Prompt. Verify. Repeat. LLMs in the Hardware Verification Cycle

Muhammad Hassan[1,2], Mohamed Nadeem[1], Khushboo Qayyum[2], Chandan Kumar Jha[1], Rolf Drechsler[1,2]
[1]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
{hassan, mnadeem, chajha, drechsler}@uni-bremen.de khushboo.qayyum@dfki.de

***Abstract*—In this paper, we review the use of *Large Language Models* (LLMs) in the context of hardware verification, using the concept of a semantic layer to organize their role. We place LLMs within a centaur-style workflow and describe their operation as an iterative loop, *Prompt. Verify. Repeat.*. In this loop, an LLM is first prompted to generate or modify a verification artifact. The artifact is then evaluated using standard EDA tools and human review. Based on the results, the prompt or constraints are adjusted, and the process is repeated until the verification objectives are met. Within this structure, LLMs function alongside existing tools to support the verification process. Furthermore, we present a simple example of illustrating this concept in deductive reasoning in hardware verification as an application.**

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

I. Introduction

Advancements in semiconductor fabrication, as outlined by Moore's Law, have led to *System-on-Chips* (SoCs) and other *Integrated Circuits* (ICs) with tens of billions of transistors. As their functionality increases, pre-silicon hardware verification now consumes about 70% of development time and resources [1]. Errors detected after tape-out typically cannot be repaired through software updates; they instead require new silicon, resulting in additional cost and delays in *Time-to-Market*. Verification teams therefore combine simulation, formal property checking, emulation, and FPGA prototyping in multi-stage sign-off flows that uncover corner cases under tight schedules. The overall hardware verification process itself is typically structured into several stages to ensure thoroughness. However, the main difficulty stems from semantic translation. Requirements start as informal natural language, behavior is written in *Hardware Description Languages* (HDLs) like Verilog or SystemVerilog, verification intent appears in assertion languages like *SystemVerilog Assertions* (SVA) or *Property Specification Language* (PSL), stimulus is described in *Universal Verification Methodology* (UVM), and debug questions are asked against waveform traces. Each hand-off can introduce misinterpretation, and the volume of artifacts strains human attention.

This work was supported in part by the German Federal Ministry of Research, Technology and Space (BMFTR) within the project ECXL under contract no. 01IW22002, the project PaSVer under contract no. 16ME0855, the project SASPIT under contract no. 16KIS1852K, the project ECXLplus under the contract no. 01IW24001 and the project Scale4Edge under contract no. 16ME0127.

979-8-3315-2037-3/25/$31.00 ©2025 IEEE

Recently, the field of *Artificial Intelligence* (AI) has witnessed a transformative breakthrough with the advent of *Large Language Models* (LLMs). Prominent examples of foundational LLMs that have driven progress include the LLMs from OpenAI (e.g., GPT-3, GPT-4) [2], Llama models from Meta [3], Claude from Anthropic [4], DeepSeek models [5], and Gemini from Google [6]. LLMs have shown great promise in generating HDLs based on natural language descriptions, partial code snippets, or specific instructions [7]. Consequently, LLMs are now being investigated for their potential to assist in hardware verification by automatically generating assertions [8]–[10]. When generated accurately and consistently, these assertions contribute to verifying the correctness, completeness, and consistency of the hardware design; core principles of effective verification. As a result, they can significantly reduce the time and effort required. Several studies have shown that LLMs are capable of producing high-quality assertions in various HDLs, making them promising candidates for improving the verification process [11]–[13].

The increasing complexity of verification processes along with the development of LLMs is prompting a shift in the verification approaches. Traditional verification methods are facing growing limitations, leading the industry to explore alternative solutions. LLMs, which have shown effectiveness in processing code and complex text, are being considered as a possible next step. However, these models are not comprehensive world models. Their probabilistic token generation can invent signals, mislabel the design hierarchy, and produce behavior or structures that deviate from the intended hardware design unless guided by a golden reference design. However, their real value emerges when they are partnered with humans in a centaur workflow [14] (i.e., a human-AI collaborative process) serving as a semantic layer [15] that links requirements, RTL, and verification artifacts, while deterministic *Electronic Design Automation* (EDA) tools continue to supply definitive pass/fail evidence.

