LLMs for Hardware Verification: Frameworks, Techniques, and Future Directions

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

***Abstract*—Large Language Models (LLMs) have gained immense popularity and are being explored for use in several domains. In this paper, we describe the LLMs for their use in the *Electronic Design Automation (EDA)* domain specifically for *hardware verification*. LLMs are being rapidly explored for hardware design generation and verification. However, given the inherent non-determinism and the limited capabilities of the current LLMs, the designs may contain bugs or the generated properties could be incorrect. The process of manually checking these bugs can be tedious and time-consuming. This highly limits the applicability of the LLMs. Researchers are looking to alleviate these limitations by incorporating verification strategies using the LLMs and enhancing the LLMs' capabilities not only for bug-free design generation but also stand-alone for verification. The current pace of development in these can cause many areas to be overlooked. Therefore in this work, we discuss the state-of-the-art tools and frameworks available for utilizing LLMs for EDA. We will then discuss the hardware verification techniques being explored using LLMs. We then discuss the consistency of the natural language properties generated using LLMs. Lastly, we will discuss the future directions in which the LLMs can aid in the hardware verification process.**

***Index Terms*—Large Language Models, Verification, Verilog, System Verilog, Assertions-based Verification, VHDL, Electornics Design Automation.**

I. Introduction

LLMs are advanced AI models designed to understand and generate human-like text [1], [2]. Trained on vast datasets, these models can recognize patterns, understand context, and identify semantic relationships within data. Their ability to generate coherent text has significantly influenced fields like *Natural Language Processing (NLP)* and AI-driven communication. A key feature of LLMs is their capability to perform zero-shot and few-shot learning, meaning they can provide accurate responses or generate relevant content when faced with new tasks or minimal examples. This adaptability and versatility make LLMs valuable across various domains, including EDA [3]–[7].

In the context of EDA, where complex design flows, verification, and optimization tasks require precision and expert knowledge, LLMs have the potential to streamline processes, reduce human error, and increase productivity throughout the IC design and verification cycle. By integrating LLMs in EDA, the goal is to automate repetitive tasks, provide insights, and optimize workflows in the design verification process [8]–[10].

There has been a lot of interest in various domains related to the utility of the LLMs. However, the LLMs are growing rapidly and the use case domains can be very different from that of the NLP. This can lead to cases where existing methodologies are re-implemented, increasing the time required to integrate the LLMs for the required tasks. In addition to this, the methodologies used in different works vary a lot leading to more ambiguity. In this paper, we make efforts to show a typical workflow of utilizing LLMs in the domain of EDA [9], [11], [12]. We also clearly highlight different tools that are used in the different stages of the workflow [13], [14]. We believe this will help researchers in EDA not only to understand the existing works but also to develop their methodologies more efficiently.

Several works have tried to use LLMs in the domain of EDA. These works encompass different aspects of the EDA including design, synthesis, verification, and testing [3]–[7]. The works vary from using standalone LLMs via the web browser to using frameworks like LangChain for the desired purpose. In this work, we dive deeper into the use of LLMs for hardware verification. We discuss the works ranging from formal verification by proof generation to generating assertions and detection of bugs [8], [9], [11], [15], [16]. We also discuss one of our prior works that iteratively detects bugs in a hardware design.

Since there are several *Hardware Description Languages (HDLs)* the LLMs may be better at generation assertions for one hardware language than the other, i.e., the consistency in the quality of properties generation can vary widely. We show some preliminary results obtained using ChatGPT-4 on simple arithmetic circuits from the ELAU benchmarks [1], [17]. The natural language-based properties varied widely for different HDLs. We believe that this will worsen when the designs are more complex. Lastly, we discuss some potential directions that can be explored to make the LLMs more consistent. Overall, we believe that this paper can serve as a reference for the researchers who are looking to explore LLMs for hardware verification and be a stepping stone to creating methodologies and frameworks in this particular area.

As a familiarization with the organization of the paper, in Section II, we discuss the tools and frameworks available for the LLMs. In Section III, we discuss the related works that use LLMs for verification. In Section IV, we discuss the results when properties are generated using the LLMs for different languages. In Section V, we discuss the future directions that can be explored using LLMs. We conclude the paper in Section VI.

