We demonstrate our framework by evaluating multiple LLMs for the quality and consistency of generated proofs for different types of adder circuits, making all code and artifacts openly available.**

## I. Introduction

Modern circuit designs are gaining complexity in accordance with Moore's law and are becoming ubiquitous [1]. They surround us humans and are responsible not only for trivial everyday tasks, but also in various safety-critical applications. This abundance of digital systems makes it imperative that they are bug-free and perform in accordance to their specifications. Failure to perform correctly can cause bugs and malfunctions, which can result in huge financial losses, product recalls [2], and threat to human safety. Therefore, it is vital that these designs are properly tested before the fabrication process is concluded. Various design testing techniques like simulation-based testing, fuzzing, code-inspection, etc. are used to perform this pre-silicon verification. However, these techniques cannot guarantee a $100\%$ bug-free device.

Formal verification methods use mathematical proofs to guarantee that a device will behave according to its specification. However, the increasing complexity of modern circuit designs makes these formal methods computationally expensive [3], making their adoption difficult. *Polynomial Formal Verification (PFV)* [4] offers a solution to this predicament by providing formal verification methods based on commonly used proof engines like *Binary Decision Diagrams (BDDs)* that ensure polynomial time and space complexity bounds for specific classes of circuits. In the past, a major drawback of PFV has been the tedious manual effort involved in the calculation and proof of these bounds. While previous works explored automated PFV [5], [6], it relies on algorithmic pattern extraction and representation, which may limit general applicability.

Recent advancements in *Artificial Intelligence (AI)*, particularly in the area of generative AI, have led to the emergence of *Large Language Models (LLMs)*, which show remarkable results in text generation and *Natural Language Processing (NLP)*. Through the use of LLMs, many cumbersome tasks can be offloaded, reducing manual work, and shifting the focus to the exploration of new domains. Proof-writing has been successfully assisted using NLP heuristics in the past [7], showing the potential of LLMs in this direction.

Therefore, this work aims to automate the manual effort of proof generation by the integration of LLMs into the PFV process. Unlike previous attempts [8], which only provided natural language output and did not prove concrete bounds, we focus on the generation of structured output. Specifically, the LLM generates an executable proof for the Z3 theorem prover [9], increasing trust in the soundness of the proof and easing its validation. This paper provides three main contributions:

1) Design and implement an open-source framework to automatically derive and prove polynomial complexity bounds for combinational circuits with LLMs.
2) Provide three metrics to determine the similarity of proofs generated by an LLM.
3) Evaluate five different LLMs on the quality and consistency of their generated proofs for three adder designs.

All code, prompts, results and other artifacts produced and used in the scope of this work are made openly available [1].

## II. Preliminaries

### A. Large Language Models

LLMs are transformer-based AI models designed to interpret and generate human language. They utilize the self-attention mechanism that allows them to consider all previous tokens in a text and add weight based on their importance to create context [10]. Using this context, LLMs predict the next coherent and context-aware token sequence. They are trained on a massive corpus of textual data, which allows them to generate organized and logical text. Their ability to predict the next sequence based on input patterns and dependencies learned from the training dataset gives rise to their reasoning

979-8-3315-6337-0/26/$31.00 ©2026 IEEE

[1]https://github.com/lucamu/LLMxPFV

abilities. Input sequences, also called prompts, act as structured data or context that guide this process. Through careful curation of prompts, by giving examples (n-shot prompting), asking questions (*Chain-of-Thought (COT)*) and giving task descriptions (role prompting), the context visible to the LLM can be enhanced to improve its output.

### B. Polynomial Formal Verification using BDDs

A BDD represents a Boolean function in the form of a directed, acyclic graph [11]. Positive (negative) function valuations are defined by a path from its root to a 1 (0) terminal node. A BDD is called ordered if every variable occurs at most once on each such path and is called reduced if it does not contain isomorphic subgraphs or redundant vertices. For *Reduced Ordered Binary Decision Diagrams (ROBDDs)*, both of the aforementioned properties hold, where the term BDD is commonly used synonymously. The size of a BDD depends on its variable ordering, where finding the optimal ordering w.r.t. BDD size is an NP-complete problem.

