Large Language Model for Verification: A Review and Its Application in Data Augmentation

Dan Yu, Siemens Industry Software Inc., dan.yu@siemens.com
Eman El Mandouh, Siemens Egypt, eman.mandouh@siemens.com
Waseem Raslan, Siemens Egypt, waseem.raslan@siemens.com
Harry Foster, Siemens Industry Software Inc, harry.foster@siemens.com
Tom Fitzpatrick, Siemens Industry Software Inc, tom.fitzpatrick@siemens.com

***Abstract*- This paper presents a comprehensive literature review for applying Large Language Models (LLMs) in multiple aspects of verification, including requirement engineering, coverage closure, formal verification, debugging, functional safety, code generation and completion, and data augmentation. To demonstrate the capability, experiments are carried out to automatically generate variants of the existing designs and their verification code using prompts. Significant productivity and quality improvements are recorded compared to traditional manual data preparation. Despite the promising advancements offered by this new technology, it is essential to be aware of the inherent limitations of LLMs that lead to incorrect predictions. The paper cautions against using raw outputs of LLMs directly in verification. In conclusion, three safeguarding mechanisms are recommended to ensure the quality of LLM outputs. Finally, the paper summarizes the observed trend of LLM development and expresses optimism about their broader prospective applications in verification.**

I. INTRODUCTION

Language models are arguably the most essential types of machine learning models used for functional verification. This process involves handling numerous forms of textual data, including specifications, source code, test plans, test benches, logs, and reports. Most of the textual content comprises natural languages, controlled natural languages, or programming languages. Therefore, effective use of language models is critical for the application of AI/ML in functional verification. The development of pretrained LLMs [1] represented by commercially available GPT 3.5/4 models and open-source Large Language Model Meta AI (LLaMA) series models has paved the way for the application of LLMs in EDA applications and specifically, functional verification. Some innovative researchers have included LLMs in their research and published exciting results, which are summarized in this paper. The paper proposes a systematic approach to finetuning a high-quality LLM and showcases its use in data augmentation. Our survey and experiments also enable us to identify three paradigms for effective LLM integration in functional verification.

II. APPLICATIONS OF LLMS IN EDA, SOFTWARE ENGINEERING AND VERIFICATION

Owing to the novelty of LLMs, research results for the application of LLMs in verification have primarily been published within the last 2 years. While a few examples have focused on broader areas such as software engineering or EDA, their findings are generally applicable to HDL as well.

A. Assertion generation

Generating hardware design assertions, normally in SystemVerilog, directly from natural language properties has been extensively researched. Some previous research mainly focuses on using Constrained Natural Language (CNL), a subset of natural language with restrictive grammar and vocabulary to describe properties. Various rule-based or machine learning model-based converters are employed to convert CNL to predictable assertions. Learning a new language negates many benefits of such systems. Due to a limited number of datasets available for training, these systems perform less than ideally.

nl2sva [2] employs LLMs to translate specifications into intermediate representations based on the context of the design being verified. To ensure the quality of the results, a human-in-the-loop approach is introduced. However, no benchmark results are provided. On the other hand, the assertion generator described in [3] employs a Syntax Fixer to

enhance the quality of the generated assertions. Examples are provided in each prompt to guide the LLMs in generating assertions. These assertions are then mass generated, compiled, and checked against the design using a simulator to evaluate their quality. The study reports that only 9.29% of the generated assertions are correct. However, the best combinations of various detailed prompt components contribute 80% of all correct assertions, which demonstrated the importance of prompt context.

B. Coverage closure

In simulation-based verification, tracking and calculating code coverage can be computationally intensive and time-consuming. [4] introduces a curated dataset specifically designed for coverage prediction for Python code. Each line of code is labeled to indicate its coverage status (executed, missed, or unreachable) based on a given test. LLMs are then tasked with predicting the code coverage using zero, one, or multiple examples. While the accuracy of line-level predictions remains relatively low, ranging from 20-30%, GPT-4 shows promising results at the statement level. It achieves accuracies of 84.47%, 90.71%, and 90.5% for zero-shot, one-shot, and multi-shot prompts, respectively. While the results are derived from Python code, the general methodology is applicable to any programming language, including HDL languages. However, it is expected that the prediction accuracy may be worse due to the relatively smaller representation of these languages used in training the LLMs.