II. Tools and Frameworks for LLMs

To maximize the effectiveness of LLM-based applications, developers can leverage existing frameworks and tools, which eliminates the need to reinvent the wheel. Fig. 1 illustrates the common architecture used to build LLM applications, incorporating off-the-shelf tools such as cosine similarity and semantic search techniques. Several design patterns, such as in-context learning, *Chain-of-Thought* (CoT) reasoning, and *Retrieval Augmented Generation (RAG)*, can be utilized to expedite development. These patterns are essential for tailoring LLM applications to specific use cases and improving response accuracy.

Two widely used frameworks that support rapid LLM-based development are LangChain [13] and Haystack [14]. These frameworks simplify integration with external APIs, custom tools, and memory management, making them suitable for both prototyping and production-level applications. Moreover, agentic approaches, such as those seen in AutoGPT [18], offer opportunities for LLMs to perform tasks based on high-level user goals autonomously, further expanding their utility in EDA.

Each LLM-based application can be divided into three stages (shown in Fig. 1): *i) data pre-processing and embedding*, *ii) prompt construction and information retrieval*, and *iii) LLM inference*.

In the first stage of *data pre-processing and embedding*, data is broken down into manageable chunks, passed through an embedding model, and stored in a vector database. For EDA applications, contextual data includes design specifications in text documents, PDFs, HDL files, etc. Tools like *Databricks* [19] and *Airflow* [20] can manage the data-loading and transformation processes. At the same time, frameworks like Haystack and LangChain offers built-in utilities (e.g., Unstructured [21]) to streamline data handling [22]. However, given the syntax complexity of HDL files and the LLMs' context window limitations, the data must be split into smaller chunks, often based on tokens, keywords, or semantic meaning [23]. This stage relies heavily on embedding models, which convert text into vector representations, allowing for mathematical operations such as semantic search. OpenAI's embedding models are widely used due to their ease of use, performance, and cost-effectiveness. However, other options, such as Cohere [24] and Sentence Transformers from Hugging Face [25], provide both commercial and open-source alternatives. While most embeddings are general-purpose, custom embeddings for EDA-specific use cases are lacking. The embeddings are stored in a vector database, essential for efficiently comparing and retrieving large volumes of data [26]. Popular vector databases include Pinecone [27], Qdrant [28], Weaviate [29], Chroma [30], Faiss [31], and pgvector [32].

In the second stage (*prompt construction and information retrieval*), a user query is transformed into a well-structured prompt, which is then submitted to the LLM. This process typically involves constructing a prompt template, providing few-shot examples (valid output examples), and including relevant documents retrieved from the vector database or external EDA tools. Crafting prompts have become increasingly sophisticated as developers aim to improve LLM response accuracy, particularly in production environments. The frameworks LangChain, Haystack, and LlamaIndex [33] are instrumental in this stage, abstracting the complexities of prompt construction, API integration, and contextual data retrieval. These frameworks maintain memory across multiple LLM calls and provides ready-made templates for common applications. The output of this stage is a prompt (or series of prompts) ready for submission to the LLM. These tools ensure integration with external EDA Tools like Yosys [34], Jasper Gold, and ABC [35] when necessary.

In the third stage (*LLM Inference*), the prompts are submitted to the LLM for inference [36]. OpenAI's *GPT-4* [1] series is widely used, but alternatives such as Google's Gemini [37], Anthropic's Claude [38], and Meta's LLaMA models [2] are also gaining traction. Meta's LLaMA models, in particular, have set a new benchmark for open-source LLMs in terms of accuracy and performance, making them viable alternatives for certain applications.

Once the inference is done, responses are logged or cached using tools such as Redis [39], SQLite [40], or GPTCache [41] to optimize future interactions and save computation time for frequently accessed data. This caching mechanism is vital in high-performance environments where the same queries or tasks might need to be revisited. Additionally, tools like MLflow [42], Helicone [43], and PromptLayer [44] can be integrated for logging and evaluation, ensuring that LLMs' performance and outputs are tracked, refined, and improved over time.

Finally, validation of LLM outputs is crucial, especially in EDA. Validation and guardrail tools such as Rebuff [45], LMQL [46], and Guidance [47] play a pivotal role in ensuring the reliability and consistency of LLM responses, acting as an additional layer of checks and safeguards before the final output is accepted or used. The tools and frameworks mentioned here represent the starting point for integrating LLMs into EDA verification processes. They enable rapid development and deployment without sacrificing accuracy.