Contribution: In this paper, we review the LLMs from the semantic layer concept and position them within a centaur workflow for hardware verification. We frame the interaction as an iterative three-step loop *Prompt. Verify. Repeat*. First, prompt the LLM to generate or transform a verification artifact. Second, verify the artifacts with existing EDA tools and human inspection, and third, repeat the cycle, refining prompts or constraints until verification goals are met. This view positions LLMs as adaptable translators that operate beside existing EDA tools. We believe this will help researchers in EDA look at LLMs in a different way to develop their novel methodologies. Additionally, using this semantic layer concept, we show a simple illustrative example of using deductive reasoning in hardware verification.

The remainder of the paper is structured as follows: Section II discusses different existing related works integrating LLMs in the hardware verification flow. Section III discusses open challenges which are creating a bottleneck for LLM integration, Section IV presents our semantic layer approach and an illustrative example, and Section V discusses promising future directions.

II. LLMs for Verification

Several works have explored machine learning techniques for verification [16]–[18] in the past. However, in this section, we aim to highlight the research problems in verification where LLMs in particular have shown potential. Even though the use of LLMs initially in the domain of EDA was limited to the generation of the hardware description code, it has significantly advanced thereafter in the area of verification.

In [19], the authors were among the first to explore the use of LLMs in the verification process. They introduced a framework called *nl2sva*, which generates SVA for a given circuit based on a general natural language specification. Their methodology employs few-shot prompting, using *chain-of-thought* examples to enhance the quality of the generated assertions. Both human feedback and model checker outputs are utilized to help the LLM debug and refine its results.

In [20], the authors also addressed the challenge of translating natural language specifications into SVA. To address this issue, they demonstrated that the process could be automated using LLMs by following a three-step automated approach: (a) extracting relevant information for SVA generation from the specification document, (b) aligning terminology between the natural language specification and the HDL implementation, and (c) generating the SVA. Using this method, they achieved an 89% success rate in producing assertions that were both structurally and functionally accurate.

In [21], the authors present a two-step approach for generating SVA for OpenTitan designs. In the first step, design specifications are converted into a JSON file with a fixed format, which serves as input to the LLM. The high-level idea is to aid the LLM in understanding the specifications in a structured manner. In the second step, the assertions produced by the LLM are validated using the VCS simulation tool. The resulting log files are then used to iteratively refine the SVA as needed. Although fewer than 27% of the assertions required modification, the authors suggest that using domain-specific LLMs tailored for Assertion-Based Verification (ABV) could further enhance the quality of the generated assertions.

In [22], the authors reduced the number of iterations required for SVA generation by introducing a feedback loop. In this loop, both the Verilog design and the specification are provided as input to the LLM, enabling better synchronization of signal names between the design and the generated assertions. Additionally, manual prompts and error messages from the simulator are used to iteratively refine the SVAs.

In [23], the authors examined the quality of invariants generated by LLMs using both design specifications and Verilog code as input. To assess these invariants, they introduced mutations into the design using the Yosys tool and analyzed how effectively the generated invariants responded to these changes. The authors also shared several key observations from their experiments: (a) LLMs perform well with structured input data, consistent with findings in [21]; (b) positive reinforcement and monetary incentives can enhance LLM output quality; (c) when provided with a counterexample, LLMs were unable to repair or patch the design; and (d) LLMs struggled to comprehend flattened netlists.

In [24], the authors advocate for the use of domain-specific language models instead of general-purpose LLMs. They propose adapting a general model by fine-tuning it with VLSI-specific data to improve performance. For evaluation, they utilize metrics such as *pass@k* and BLEU score to compare their model against existing LLMs. Although their domain-adapted model outperformed others of similar size, GPT-4 remained the most effective in generating SVA from natural language specifications.