In the domain of formal verification, BDDs can be used for model checking as well as equivalence checking. The latter makes use of the canonicity property of ROBDDs, i.e. two circuits implementing a given Boolean function are equivalent, iff their ROBDDs are isomorphic. Even for linear final BDD sizes, the construction process may take exponential time in the size of the circuit. Hence, for a comprehensive complexity analysis of the verification, the whole construction process needs to be considered. Software implementations of this construction make use of *If-Then-Else (ITE)* [12] as the universal operator for all Boolean operations. Thus, PFV commonly regards the number of ITE calls required for BDD construction as a measure for verification complexity.

By proving polynomial time and space complexity bounds, PFV offers a crucial advantage over conventional formal verification methods, ensuring predictability on runtimes and feasibility ahead of running the verification itself. However, establishing these bounds requires a lot of manual effort, which is evident from the high number of equations in some past papers in this field [13] [14].

## III. Adding LLMs to the PFV Flow

To automate this manual effort, we propose a PFV workflow that integrates LLMs to prove polynomial complexity bounds. A schematic of this workflow is depicted in Fig. 1. Components ① through ④ are modeled after the existing PFV process, while components ⑤ and ⑥ are newly integrated for automation.

The workflow begins with the production of circuit definitions by the *Adder module* ①. It implements a variety of basic building blocks that can be used for the construction of different adder structures. Currently, the module supports three different adder architectures which have been studied extensively for PFV: *Ripple-Carry Adder (RCA)*, *Conditional-Sum Adder (CoSA)*, and *Carry-Lookahead Adder (CLA)* [13] [15] [16]. In future work, this list may be extended to more adder types, arithmetic or other combinational circuit designs.

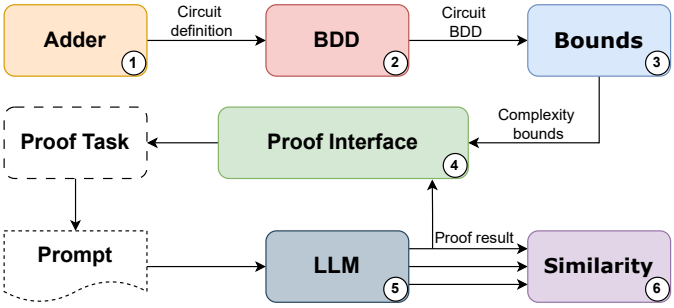


Fig. 1. PFV Workflow with the Proposed Framework

The given circuit definition, obtained from the adder module, is passed to the *BDD module* ②, which implements a small BDD package featuring a unique table for caching operations, Boolean operator interfaces, and some utility functions like size calculation. The central operation used internally during construction is the ITE operator, which all Boolean operations map against. The size of a BDD is very sensitive to its variable ordering, so for these adders we use the interleaved ordering, which is known to be efficient [17]. BDDs are constructed for the given adder circuit with different input widths (e.g., 8-bit, 16-bit, 32-bit, ...) and the number of ITE calls is recorded. These numbers together with the BDDs are passed to the *Bounds module* ③ which analyzes this data to establish bounds for the given adder circuit.

The bounds module is responsible for the calculation of complexity bounds for the considered circuit. It constructs a Z3 optimization problem that fits a polynomial to the given samples of BDDs for different input sizes. As an objective function, it first aims to minimize the slack of this polynomial over the provided bounds and then secondly tries to minimize the degree of the polynomial. The final solution of this optimization problem is a polynomial representing the complexity bounds, which is then passed to the *Proof Interface module* ④. In this module, a proof task in JSON format is created, which appends metadata, like an ID and the name of the circuit, to the bounds to be proven. This metadata enables a more straightforward evaluation of the final proof by the proof interface module. Once this process is completed, the LLM starts its work.