In the following sections, we will explore specific applications of LLMs within the EDA domain, followed by a proposed methodology for consistency evaluation in LLM-assisted verification workflows.

III. LLM Based Techniques for Verification

With the ever so growing body of works that utilize and exploit the capabilities that LLMs offer, various new and promising directions open up in which LLMs aid EDA [48], [49] by reducing human error, or acting as an additional source of information in specification, coding, verification, and documentation. Moreover, researchers investigate the aspect of security as part of the EDA process [3]–[7]. As EDA consists of utilizing various tools, scripts, and programming languages, LLMs help reduce human error in tedious and tailored tasks when setting up and interconnecting EDA tools [50]–[52].

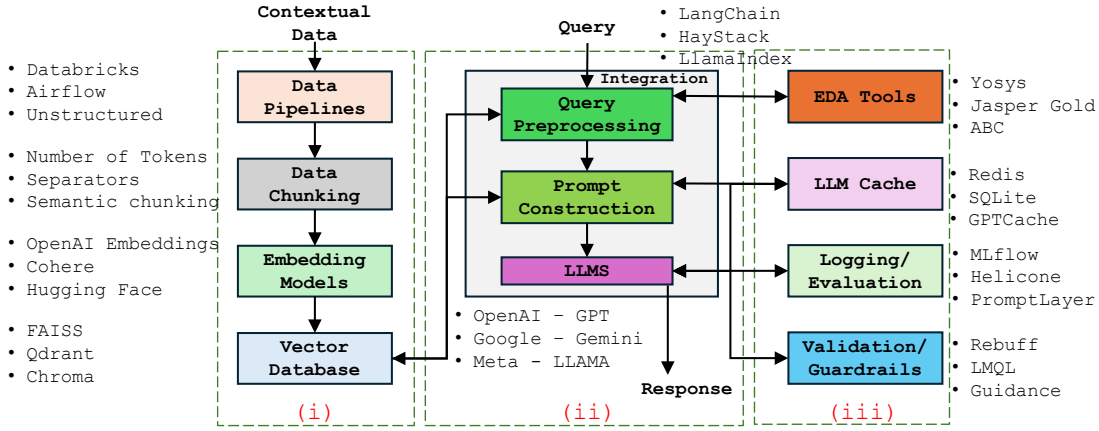

Fig. 1. LLM Application Architecture

One emerging body of works explores the generation of *Register-Transfer Level (RTL)* as HDLs in either Verilog/*SystemVerilog (SV)* or VHDL [53]–[58]. Here, some works focus on the generation of code from *Natural Language (NL)* specification [53], [59], while others explore aiding engineers in improving code or fixing bugs and errors [9], [60]. Works such as [5], for example, also present how the security domain approaches fixing security-related bugs in RTL codes.

Another direction of the research of LLM-aided EDA explores the verification process. Next, generating the code for regular [53], [56], [61] test benches, LLMs show promising results for test case generation [62], [63], stimuli generation for fuzzing-like environments [64] and the generation of properties and assertions for formal verification [8], [10], [11], [15], [16], [65], [66]. Lastly, a few works explore the design understanding as in the specification, documentation, and analysis/reverse engineering of hardware designs [67], [68]. Tasks in which LLMs handle documents that are formulated in NL, such as specifications, benefit from the characteristics of generative AI, as reformulating or rephrasing specifications can immensely benefit the design understanding. In many of these works, authors present curated datasets to benchmark the specific EDA task with their LLM-aided approach [55], [59], [69] or identify how *Intellectual Property (IP)* from projects like OpenTitan can be utilized [5], [8], [9], respectively.

In this work, we want to highlight the works dedicated to test and verification of hardware. Unlike the generation of RTL code, in the domain of testing and verification, LLMs can aid in not only generating code but in exploring and analyzing the design. LLMs can combine specifications and additional artifacts to identify undiscovered test patterns and unchecked properties or help in understanding counterexamples provided by tools performing formal verification.

In [8], [15], the authors present LLM-based frameworks, that can generate *SystemVerilog Assertion (SVA)* from NL specifications of a *Hardware (HW)* design. The frameworks systematically break the specifications down into a specified format, categorizing the information into parts like parameters, functional requirements, timing requirements, and more. This preprocessed format is utilized to generate the SVA-based formal properties of the system that are collected in the respective formal test bench. Moreover, the framework integrates the checking of properties through model checking and feeds back the bugs and error logs into the LLM to either fix the bug or refine the property. The authors perform their evaluation with the OpenTitan IPs, and report results for `GPT-4-turbo`.