In [25], [26], the authors introduce a methodology for identifying and patching design bugs using LLMs in combination with *Retrieval-Augmented Generation* (RAG). They develop an iterative bug detection and fixing framework, where RAG enhances the LLM's context awareness during prompting. Five types of bugs were manually introduced into the OpenTitan design, and the LLM successfully patched four of them. The exception involved a bug caused by an incorrect constant value, which the LLM was unable to fix.

In [27], the authors tackle the issue of syntax errors in Verilog code generated by LLMs. Their approach combines RAG with a React-based prompting strategy—comprising steps of thought, action, and observation—to enhance correction quality. Their methodology corrected 98.5% of syntax errors. Additionally, GPT-4 alone achieved 98% correction accuracy with just one-shot prompting, but their method offers a more scalable solution for smaller models.

In [28], an iterative methodology is introduced to correct both syntactic and functional errors in RTL code. The approach employs two LLM-based agents, one dedicated to debugging and the other to evaluating the design's completeness and overall quality. The authors further enhance the correction process by fine-tuning the LLM and employing advanced prompting strategies like self-planning and role prompting.

In [29], the authors propose using a domain-specific LLM for hardware debugging. They build a training dataset from design defects and fixes extracted from version control systems of open-source repositories. A LLM is then trained on this data to detect and correct bugs. However, the authors observe variability in the performance of fine-tuned models, highlighting the need for further refinement in training methodologies.

In [30], the authors present the first comprehensive benchmark suite and evaluation framework for assessing the ability of LLMs to generate SVA. The benchmark comprises three test scenarios: (1) NL2SVA-Human, which evaluates LLM performance when provided with high-quality human-written testbenches and specifications; (2) NL2SVA-Machine, which tests LLMs on diverse natural language specifications; and (3) the most challenging, Design2SVA, which assesses whether an LLM can generate SVA using only RTL code.

In [31], the authors construct a large dataset of over 1,000 *High-Level Synthesis* (HLS) programs, each with up to 40 injected bugs. This dataset serves as a valuable resource for training LLMs on bug detection and correction in high-level synthesis designs.

In [32], the authors explore using LLMs for verifying both conventional RISC-V and neuromorphic processors. Their results show that LLMs can achieve up to 89% coverage. However, their methodology depends on human intervention, which limits scalability. To address this, they suggest automating certain steps, such as converting coverage reports into prompts to improve scalability.

In [33], the authors propose a multi-step process for using LLMs to generate formal proofs for verification. The process involves breaking the code into smaller modules, identifying simple modules, generating properties for them, and progressively connecting and verifying these modules in a hierarchical fashion. This iterative approach is especially suited for hierarchical designs. In [34], the authors use LLMs to generate auxiliary properties that support complex verification tasks. They also leverage counterexamples from induction-based methods to create assertions that aid in k-induction proofs.

In [35], the authors introduce SYNTHTL, a tool designed to translate natural language specifications of hardware behavior into *Temporal Logic* (TL) specifications. The approach addresses challenges such as ambiguity in natural language and the difficulty of validating potentially incorrect translations. SYNTHTL integrates LLMs, model checking, and human feedback in a loop. It decomposes complex specifications into simpler sub-statements, translates each into TL, and then combines them, reducing validation effort and enhancing tool-assisted verification of natural language inputs.

In [36], a comprehensive framework is proposed to support RTL code generation and verification from natural language instructions. The framework includes: (1) a benchmark for evaluating LLM performance in RTL code and assertion generation, (2) a large dataset of instruction-code pairs, and (3) a unified infrastructure for both training and fair evaluation of LLMs in RTL-related tasks.

Finally, in [37], the author explores the use of LLMs for generating human-readable proofs in the context of *Polynomial Formal Verification* (PFV). Using OpenAI's GPT-4o, the model is able to generate complete proof structures, including the base case, hypothesis, and inductive step, which can then be validated through deterministic reasoning engines.

From the prior works, it is clear that the LLMs indeed have immense potential for use in hardware verification. However, there exist some challenges which we discuss in the next section.