C. Formal verification

Formal verification uses mathematical models and formal methods to systematically validate the functionality and reliability of the design against specific requirements. Although rigorous mathematics is not the strength of most LLMs, Baldur [5] introduces a repair mechanism to generate formal proofs with LLMs. The proof repair mechanism incorporates error messages from failed proofs to improve the next prompt to LLMs. The experiment trained a small LLM with only 700m parameters on natural language text and code and fine-tuned on proofs. It demonstrates that it is possible to generate enough assertions to check the performance of the technique. When combined with previous automatic proving tools, it can prove 65.7% of theorems in the extensive Isabelle/HOL benchmark. This achievement represents a new state-of-the-art result.

D. Debugging

AutoFL [6] automates fault localization by providing prior failed test reports to an LLM. Its performance on the Defects4J benchmark is comparable to that of classic state-of-the-art techniques. Notably, it has successfully identified 40% more new bugs compared to the best classic approaches at acc@1. Additionally, the Libro framework [7] uses LLMs to generate tests using a few-shot approach based on bug reports. LLMs are presented with example bug reports and tests as part of the prompt and tasked with generating a few test examples specific to the bug report in question. These generated test examples are then ranked based on their likelihood of uncovering the reported bug before the actual test runs. The tests generated by Libro successfully reproduce 33.5% of all bugs in the benchmark dataset.

E. Test stimulus generation

As a classic methodology, Constrained-Random Verification (CRV) is an important tool for hardware verification. However, generating stimuli to hit the desired verification target requires years of experience and deep understanding of the Design Under Test (DUT). LLM4DV [8] introduces a benchmark frame for the experiments to elicit test stimuli with LLM. It reports various levels of success depending on the complexity of the DUT. Code coverage of 98.94% is achieved for a primitive data prefetcher core. However, with increasing complexity, the result drops to 86.19% for an Ibex CPU Instruction Decoder. With a coverage rate of merely 5.61%, it does not demonstrate any meaningful stimulus generation capability for a practical Ibex CPU design, Although better than the baseline CRV, its usefulness is restricted in smaller-scale designs or module/unit-level verification.

F. Functional safety and security

DIVAS [9] uses the knowledge embedded in LLMs to automate the entire workflow, starting from vulnerability identification to mitigation with a security policy. It begins by identifying security vulnerabilities based on user-defined SoC specifications, which are then converted to SystemVerilog Assertions (SVAs) after mapping them to Common Weakness Entities (CWEs). A simulator is employed to evaluate the quality of the generated SVAs. The SVAs that are successfully simulated are subsequently transformed into a security policy to address the identified vulnerabilities. The paper reports a success rate of 40%-55% in generating relevant CWEs using the GPT-4 model. However, the success rate of the final policy fixes is not specifically mentioned in the paper.

G. *Code generation and completion*

The research presented in [10] introduces an experimental system that uses finetuned LLMs to generate Verilog code completions. To train these LLMs, the researchers collected a comprehensive training corpus consisting of Verilog code sourced from open-source repositories and selected textbooks. The finetuned LLMs, which vary in size, demonstrate impressive capabilities in generating compilable code for different levels of complexity. Specifically, they achieve success rates of 98.7%, 72.8%, and 59.9% for basic, intermediate, and advanced problems, respectively. In comparison, the performance of open-source pretrained models is much lower, reaching only 24% in the best-case scenario. The commercial code-davinci-002 model, on the other hand, achieves success rates of 84.7%, 45.2%, and 56.9% for the same tests. Overall, the finetuned LLMs are able to produce functionally correct code 41.9% of the time, which is significantly better than the performance of code-davinci-002 at 35.4%.

III. DESIGN AND VERIFICATION EXPERIMENTS WITH OPEN-SOURCE LLMS

Due to concerns about the potential risk of IP leakage, many users of EDA software have chosen to host permissive open-source LLMs instead of using publicly available LLMs. In this paper, we focus on evaluating the performance with open-source LLM Code LlaMa 34B [10] in hardware design and verification tasks.