In [16], the authors present an LLM-guided formal verification methodology, to generate Z3-based properties from design specifications and code. Here, the specifications are utilized as the basis for the properties and invariants, while the *Design under Verification (DUV)* is converted into a Z3-based formal model with the help of LLMs. The authors perform a mutation-based evaluation of the methodology, by comparing how well the generated properties cover different mutations introduced into the DUV. As a basis for the evaluation, the ISCAS-85 benchmark circuits [70] were used.

In [11], the authors present AssertLLM, a methodology to generate a wide range of SVA properties from specification documents. The AssertLLM workflow contains separate steps that are meant to decompose the different tasks in generating the SVA from the specifications, by guiding the LLM to split respective signals, instructions, and how they map within the specification. With the help of RAG, the decomposed information for each signal is transformed into respective SVA properties, which are checked against the golden RTL design, to evaluate the quality of the generated properties.

In [65], the authors present a domain-adapted LLM for the design and verification of *Very Large Scale Integration (VLSI)* systems. While the authors present an LLM adapted for the needs and requirements of VLSI design and verification, the case study is particularly focused on the generation of SVA properties for formal verification. The authors evaluated a set of benchmark circuit designs to determine the capabilities of generating formal properties for different systems. Moreover, the authors include a comparison against state-of-the-art LLMs to show how well the domain-adapted LLM performs.

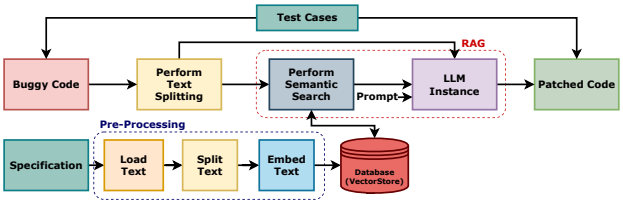

Fig. 2. Bug identification and patching with LLMs and RAG

The authors of [66] propose their nl2sva framework, a circuit-aware translation of NL to SVA. By utilizing in-context learning, the authors provide more meaning to the general specification statements and provide a feedback loop with a model checker that provides information about the tested property for the LLM and user. Moreover, the authors adopt the popular few-shot prompting [71] and chain-of-thought [72] techniques for LLMs to improve their LLMs' results. Lastly, the authors of [10] present a methodology that assists in incrementally generating formal properties and proofs for the Z3 solver. By breaking the verification process down into the hierarchical components of the DUV, the LLM first generates properties for the units, and later the properties for the system integration. Hence, by building more context for the LLM, the properties can cover more and more of the integrated system. In their case study, the authors identify various properties of *Ripple Carry Adder (RCA)* and *Dadda Tree Multiplier (DTM)* that are based on half adders and full adders.

To give an example of an approach that uses LLMs to aid in verification related tasks, we present the methodology mentioned in [9]. Fig. 2. shows the block diagram of the methodology, in which the authors use LLM for locating bugs in a given HDL code and patching it. The methodology works in three stages in which the first two stages are obligatory while the third stage only sets in motion when the second ones fail to provide a necessary outcome. In the first stage, i.e., the pre-processing stage, a database is created and data is split and stored in the database. This allows the management of unstructured data and also that the data will fit the context window of the LLM instance. In the second stage of semantic searching, the buggy line of code is extracted from the original line of code, and with the help of RAG relevant data is extracted from the database. This data is given to the LLM instance to assess and locate the bug within the given code and rectify it. If the bug is not identified and corrected in the second stage, the third stage (of bug identification and closure) is initiated, where an iterative process is performed to help the LLM to narrow the context and reach the desired outcome. After every iteration, the buggy code is replaced by the solution and test cases are performed on the patched circuit to assert if the bug has been eliminated. If the test cases fail, the methodology proceeds to the next step. The methodology was tested on three OpenTitan IPs with 5 different mutation types. The methodology was able to correct all bug types except for constant mutation.

TABLE I
CONSISTENCY OF LLM-GENERATED FUNCTIONAL PROPERTIES ACROSS SYSTEMVERILOG, VERILOG AND VHDL FOR THE ELAU BENCHMARKS.