III. Key Challenges

LLMs have notable potential in hardware verification, but their broad and dependable use depends on addressing several technical and practical challenges. In this section, we briefly discuss a few of the challenges.

A. Data Scarcity for HDLs

The performance of LLMs is heavily dependent on the quality and quantity of their training data. A major bottleneck to developing highly proficient LLMs for hardware verification is the relative scarcity of large-scale, high-quality, publicly available datasets of HDL code, especially for specialized artifacts like complex SVA or comprehensive testbenches, when compared to general-purpose software programming languages. This data deficit is a primary contributor to poor LLM performance and the tendency to hallucinate when generating HDL-specific content. Much of the existing HDL code within corporations is proprietary and represents valuable intellectual property, making it challenging to use for training publicly accessible models. While fine-tuning open-source LLMs on smaller, curated HDL datasets is a viable approach, this process itself requires significant effort, computational resources, and domain expertise. In this regard, [30] introduced the first comprehensive benchmark suite and evaluation framework to assess the ability of LLMs in generating SVA. Similarly, initiatives like the VERT dataset [18], which provides open-source SVA represent crucial steps towards addressing this data gap. However, more such efforts are required at this end.

B. Scalability and Explainability

As SoCs continue to grow in size and complexity, the scalability of LLM-based verification solutions becomes a critical consideration. Integrating LLMs to verify designs containing billions of transistors and expansive state spaces presents significant challenges. These include the high computational resources required, the limited context windows of current LLMs that may be insufficient to capture an entire design or its documentation, and the overwhelming volume of information that must be processed and analyzed. Equally important is the issue of explainability. For verification engineers to trust and effectively utilize LLM-generated artifacts (e.g., SVA, test stimuli, bug patches), they often need to understand why the LLM produced a particular output or made a specific recommendation. The *black box* nature of many deep learning models, including LLMs, makes their internal reasoning processes opaque, which is a significant hurdle for adoption in safety-critical applications.

C. Evaluation Metrics and Benchmarking

Defining appropriate, robust, and comprehensive metrics to evaluate the quality, correctness, and practical effectiveness of LLM-generated verification artifacts is a non-trivial challenge.

Simple metrics like code similarity to a reference, or pass/fail rates on a limited set of tests, may not be sufficient to capture the true utility or potential pitfalls of an LLM-generated SVA, test case, or debug suggestion. Metrics currently employed include syntax correctness and functional correctness determined by formal property verification pass/fail status as outlined in Section II. There is a pressing need for standardized benchmarks and dedicated evaluation frameworks specifically designed for assessing LLMs in the context of hardware verification. A further consideration is the potential risk of *LLM-induced bugs*. If not managed and validated with extreme care, LLMs could inadvertently introduce new, subtle, and difficult-to-detect errors into the verification process itself, e.g., an incorrectly generated assertion might pass on a correct design but fail to detect a genuine design flaw, or, sneakily it might pass on a buggy design, thereby creating a false sense of security. Similarly, a poorly designed test case generated by an LLM might achieve superficial coverage metrics without actually stressing the critical functionalities or corner cases of the design. This implies that the outputs of LLMs used in verification must themselves be subjected to rigorous *meta-verification*. This might involve applying formal methods to verify properties of LLM-generated SVA. This adds another layer of complexity but is crucial for ensuring that LLMs genuinely enhance, rather than compromise, the integrity of the hardware verification process.

The combination of aforementioned challenges in addition to LLM's potential to hallucinate or produce functionally incomplete HDL creates a trust deficit that can be a major barrier in the adoption of LLMs in critical verification roles. Even if LLMs demonstrate promising results in controlled benchmarks, practitioners will be reluctant to rely on them for mission-critical tasks without stronger guarantees of reliability or clearer insights into their reasoning. Overcoming this trust deficit requires not only technically superior LLMs but also the development of robust methods for validating LLM outputs, explaining their derivations, and managing the associated risks. To address some of these challenges, we propose integrating LLMs as a semantic layer in the verification flow, as detailed in the next section.