A. *LLM with Mutation-Based Testing*

Mutation testing is a technique to identify significant weakness and holes otherwise unnoticed in functional verification testbenches. A "mutation" is an artificial modification in the tested design induced by a fault operator. It changes the behavior of the tested design. When a test set detects all the induced mutations, or “kills the mutants,” the test set is said to be *mutation-adequate*. The proposed LLM-based mutation testing framework is illustrated in Figure 1. After extracting relevant design signals, variables, and ports, the RTL source code is scanned to extract specific context candidates for fault injection. The LLM is then directed by a set of prompts to change logical, arithmetic, bitwise and equality operators to generate a set of faulty versions of the original DUT. A compilation step is followed to assure that the injected changes are syntactically correct. The “Mutation Killing Ratio” is derived by comparing the simulation dump data of the original DUT and revised DUTs. Its value can be used to determine how many of the injected faults have been detected. Undetected cases are provided as feedback to help direct a fix for the coverage hole in the testbenches.

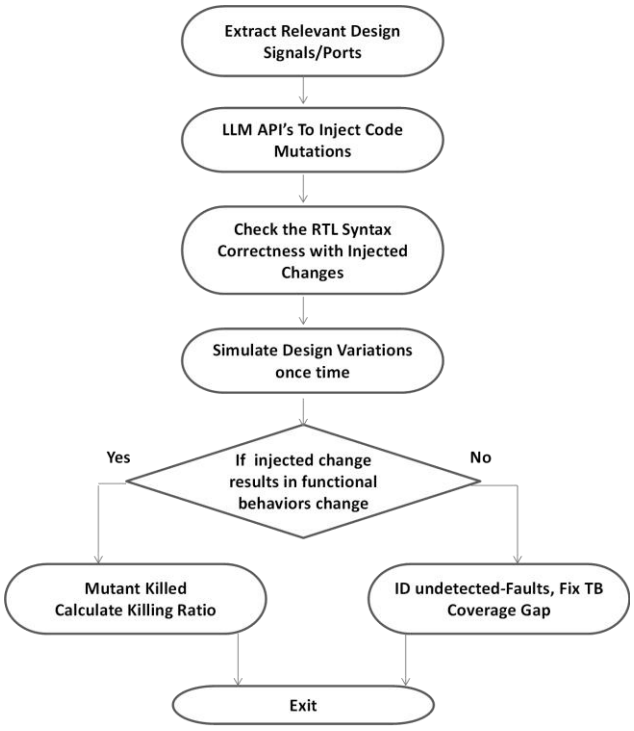

Figure 1 LLM Mutation-Based-Testing Framework

Table1summarizes the experiment results on 3 in-house designs. 12 variations have been created for each design with 63, 69, and 105 injected change ranges, respectively. Among all LLM-generated changes, although 75% compile

without errors, 25% still requires manual fixes. The testbench can detect an average of 50.59% of injected changes only and 49.41% were undetected by the current testbench. This exercise highlights test coverage gaps that should be fixed in the design testbenches. These results prove the power of LLM in a mutation-injection methodology to automate testbench hole identification.

	Design_1	**Design_2**	**Design_3**	**Average**
No of LLM Generated Design Versions	12	12	12	12
No of LLM RTL Changes Injected	69	63	105	79
Mutation Killing Ratio (Detected Injected Changes)	38.66%	66.67%	46.43%	50.59%
Undetcted Changes Ratios (Bug in Testbench)	61.34%	33.33%	53.57%	49.41%

Table 1. LLM Change Injection Experimental Runs

B. Comparison of Design Creation Strategies

The other experiments are conducted to evaluate the LLM's capabilities to accelerate RTL design creation. Advanced prompt engineering patterns described in [12] are used to generate high quality prompts for the experiment. A locally hosted Code Llama model is used to ensure consistency and controllability.

One may also need to instruct the LLM to generate code that follows certain coding guidelines that do not violate subsequent Lint checks. Both bottom-up and top-down methods are evaluated in creating a simple ALU/CPU design from prompts.