Languages	Consistency	AddMulSgn	DivArrSgn	SqrSgn	SubVZ	IncGrayC
SV-V	min	0.101	0.358	0.652	0.67	0.272
	max	0.885	0.963	0.896	1	0.703
	avg	0.617	0.689	0.755	0.822	0.551
	% > 0.75	50%	50%	28%	66%	0%
	% > 0.80	50%	33%	28%	50%	0%
SV-VHDL	min	0.336	0.404	0.627	0.649	0.208
	max	0.471	0.916	0.896	0.963	0.834
	avg	0.381	0.653	0.742	0.782	0.569
	% > 0.75	0%	33%	28%	50%	16%
	% > 0.80	0%	33%	14%	50%	16%
V-SV	min	0.272	0.588	0.542	0.426	0.448
	max	0.885	0.962	0.896	1	0.703
	avg	0.655	0.753	0.708	0.723	0.598
	% > 0.75	50%	50%	28%	66%	0%
	% > 0.80	50%	33%	28%	50%	0%
V-VHDL	min	0.353	0.564	0.572	0.482	0.529
	max	0.751	0.913	0.891	0.963	0.792
	avg	0.529	0.772	0.736	0.731	0.649
	% > 0.75	25%	66%	28%	50%	16%
	% > 0.80	0%	66%	28%	50%	0%
VHDL-SV	min	0.331	0.515	0.294	0.721	0.368
	max	0.471	0.916	0.896	0.963	0.835
	avg	0.397	0.652	0.579	0.830	0.594
	% > 0.75	0%	33%	14%	66%	16%
	% > 0.80	0%	33%	14%	50%	16%
VHDL-V	min	0.364	0.411	0.332	0.791	0.363
	max	0.751	0.913	0.891	0.963	0.793
	avg	0.517	0.694	0.639	0.862	0.560
	% > 0.75	25%	66%	28%	83%	33%
	% > 0.80	0%	66%	28%	66%	0%

Overall, researchers identified the generative nature of LLMs as an aid in property generation. While works commonly generate properties and assertions in SVA or Z3, it seems plausible to generate these in PSL or a formal language such as LTL or CTL. An emerging question from this will be: *Given a DUV, will the generated properties and assertions be consistent across the available specification languages?* As properties can be effortlessly generated with LLMs, another open question would be: *How many properties/assertions can LLMs generate from a specific set of specifications?* This is corresponding to the works regarding the completeness of verification properties [73]–[75].

IV. PRELIMINARY STUDY ON CONSISTENCY

In this section, we present the results of the experiments in which we assess the consistency of the LLMs. We performed these experiments using the Langchain framework [13]. The cosine similarity was used as a metric to compare the consistency of the functional properties generated by LLMs in natural language. For this purpose, the properties were converted into embedding using the hugging face all-MiniLM-L6-v2 sentence transformer model [76]. The experiments were carried out on 5 different modules from the ELAU library [17], [77] i.e. AddMulSgn, DivArrSgn, SqrSgn, SubVZ, and IncGrayC.

The results of the experiments are summarized in the Table I. The first column of the table shows the languages that are compared where SV stands for System Verilog, V stands for Verilog and VHDL stands for Very High Speed Integrated Circuit Hardware Description Language. In the second column, different parameters we use to assess the consistency

are listed like the minimum cosine similarity score obtained by comparing two properties (listed as *min* in the table). The maximum cosine similarity is achieved when comparing all properties of two different languages listed under *max* in the table. The average similarity score is listed under *avg* in the table. The relative number of properties for which the cosine similarity score is higher than 0.75 and 0.80 are also shown as percentages in the table, respectively. These percentages provide insight into how consistent are the properties generated by LLM when the same circuit is implemented in different languages. In the ideal case, the properties generated by LLMs should be the same regardless of the language format i.e. the percentage of properties with a cosine similarity score greater than 0.75 should be as high as possible. Some difference in the scores is expected as the properties are written in natural language and can be written and formulated in various ways.

Overall it can be seen that the consistency of the properties generated by LLMs in natural language varies across different languages. The properties generated for the same circuits in different languages hardly ever completely match. Despite the inconsistency, the results seem language agnostic i.e. the average similarity score of the properties is very similar when one language is compared to another and vice versa. Usually, the number of properties created for one language is different from the other so a little difference in the average is predictable. The percentage of properties that match strongly remain the same in almost all cases except in the SubVZ. Such behaviors can happen when LLM merges two properties into one. The inconsistency of LLMs in generating properties in natural language should not be presumed as a weakness but a potential area of research that can be explored. In the next section we highlight more areas for future research.