IV. LLMs as Semantic Layer

A. Overview

Fig. 1 shows an overview of the semantic layer concept, *Prompt. Verify. Repeat.* to integrate LLMs in the hardware verification flow. The high-level idea is that LLMs should be used as a tool in conjunction with other EDA tools. First, the LLMs are prompted along with verification plans, natural language requirements, and the *Design Under Verification* (DUV) to perform a certain task. Additionally, the LLMs are provided with a schema. The LLMs are prompted to either parse the given information and provide some text, or they are prompted to synthesize the given information into SVA, bug patches, code refactoring and optimizations, or test stimuli, or they are prompted to explain certain behaviors in the given information. Hence, using the schema the LLMs

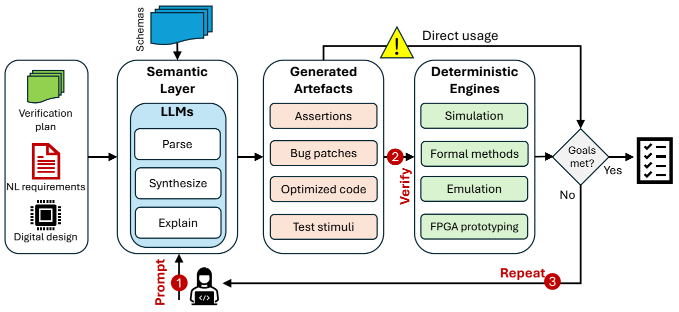

Fig. 1: Overview of using LLMs as a semantic layer in hardware verification.

are able to generate the verification artifacts. Due to the probabilistic nature of LLMs as outlined in Section I, the generated artifacts need to be verified for correctness. Hence, as a second step, the artifacts are verified using deterministic EDA tools (Deterministic Engines) and human inspection via simulation, formal methods, or emulation, etc. Once the artifacts are verified, it is checked if the goals have been met or not. If the goals are not met, the third step is started to repeat and refine the prompts or constraints until the verification goals are met.

In Fig. 1, we can see that LLMs inside the semantic layer are a small part of a bigger system. Please note, the generated artifacts may only be used directly to check if goals are met with extreme caution as the artifacts can be very inconsistent [38].

In the following subsection, using this concept of semantic layer, we show a rudimentary example of using deductive reasoning in hardware verification.

B. Deductive Reasoning in Hardware Verification

One of the essential tasks in artificial intelligence is to build reasoning systems that simulate human thinking in decision-making and inference based on given knowledge [39]. In this context, *Deductive Reasoning* [40], [41] is a central task that relies on a set of deduction rules to draw logically sound conclusions from explicitly given premises. One of the intuitive deduction rules is *Modus Ponens*, which performs a single reasoning step (i.e., deriving a direct conclusion $\mathcal{G}$ from a given fact $\mathcal{F}$ and an implication "If $\mathcal{F}$, then $\mathcal{G}$"). This is formulated as follows:

$$\frac{\mathcal{F} \qquad\qquad \mathcal{F} \rightarrow \mathcal{G}}{\mathcal{G}}$$

For multi-step reasoning, each observed conclusion is treated as a fact, and deduction rules are applied to derive further conclusions.

In order to enable automated reasoning, formulating the given knowledge into a formal logic is essential. Therefore, *Answer Set Programming* (ASP) [42]–[44], a well-known declarative programming framework from the area of knowledge representation and non-monotonic reasoning, has been introduced. ASP enables the description of given knowledge using a first-order logic-based representation (called an ASP

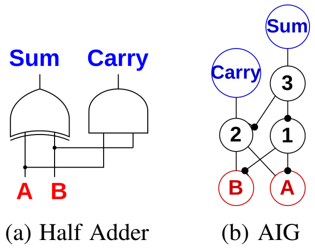

Fig. 2: Half adder block diagram and its corresponding AIG graph.

program). It consists of a set of rules, facts, and constraints such that facts represent information that is valid within the knowledge, and constraints express information that is negated or considered invalid. Rules represent implications that allow conclusions to be derived. To enable performing deductive reasoning, *Stable Model* semantics [45], [46] is used to compute the intended models of the program (called *Answer Sets* (AS) of the program).