The task of the *LLM* ⑤ is to construct a formal proof of the polynomial bounds. We engineer a prompt by combining n-shot learning with schema-based prompting to maximize effectiveness. The construction of the prompt heavily depends upon the proof task but is universal w.r.t. the concrete circuit under consideration. A single prompt is used to generate the final proof result for all kinds of adder circuits. Listing 1 shows the outline of the prompt that we use for proof-generation. This prompt instructs the LLM to construct a proof for the Z3 Python [18] *Application Programming Interface (API)* and provides vital information in natural language. In the beginning of the prompt (Line 1), fundamental details about the expected response are outlined. This is followed by the context of the given task to establish a frame within which its intent and constraints are to be interpreted (Lines 3-4). After

```
1  Based on the information below, output one
       value: a single Python script for a
       proof engine. No prose, no extra
       characters - just the Python script.
2
3  Context (what you are proving)
4  <Context on the proof>
5
6  What the incoming Task JSON contains:
7  <JSON schema>
8
9  Your freedom (method)
10 You may use any sound Z3-based approach, for
       example:
11 <List of four example approaches>
12
13 Hard I/O contract for your script
14 <I/O contract for the proof script>
15
16 Output formatting requirements (very
       important)
17 Your entire response must be one valid
       Python script.
18 Do not add prose or any other text - only
       the Python script. Do not add any
       Markdown or other formatting.
19 The script must be self-contained (imports
       inside) and must not execute code at
       import time (no top-level I/O or prints).
```

Listing 1. Outline of the prompt provided to the LLM for proof generation

this information, the JSON schema of the proof task is defined, followed by some examples of possible proof methods (Lines 6-11). This guides the LLM toward efficiently solving the task at hand. Since the response of the LLM is expected to be a ready-to-execute Python script, it is vital that the I/O contract for the script is properly defined. We do this after providing all the context and examples related to the task (Line 13-14). In the end, we reinforce important details regarding the output of the LLM (Lines 16-19).

Once the response is received from the LLM, it is transformed into a JSON proof result and passed back to the proof interface. In this concluding iteration, the LLM-generated proof is executed with a circuit-specific proof task as its input to finally prove the given bounds. The proof interface also asserts whether the polynomial complexity bounds were successfully proven.

To evaluate the consistency of the proofs generated through the workflow, results of multiple proofs produced by the LLM may also be given to the *Similarity module* ⑥, which computes a score based on three metrics: *Abstract Syntax Tree (AST)* similarity, Z3 calls and solver states. The *AST Similarity* metric compares the AST of two different proofs for their structure, normalizing names such that variable naming does not affect the similarity score. The *Z3 Calls* metric statically collects and compares all calls made to the Z3 Python API. The *Solver States* metric employs monkey-patching [19] to dynamically track the internal states of the Z3 theorem prover during the proof process. The similarity module also offers a comprehensive similarity score, where these three metrics can be assigned custom weights to enable the evaluation of different consistency aspects.

TABLE I
COMPARISON OF LLM PROOF RESULTS ACROSS CIRCUITS.

| Circuit / Outcome | GPT-5.1 | GPT-5 | Gemini 3 | Qwen 3 |
|---|---|---|---|---|
| **RCA** | | | | |
| Proved | 7 | 9 | 7 | 0 |
| Failed | 0 | 0 | 3 | 3 |
| Error | 3 | 1 | 0 | 7 |
| **CoSA** | | | | |
| Proved | 7 | 9 | 8 | 0 |
| Failed | 0 | 0 | 2 | 3 |
| Error | 3 | 1 | 0 | 7 |
| **CLA** | | | | |
| Proved | 7 | 9 | 8 | 1 |
| Failed | 0 | 0 | 2 | 2 |
| Error | 3 | 1 | 0 | 7 |