V. Future Directions

Several promising directions can be explored to enhance the application of LLMs in hardware verification. Incorporating RAG can improve response quality by providing relevant design information alongside LLM outputs, helping to reduce ambiguity and tailor responses to specific hardware designs. As a next step, Fine-tuning LLMs on domain-specific data can improve performance across different HDLs out-of-the-box, ensuring high-quality property generation regardless of the HDL in use. Ensuring completeness in generated properties using LLMs is crucial and currently an open research problem. In this regard, exploring formal methods in combination with state-of-the-art metrics is an interesting direction.

To enhance the reasoning capabilities of LLMs, integrating reasoning engines or formal verification tools with LLMs could ensure that generated properties are logically sound. Furthermore, LLMs could also be useful in defining coverage goals for coverage-directed simulations by analyzing design specifications and suggesting critical coverage points, thereby reducing manual effort. As a next step, LLMs can be utilized in coverage-guided simulations to provide automated coverage guidance, identifying coverage holes, and recommending new tests. Another potential area of improvement is using LLMs to define criteria for splitting complex designs and specifications into smaller, manageable sections for efficient analysis, ensuring no critical interactions are missed. This is important because of the context window limitations. Finally, developing embedding models tailored specifically for EDA can improve LLMs' understanding and generation of hardware-related content by capturing the nuances of design concepts, signals, and properties more effectively. These enhancements could streamline verification processes, making LLMs more reliable and efficient tools for hardware verification.

VI. Conclusion

In this paper, we highlighted the typical workflow of using LLMs in the EDA domain. In addition to this several libraries, frameworks, tools, etc., that are required at each step of the methodology are discussed. We then summarize the recent advancements in the domain of using LLMs for hardware verification. We took an example of one of the workflows used in the work which used LLMs and RAG for identifying bugs in the verification process. While the LLMs have been used for generating properties we showed that they are very inconsistent when used as is and also vary widely depending upon the language chosen for the implementation. We then propose some directions that can help to improve the use of LLMs in hardware verification and suggest potential research directions.

Acknowledgements

This work was supported in part by the German Research Foundation (DFG) within the Reinhart Koselleck Project PolyVer (DR 287/36-1) and by the German Federal Ministry of Education and Research (BMBF) within the project ECXLplus under contract no. 01IW24001.

References

[1] J. Achiam *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[2] META, "Llama models." [Online]. Available: https://github.com/meta-llama/llama

[3] D. Saha *et al.*, "Empowering hardware security with LLM: The development of a vulnerable hardware database," in *2024 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, vol. 33. IEEE, 2024, p. 233–243.

[4] R. Afsharmazayejani *et al.*, "Toward hardware security benchmarking of LLMs," in *2024 IEEE LLM Aided Design Workshop (LAD)*, vol. 55. IEEE, 2024, p. 1–7.

[5] B. Ahmad *et al.*, "On hardware security bug code fixes by prompting large language models," *IEEE Transactions on Information Forensics and Security*, vol. 19, p. 4043–4057, 2024.

[6] 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*. ACM, 2024, p. 1–7.

[7] D. Saha *et al.*, "LLM for SoC security: A paradigm shift," *IEEE Access*, p. 1–1, 2024.

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

[9] 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, p. 1–5.

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

[11] 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, p. 1–1.