In the EDA field, ASP has been successfully employed to enable *Polynomial Formal Verification* (PFV) [47], [48] of circuits with constant cutwidth [49], [50], where cutwidth is defined as the maximum number of intermediate nodes that each output bit relies on. PFV is performed by translating the *And-Inverter Graph* (AIG) representation of the circuit design into a set of rules and the high-level specifications as constraints within an ASP program Π (i.e., as the logical complements of the specifications, capturing the violation conditions).

Consider the AIG graph of Figure 2(b) constructed w.r.t. the half adder of Figure 2(a). The program Π can be constructed as shown in Listing 1, where Π is in *smodels* format accepted by the *Clingo* solver [51] (i.e., :-, "x^1", and & correspond to "$\leftarrow$", $\neg x$, and "$\wedge$", respectively). The specifications of the sum and carry (labeled as ω) are added to the program Π. The ASP program Π is then passed to an ASP solver to check whether it has an answer set (i.e., whether the program is consistent) under each possible input vector. If the program has no answer set (i.e., Π is inconsistent), the design matches the specification functions. Otherwise, the design is incorrect, and a counterexample that violates the specifications is obtained.

Listing 1: Π constructed w.r.t. the AIG of Figure 2(b).

$$\begin{aligned}
&\Pi = \{and1(X)\text{:-}a(Y), b(Z), X = Y\&Z.\\
&and2(X)\text{:-}a(Y), b(Z), X = (Y\hat{}1)\&(Z\hat{}1).\\
&and3(X)\text{:-}and1(Y), and2(Z), X = (Y\hat{}1)\&(Z\hat{}1).\\
&carry(X)\text{:-}and2(X).\\
&sumOut(X)\text{:-}and3(X).\}
\end{aligned}$$

However, the process of translating both the AIG and the specification functions into an ASP program is manual, making it time-consuming and error-prone. Additionally, the specification functions may be described in natural language, which makes them harder to translate into an ASP program. Therefore, LLMs have gained interest for automating the translation process due to their capability to process complex natural language.

Listing 2: Single-shot Prompt and Guided-LLM Prompting.

```
% Single-shot Prompting:
Please write ASP rules for a half adder such that
     the carry is 0 if both inputs A and B are
    not 1. The carry is equal to 1 if inputs A
    and B are 1. The sum is equal to 1 if inputs
    A and B are not equal. The sum is equal to 0
    if inputs A and B are equal.

% Guided-LLM Prompting:
Step 1): Please re-formulate the following text as
    "If-Then" statements:
The carry is 0 if both inputs A and B are not 1.
    The carry is equal to 1 if inputs A and B are
     1. The sum is equal to 1 if inputs A and B
    are not equal. The sum is equal to 0 if
    inputs A and B are equal.
Step 2): Please write "If-Then" statements as ASP
    rules such that the "If part" appears in the
    body of the rule, and the "Then part" appears
     as the head of the rule. Please don't add
    extra information. Please, represent the
    values of the carry, sum, and inputs A and B
    using the predicates carry(X), sum(X), a(X),
    and b(X), respectively, where X denotes the
    corresponding bit value.
```

Relying on the LLM single-shot prompting to encode specifications into ASP introduces numerous errors, including both syntactic and semantic ones. This necessitates manual analysis and correction by domain experts. However, guiding the LLM through the encoding process significantly improves the quality of the resulting encoding.

Consider the prompts of Listing 2. Performing single-shot prompting produced syntactical errors and a mismatch with the AIG encoding. However, breaking the translation process into two steps (i.e., **Step 1** and **Step 2**) has a significant improvement in the encoding. More precisely, LLM was able to encode syntactically and semantically correct ASP rules. This enables using LLM in the field of formal verification and automating the encoding process.