**RCA**: Ripple-Carry Adder **CoSA**: Conditional-Sum Adder
**CLA**: Carry-Lookahead Adder

## IV. EVALUATION OF DIFFERENT LLMS

### A. Experimental Setup

For the purpose of evaluation, we generate proof tasks for three kinds of adder circuits: RCA, CoSA and CLA. We select five different models for evaluation, containing both open- and closed-weight LLMs. From the closed-weight category, we choose OpenAI's *GPT-5.1* [20], *GPT-5* [21], and Google DeepMind's *Gemini 3* [22]. Among the open-weight models, *Qwen3:coder* [23] by Alibaba Cloud and *Gemma 3* [24], also developed by Google DeepMind, are selected with $30b$ and $27b$ parameters, respectively. We use the default temperature and token size for all models and access them through their respective API using Python scripts. All experiments are performed on an Intel(R) Core(TM) i7 10700k 8-Core Processor with $3.8\,\mathrm{GHz}$, $64\,\mathrm{GB}$ of memory and an Nvidia(TM) A40 GPU. For each LLM, ten responses are generated using our engineered prompt. The LLM output is run for all three adders with the proof interface and the consistency of LLMs is evaluated by the similarity module.

### B. Experimental Results

Table I summarizes the outcomes for each combination of circuit and the four LLMs under consideration that produced viable results. Results for the adder circuits are divided into three categories: *Proved* indicates that the complexity bounds provided by the proof task were proven successfully by the proof result generated by the LLM. *Failed* means that the proof was executed but the bounds could not be proven. *Error* signifies that there was either a syntax error in the code or a runtime error occurred during execution.

Overall, GPT-5 was able to provide the most instances that proved complexity bounds, followed by Gemini 3. The performance of Gemini 3 was closely followed by the GPT-5.1 model. GPT-5 was able to prove the provided complexity bounds in nine out of the ten cases for all circuits, while for GPT-5.1, three of the generated proofs ran into an error, two of which were runtime errors. This effect may be attributed to

TABLE II
COMPARISON OF LLM CONSISTENCY METRICS.

| Metric | GPT-5.1 | GPT-5 | Gemini 3 | Qwen 3 |
|---|---|---|---|---|
| AST Similarity | 16.7% | 7.7% | 12.5% | **22.8**% |
| Z3 Calls | **29.0**% | 13.9% | **35.9**% | 22.2% |
| Solver States | 12.6% | **46.7**% | 20.8% | 0.0% |
| Similarity Score | 19.4% | 22.8% | **23.1**% | 15.0% |

the sample size, as statistical models are prone to fluctuations. Gemini 3 produced no syntactical errors in the generated code, but three (two) proof results were not able to prove the given complexity bounds for RCA (CoSA/CLA). The lack of syntax errors can be attributed to the Gemini 3's optimization for coding and agentic coding tasks.

The open-weight models we considered struggled to perform the given task, where Qwen3:coder managed to produce a single script that proved bounds for CLA. Gemma 3 did not produce any viable code and is therefore omitted from comparison in the table.

Table II shows the consistency of the four LLMs which generated any valid output. For each model, all three individual metrics are reported, as well as the total similarity score, where all metrics are assigned an equal weight. While the overall similarity score is within a comparable range for all models, it is interesting to see how they perform differently in the individual metrics. GPT-5 scores highest in similarity of solver states, but ASTs vary greatly. The AST similarity and Z3 calls metrics improve by more than twice in GPT-5.1, which may be attributed to a greater instruction adherence. GPT-5.1 and Gemini 3 are more balanced, producing more consistent Z3 API calls. The Gemini 3 model is targeted heavily for code-agentic task, which makes it score highest in the Z3 calls metric. Qwen3:coder scores best in AST similarity and also higher in Z3 calls, but the poor performance in solver states shows a lack of reasoning capabilities of the model.

## V. DISCUSSION

One major aspect that was observed during the experiments was the effect of creativity on the generation of code. While GPT-5.1 managed to score better in the similarity of the code structurally (attributed to instruction adherence), it scored lower in functional similarity. Creativity and less context adherence of GPT-5 also influence the verbosity of the output, i.e., producing more lines of code and thus reducing AST similarity compared to GPT-5.1.