[12] V. Pulavarthi *et al.*, "AssertionBench: A benchmark to evaluate large-language models for assertion generation," 2024.
[13] Langchain, "Langchain." [Online]. Available: https://www.langchain.com/
[14] Haystack, "Haystack." [Online]. Available: https://haystack.deepset.ai/
[15] 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, p. 1–6.
[16] M. Hassan *et al.*, "LLM-guided formal verification coupled with mutation testing," in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2024, p. 1–2.
[17] ETH Zurich & Università di Bologna, "Library of arithmetic units (ELAU)," https://github.com/pulp-platform/ELAU, 2024.
[18] AutoGPT, "Autogpt: Build, deploy, and run ai agents." [Online]. Available: https://github.com/Significant-Gravitas/AutoGPT
[19] Databricks, "Databricks." [Online]. Available: https://www.databricks.com/
[20] Airflow, "Airflow." [Online]. Available: https://airflow.apache.org/
[21] Unstructured, "Unstructured." [Online]. Available: https://unstructured.io/
[22] Langchain, "Embedding models." [Online]. Available: https://python.langchain.com/docs/integrations/text_embedding/
[23] G. Kamradt, "5 levels of text splitting." [Online]. Available: https://github.com/FullStackRetrieval-com/RetrievalTutorials/blob/main/tutorials/LevelsOfTextSplitting/5_Levels_Of_Text_Splitting.ipynb
[24] Cohere, "Cohere." [Online]. Available: https://github.com/cohere-ai/cohere-toolkit
[25] H. Face, "Embedding models leaderboard." [Online]. Available: https://huggingface.co/spaces/mteb/leaderboard
[26] Langchain, "Vector stores." [Online]. Available: https://python.langchain.com/docs/integrations/vectorstores/
[27] Pinecone, "Pinecone." [Online]. Available: https://github.com/pinecone-io/pinecone-python-client/blob/main/README.md
[28] Qdrant, "Qdrant." [Online]. Available: https://github.com/qdrant/qdrant
[29] Weaviate, "Weaviate." [Online]. Available: https://github.com/weaviate/weaviate
[30] Chroma, "Chroma." [Online]. Available: https://github.com/chroma-core/chroma
[31] META, "Faiss." [Online]. Available: https://github.com/facebookresearch/faiss
[32] Pgvector, "Pgvector." [Online]. Available: https://github.com/pgvector/pgvector
[33] LlamaIndex, "Llamaindex." [Online]. Available: https://github.com/run-llama/llama_index
[34] C. Wolf *et al.*, "Yosys-a free verilog synthesis suite," in *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, vol. 97, 2013.
[35] R. Brayton *et al.*, "ABC: An academic industrial-strength verification tool," in *Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings 22*. Springer, 2010, pp. 24–40.
[36] LangChain, "Large language models." [Online]. Available: https://python.langchain.com/docs/integrations/llms/
[37] G. Team *et al.*, "Gemini: a family of highly capable multimodal models," *arXiv preprint arXiv:2312.11805*, 2023.
[38] Anthropic, "Anthropic llms." [Online]. Available: https://github.com/anthropics
[39] Redis, "Redis." [Online]. Available: https://github.com/redis/redis
[40] SQLite, "Sqlite." [Online]. Available: https://www.sqlite.org/
[41] GPTCache, "Gptcache." [Online]. Available: https://github.com/zilliztech/GPTCache
[42] MLFlow, "Mlflow: A machine learning lifecycle platform." [Online]. Available: https://github.com/mlflow/mlflow
[43] Helicone, "Helicone - open source llm-observability platform for developers." [Online]. Available: https://github.com/Helicone/helicone
[44] PromptLayer, "Promptlayer - maintain a log of your prompts and openai api requests." [Online]. Available: https://github.com/MagnivOrg/prompt-layer-library
[45] Rebuff, "Rebuff - self-hardening prompt injection detector." [Online]. Available: https://github.com/protectai/rebuff
[46] LMQL, "Lmql - a programming language for large language models." [Online]. Available: https://github.com/eth-sri/lmql
[47] Guidance, "Guidance - a guidance language for controlling large language models." [Online]. Available: https://github.com/guidance-ai/guidance
[48] Z. He *et al.*, "Large language models for EDA: Future or mirage?" in *Proceedings of the 2024 International Symposium on Physical Design*, vol. 35. ACM, 2024, p. 65–66.
[49] N. Wu *et al.*, "Survey of machine learning for software-assisted hardware design verification: Past, present, and prospect," *ACM Transactions on Design Automation of Electronic Systems*, vol. 29, no. 4, p. 1–42, 2024.