V. Promising Directions

The rapid development of LLMs in the context of hardware verification suggests a future in which LLMs play a central role in ensuring design correctness. A practical and sustainable approach is to integrate LLMs as aids to humans, rather than autonomous replacements as motivated in Section I. This synergistic approach leverages the strengths of both humans (creativity, critical reasoning, domain knowledge) and LLMs (speed, pattern recognition, handling vast data). In this regard, there are several promising directions where LLMs can be used effectively. A critical area is the development of explainable AI methods tailored for LLM applications in verification. Verification engineers need to understand the rationale behind an LLM's suggestions, i.e., why a particular SVA was generated, why a test stimuli was deemed important, or how a bug patch was derived. Making LLM decisions more transparent and interpretable is essential for building trust and enabling effective debugging of the LLM's own reasoning processes. Another promising direction is fine-tuning open-source LLMs on tailored proofs data to improve their accuracy in generating human-readable proofs. For instance, LLMs might be trained to generate formal proofs for symmetric and partially symmetric functions, which are common in hardware designs but challenging to verify.

VI. Conclusion

The use of LLMs in hardware verification is an emerging area within EDA. Rather than replacing human experts, LLMs are expected to serve as helpful assistants by handling routine tasks such as generating testbench code, drafting assertions, analyzing simulation logs, and supporting debugging. LLMs show potential in improving various aspects of the verification process, including assertion-based and simulation-based methods, test generation, and translation of natural language specifications into formal representations. These capabilities could streamline workflows and reduce manual effort. However, concerns persist around the accuracy, reliability, and transparency of LLM outputs, limited domain-specific training data, and integration and security challenges. Progress will depend on collaboration between AI and EDA experts to develop better datasets, improve model training, ensure transparency, validate results, and adapt tools for effective LLM use.

References

[1] H. Shin, "Efficient bug discovery with machine learning for hardware verification," *Retrieved March*, 2022. [Online]. Available: https://community.arm.com/arm-research/b/articles/posts/efficient-bug-discovery-with-machine-learning-for-hardware-verification

[2] OpenAI, "Openai." [Online]. Available: https://openai.com/

[3] Meta, "Llama by meta." [Online]. Available: https://www.llama.com/

[4] Anthropic, "Anthropic." [Online]. Available: https://claude.ai

[5] A. Liu *et al.*, "Deepseek-v3 technical report," *arXiv preprint arXiv:2412.19437*, 2024.

[6] G. Inc., "Google inc." [Online]. Available: https://gemini.google.com/app

[7] S. Thakur *et al.*, "Autochip: Automating hdl generation using LLM feedback," *arXiv preprint arXiv:2311.04887*, 2023.

[8] R. Kande *et al.*, "LLM-assisted generation of hardware assertions," *arXiv preprint arXiv:2306.14027*, 2023.

[9] M. Orenes-Vera *et al.*, "From RTL to SVA: LLM-assisted generation of formal verification testbenches," *arXiv preprint arXiv:2309.09437*, 2023.

[10] C. Sun *et al.*, "Towards improving verification productivity with circuit-aware translation of natural language to systemverilog assertions," in *DAV*, 2023.

[11] D. Saha *et al.*, "LLM for SoC security: A paradigm shift," *arXiv preprint arXiv:2310.06046*, 2023.

[12] X. Meng *et al.*, "Unlocking hardware security assurance: The potential of LLMs," *arXiv preprint arXiv:2308.11042*, 2023.

[13] B. Ahmad *et al.*, "Fixing hardware security bugs with large language models," *arXiv preprint arXiv:2302.01215*, 2023.

[14] S. Saghafian *et al.*, "Effective generative ai: The human-algorithm centaur," *arXiv preprint arXiv:2406.10942*, 2024.

[15] S. Hoseini *et al.*, "A survey on semantic data management as intersection of ontology-based data access, semantic modeling and data lakes," *Journal of Web Semantics*, vol. 81, p. 100819, 2024.

[16] R. Drechsler *et al.*, "Formal specification level: Towards verification-driven design based on natural language processing," in *Proceeding of the 2012 Forum on Specification and Design Languages*. IEEE, 2012, pp. 53–58.