Another aspect to note is that the performance of the open-weight models is still lacking by a huge margin compared to the closed-weight models, even from the same provider when comparing Google's Gemini 3 and Gemma 3 models. The performance of Gemma 3 models was similar across different parameters sizes ($4b$,$12b$ and $30b$ parameters). From the open-weight category, code-optimized models managed to perform slightly better (Qwen3:coder), as this optimization leads to better viability of code produced by the model.

## VI. RELATED WORK

Since the rise of LLMs, a variety of works have focused on their application in the domain of hardware design and especially verification [25]. The authors of [26] present a framework to manage LLMs for automated test stimuli generation, evaluating different LLMs and offering improved prompting techniques. In [27], the authors integrate coverage information into the process of test generation by giving the LLM direct access to the Verilog [28] code so it can reason about unexplored code branches.

The authors of [29] shift the focus toward more formal methods, introducing a framework which automatically generates assertions from complete specification documents using LLMs. The idea outlined in [30] goes in a similar direction, generating invariants and formal models to enable LLM-guided formal verification coupled with mutation testing. While all the aforementioned works address hardware verification from different angles, none of them attempts to reason about the complexity of their approach.

To the best of our knowledge, the only work that takes the complexity of formal verification into consideration is [8]. Here, human-readable proofs for PFV are generated with the help of LLMs. While some examples are provided, there is no connection to formal tools and natural language outputs still have to be validated manually. Our proposed framework aims to bridge this gap by offering a direct connection from LLM-generated proofs to formal verification tools.

## VII. CONCLUSION

In this work, we presented an open-source framework which integrates LLMs into the PFV workflow to assist in the generation of proofs for polynomial complexity bounds. We evaluated this workflow using five different LLMs (OpenAI's GPT-5.1 and GPT-5, Google's Gemini 3 and Gemma 3 and Alibaba's Qwen3:coder), assessing the quality and consistency of generated proofs for three adder circuit architectures (RCA, CoSA and CLA). The consistency of generated proofs was evaluated using three different metrics (AST similarity, Z3 Calls and Solver states). OpenAI's GPT-5 model was the best performing model, but Gemini 3 produced better scores across coding-related metrics. The closed-weight models we considered tied closely in performance, while the performance of open-weight models was severely lacking.

As future work, the workflow may be extended to prefix adders, multipliers and other circuits. Additionally, instead of BDDs, proving bounds for SAT-based verification may be an interesting area to investigate. Since LLMs are better when tasks are broken down, different prompting techniques like COT prompting can be evaluated for proof generation.

## ACKNOWLEDGEMENTS

This work was supported by the German Research Foundation (DFG) within the Reinhart Koselleck Project *PolyVer* (DR 287/36-1) and by the German Ministry for Research, Technology and Space (BMFTR) with project *ExaVerse* (grant number 01IW25003).

## REFERENCES


[1] G. Moore, "Cramming more components onto integrated circuits," *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82–85, 1998.

[2] M. Blum and H. Wasserman, "Reflections on the pentium division bug," *IEEE Transactions on Computers*, vol. 45, no. 4, pp. 385–393, 2002.

[3] E. Seligman, E. T. Schubert, and M. V. A. K. Kumar, *Formal verification: an essential toolkit for modern VLSI design*. Amsterdam Boston Heidelberg: MK, Morgan Kaufmann,an imprint of Elsevier, 2015.

[4] R. Drechsler, "Fast and exact is doable: Polynomial algorithms in test and verification," in *2022 IEEE 23rd Latin American Test Symposium (LATS)*, (Montevideo, Uruguay), p. 1–2, IEEE, 2022.

[5] M. Schnieber and R. Drechsler, "Automated polynomial formal verification using generalized binary decision diagram patterns," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 383, p. 20230390, 01 2025.