[50] U. Sharma *et al.*, "Openroad-assistant: An open-source large language model for physical design tasks," in *Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD*, 2024, pp. 1–7.
[51] B.-Y. Wu *et al.*, "Eda corpus: A large language model dataset for enhanced interaction with openroad," in *2024 IEEE LLM Aided Design Workshop (LAD)*, vol. 36. IEEE, Jun. 2024, p. 1–5.
[52] M. Liu *et al.*, "Chipnemo: Domain-adapted llms for chip design," *arXiv preprint arXiv:2311.00176*, 2023.
[53] W. Salcedo *et al.*, "Leveraging generative AI for rapid design and verification of a vector processor SoC," *IEEE Design & Test*, p. 1–1, 2024.
[54] J. Blocklove *et al.*, "Chip-Chat: Challenges and opportunities in conversational hardware design," in *2023 ACM/IEEE 5th Workshop on Machine Learning for CAD (MLCAD)*. IEEE, 2023.
[55] L. J. Wan *et al.*, "Invited paper: 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, p. 435–441.
[56] J. Blocklove *et al.*, "Evaluating LLMs for hardware design and test," in *2024 IEEE LLM Aided Design Workshop (LAD)*, vol. 1. IEEE, 2024, p. 1–6.
[57] X. Wang *et al.*, "ChatCPU: An agile CPU design & verification platform with LLM," in *61st ACM/IEEE Design Automation Conference (DAC'24)*, 2024, p. 6.
[58] S. Thakur *et al.*, "Verigen: A large language model for verilog code generation," *ACM Transactions on Design Automation of Electronic Systems*, vol. 29, no. 3, pp. 1–31, 2024.
[59] S. Liu *et al.*, "RTLCoder: Outperforming GPT-3.5 in design RTL generation with our open-source dataset and lightweight solution," in *2024 IEEE LLM Aided Design Workshop (LAD)*. IEEE, Jun. 2024, p. 1–5.
[60] H. Huang *et al.*, "Towards LLM-powered verilog RTL assistant: Self-verification and self-correction," 2024.
[61] R. Qiu *et al.*, "AutoBench: Automatic testbench generation and evaluation using LLMs for HDL design," in *Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD*. ACM, 2024, p. 1–10.
[62] Z. Zhang *et al.*, "LLM4DV: Using large language models for hardware test stimuli generation," in *Machine Learning for Systems 2023*, 2023.
[63] R. Ma *et al.*, "VerilogReader: LLM-aided hardware test generation," in *2024 IEEE LLM Aided Design Workshop (LAD)*, vol. 34. IEEE, 2024, p. 1–5.
[64] M. Rostami *et al.*, "Beyond random inputs: A novel ML-based hardware fuzzing," in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2024, p. 1–6.
[65] 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)*, vol. abs/2312.11805. IEEE, 2024, p. 1–4.
[66] 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.
[67] S. Qiu *et al.*, "LLM-aided explanations of EDA synthesis errors," in *2024 IEEE LLM Aided Design Workshop (LAD)*, vol. 20. IEEE, 2024, p. 1–6.
[68] S. Fernando *et al.*, "Boosting productivity of hardware documentation using large language models," in *2024 IEEE LLM Aided Design Workshop (LAD)*. IEEE, 2024, p. 1–1.
[69] S. Yang *et al.*, "FormalEval: A method for automatic evaluation of code generation via large language models," in *2024 2nd International Symposium of Electronics Design Automation (ISEDA)*. IEEE, 2024, p. 660–665.
[70] M. Hansen *et al.*, "Unveiling the ISCAS-85 benchmarks: a case study in reverse engineering," *IEEE Design & Test of Computers*, vol. 16, no. 3, p. 72–80, 1999.
[71] T. B. Brown *et al.*, "Language models are few-shot learners," in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, ser. NIPS '20. Red Hook, NY, USA: Curran Associates Inc., 2020.
[72] J. Wei *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.
[73] S. Katz *et al.*, *"Have I Written Enough Properties?" - A Method of Comparison Between Specification and Implementation*. Springer Berlin Heidelberg, 1999, p. 280–297.
[74] H. Chockler *et al.*, "Coverage metrics for formal verification," in *Correct Hardware Design and Verification Methods: 12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003, L'Aquila, Italy, October 21-24, 2003. Proceedings 12*. Springer, 2003, pp. 111–125.
[75] R. Drechsler *et al.*, "Completeness-driven development," in *International Conference on Graph Transformation*. Springer, 2012, pp. 38–50.
[76] all-MiniLM-L6-v2, "all-MiniLM-L6-v2," 2023. [Online]. Available: https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2
[77] R. Zimmermann, "VHDL library of arithmetic units," in *Proc. First Int. Forum on Design Languages (FDL'98), Lausanne, Switzerland*. Citeseer, 1998, pp. 267–272.