[17] M. Soeken *et al.*, "Automating the translation of assertions using natural language processing techniques," in *Proceedings of the 2014 Forum on Specification and Design Languages (FDL)*, vol. 978. IEEE, 2014, pp. 1–8.

[18] A. Menon *et al.*, "Enhancing large language models for hardware verification: A novel systemverilog assertion dataset," *arXiv preprint arXiv:2503.08923*, 2025.

[19] 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.

[20] W. Fang *et al.*, "AssertLLM: Generating hardware verification assertions from design specifications via multi-LLMs," in *2024 IEEE LLM Aided Design Workshop (LAD)*. IEEE, 2024, pp. 1–1.

[21] 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.

[22] 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.

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

[24] 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)*. IEEE, 2024, pp. 1–4.

[25] 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)*. IEEE, 2024, pp. 1–5.

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

[27] 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.

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

[29] 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.

[30] M. Kang *et al.*, "FVEval: Understanding language model capabilities in formal verification of digital hardware," *arXiv preprint arXiv:2410.23299*, 2024.

[31] 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)*. IEEE, 2024, pp. 435–441.

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

[33] 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*, 2024, pp. 1–2.

[34] A. Kumar *et al.*, "Generative ai augmented induction-based formal verification," in *2024 IEEE 37th International System-on-Chip Conference (SOCC)*. IEEE, 2024, pp. 1–2.

[35] D. Mendoza *et al.*, "Translating natural language to temporal logics with large language models and model checkers," in *2024 Formal Methods in Computer-Aided Design (FMCAD)*, 2024, pp. 1–11.

[36] 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*, 2024, pp. 1–9.

[37] R. Drechsler, "Towards LLM-based generation of human-readable proofs in polynomial formal verification," *arXiv preprint arXiv:2505.23311*, 2025.

[38] K. Qayyum *et al.*, "LLMs for hardware verification: Frameworks, techniques, and future directions," in *2024 IEEE 33rd Asian Test Symposium (ATS)*, 2024, pp. 1–6.

[39] J. McCarthy, "Programs with common sense," in *Proceedings of the Teddington Conference on the Mechanization of Thought Processes*, 1959, pp. 75–91.

[40] L. J. Rips, "Two kinds of reasoning," *Psychological Science*, vol. 12, no. 2, pp. 129–134, 2001.

[41] M. D. S. Braine, "On the relation between the natural logic of reasoning and standard logic." *Psychological Review*, vol. 85, pp. 1–21, 1978. [Online]. Available: https://api.semanticscholar.org/CorpusID:120217462

[42] V. W. Marek *et al.*, "Stable models and an alternative logic programming paradigm," *A Computing Research Repository*, 1998.

[43] I. Niemelä, "Logic programs with stable model semantics as a constraint programming paradigm," *Ann. Math. Artif. Intell.*, vol. 25, no. 3-4, pp. 241–273, 1999.

[44] M. Gebser *et al.*, "Conflict-driven answer set solving: From theory to practice," *Artificial Intelligence*, 2012.

[45] M. Gelfond *et al.*, "The stable model semantics for logic programming," in *Proceedings of International Logic Programming Conference and Symposium*, 1988, pp. 1070–1080.

[46] M. Gelfond *et al.*, "Classical negation in logic programs and disjunctive databases," *New Generation Computing*, pp. 365–385, 1991.

[47] R. Drechsler *et al.*, "Polynomial formal verification: Ensuring correctness under resource constraints," in *ICCAD*, 2022, pp. 70:1–70:9.

[48] R. Drechsler, "PolyAdd: Polynomial formal verification of adder circuits," in *DDECS*, 2021, pp. 99–104.

[49] M. Nadeem *et al.*, "Polynomial formal verification exploiting constant cutwidth," in *Proceedings of the 34th International Workshop on Rapid System Prototyping*. Association for Computing Machinery, 2024.

[50] M. Nadeem *et al.*, "Polynomial formal verification of sequential circuits using weighted-aigs," in *2025 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2025.

[51] M. Gebser *et al.*, "Advances in gringo series 3," 2011, pp. 345–351.