[6] R. Drechsler and M. Schnieber, "Next-generation automatic human-readable proofs enabling polynomial formal verification," in *Proceedings of the 21st ACM-IEEE International Conference on Formal Methods and Models for System Design*, MEMOCODE '23, (New York, NY, USA), p. 122–125, Association for Computing Machinery, 2023.

[7] O. Tafjord, B. Dalvi, and P. Clark, "ProofWriter: Generating implications, proofs, and abductive statements over natural language," in *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pp. 3621–3634, 2021.

[8] R. Drechsler, "Towards LLM-based generation of human-readable proofs in polynomial formal verification," *ArXiv*, vol. abs/2505.23311, 2025.

[9] "Z3 theorem prover." https://github.com/Z3Prover/z3. accessed 25-01-26.

[10] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[11] Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. C–35, p. 677–691, Aug. 1986.

[12] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient implementation of a BDD package," in *Conference proceedings on 27th ACM/IEEE design automation conference - DAC '90*, (Orlando, Florida, United States), p. 40–45, ACM Press, 1990.

[13] A. Mahzoon and R. Drechsler, "Polynomial formal verification of area-efficient and fast adders," in *2021 Reed Mulller Workshop (RM2021)*, 2021.

[14] A. Mahzoon and R. Drechsler, "Polynomial formal verification of prefix adders," in *2021 IEEE 30th Asian Test Symposium (ATS)*, (Matsuyama, Ehime, Japan), p. 85–90, IEEE, Nov. 2021.

[15] R. Drechsler, "PolyAdd: Polynomial formal verification of adder circuits," in *2021 24th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, (Vienna, Austria), p. 99–104, IEEE, Apr. 2021.

[16] L. Müller and R. Drechsler, "Polynomial formal verification parameterized by cutwidth properties of a circuit using Boolean satisfiability," *Microprocessors and Microsystems*, vol. 118, p. 105199, Nov. 2025.

[17] I. Wegener, *Branching Programs and Binary Decision Diagrams*. Society for Industrial and Applied Mathematics, 2000.

[18] G. Van Rossum and F. L. Drake, *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009.

[19] J. Hunt, *Monkey Patching*, pp. 487–490. Cham: Springer International Publishing, 2023.

[20] OpenAI, "GPT-5.1 instant and GPT-5.1 thinking system card addendum." https://cdn.openai.com/gpt-5-system-card.pdf. accessed 25-01-26.

[21] OpenAI, "GPT-5 system card." https://cdn.openai.com/gpt-5-system-card.pdf. accessed 25-01-26.

[22] Google, "A new era of intelligence with Gemini 3." https://blog.google/products/gemini/gemini-3/. accessed 25-01-26.

[23] Alibaba Cloud, "Qwen 3." https://github.com/QwenLM/Qwen3. accessed 25-01-26.

[24] Google DeepMind, "Gemma 3." https://deepmind.google/models/gemma/gemma-3/. accessed 25-01-26.

[25] M. Abdollahi, S. F. Yeganli, M. A. Baharloo, and A. Baniasadi, "Hardware design and verification with large language models: A scoping review, challenges, and open issues," *Electronics*, vol. 14, p. 120, Dec. 2024.

[26] 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," 2023.

[27] R. Ma, Y. Yang, Z. Liu, J. Zhang, M. Li, J. Huang, and G. Luo, "VerilogReader: LLM-aided hardware test generation," in *2024 IEEE LLM Aided Design Workshop (LAD)*, pp. 1–5, 2024.

[28] "IEEE standard for SystemVerilog–unified hardware design, specification, and verification language," *IEEE Std 1800-2023 (Revision of IEEE Std 1800-2017)*, pp. 1–1354, 2024.

[29] 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*, (Tokyo Japan), p. 614–621, ACM, Jan. 2025.

[30] M. Hassan, S. Ahmadi-Pour, K. Qayyum, C. K. Jha, and R. Drechsler, "LLM-guided formal verification coupled with mutation testing," in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1–2, 2024.