Create a multadd Verilog module that is a two-stage pipeline. It accepts four 8-bit numbers (a, b, c, d) multiplies "a" and "b" and multiplies "c" and "d" in the first cycle. It returns the sum of the two multiplications in one 17-bit on output "prodsum" port on the second clock cycle.
Good start, add a new input signal, pipeline_ready, that indicates when the data is ready on the input ports (a-d)
OK, good. Add another output port that indicates when the summation data on prodsum is ready.
Great, now I need to create a simple ALU that utilizes this multadd_pipeline module you have created.
Add a new operation to this ALU to give the summation of the four input data ports [a-d].
Not there yet, we need to add a new port, operation, to the ALU that identifies when the ALU should do the multadd operation and when it should do the summation operation.
OK, refactor the summation operation in a new Verilog module and use that module in the ALU with the multadd_pipeline.

Table 2. LLM Change Injection Experimental Runs

```
module multadd_pipeline(
    input wire clk, // Clock input
    input wire reset, // Reset input
    input wire pipeline ready, // Data ready signal
    input wire [7:0] a, // Input a (8 bits)
    input wire [7:0] b, // Input b (8 bits)
    input wire [7:0] c, // Input c (8 bits)
    input wire [7:0] d, // Input d (8 bits)
    output reg [16:0] prodsum, // Cutput prodsum (17 bits)
    output reg result_ready // Oatput result_ready signal
);
    reg [16:0] multl_result, mult2_result; // Internal registers to store multiplication
results
    always @(posedge clk or posedge reset) begin
        if (reset) begin
            multl_ result <= 0;
            mult2_result <= 0;
        end
        else begin
            if (pipeline_ready) begin
                multl_ result <= a* b; // Multiply a and b
                mult2_result <= c * d; // Multiply c andd
            end
        end
    end
    always @(posedge clk or posedge reset) begin
        if (reset) begin
            prodsum <= 0;
            result_ready <= 0;
        end
        else begin
            if (pipeline_ready) begin
                prodsum <= multl_result + mult2_result: // Sum of multiplication results
                result_ready <= 1; // Set result_ready signal high when result is valid
            end
            else begin
                result_ready <= 0; // Reset result_ready signal when data is not valid
            end
```

```
        end
    end
endmodule
```

Figure 2 multadd_pipeline module generated with bottom-up creation

The Bottom-up design creation starts by implementing small building blocks and assembles them into larger blocks until the top-level design goal is achieved. Starting from a primitive multadd module, more prompts are given to the LLM to refactor the code to add more functionality. Some prompts are recorded in Table 2. The generated Verilog was almost correct except that the output port was defined as a wire instead of a reg. The LLM was able to correct it after one additional prompt reporting the issue. Notably, the generated Verilog code compiled without any errors or warnings, as illustrated in Figure 2.

Top-down creation starts with a top-level high-level design that instantiates several building blocks, which is improved with new building blocks. To design a new CPU module, the “Flipped Interaction Pattern" [14] is used in prompting. It’s left to the LLM to discover the specs with questions. The creation starts with the first prompt: “From now on, I would like you to ask me questions to create a new CPU using Verilog. When you have enough information or when I say "generate using the information you have", create a Verilog module to model this new CPU. Ask me the questions one by one.” The prompts generated by the LLM are recorded in Table 3.

What is the word size of this CPU (e.g., 8-bit, 16-bit, 32-bit)?
What are the main components of the CPU (e.g., ALU, register file, control unit, etc.)?
Are there any specific instructions that this CPU will support?
Do you have any specific clock frequency or timing requirements in mind?
Are there any specific addressing modes that you would like to support in this CPU?

Table 3 Prompts generated in flipped interaction for a CPU design

Answering those questions resulted in creating various basic Verilog modules for the selected CPU components: ALU (Arithmetic Logic Unit), Register File, Control Unit, Instruction Memory, and Data Memory. Finally, the LLM generated a Verilog module that instantiates and connects those modules. A code snippet appears in the following Figure 3. The LLM is further instructed to make those different components more sophisticated using refactoring techniques outlined in top-down creation. The generated top-level module compilis without errors or warnings but gives synthesis Lint violations since the different instantiated components are not added to the design. Those violations disappear once all smaller modules are added to the design.

```
module simple _lébit_cpu (
    input clk,
    input rst,
    input [15:0] instruction,
    input [15:0] data_input,
    output reg [15:0] data_output
);

    // Internal signals
    reg [15:0] pe;
    reg [15:0] ir;
    reg [15:0] reg file read_data_a, reg file read_data_b;
    reg [15:0] alu_result;
    reg [15:0] data_mem_read_data, data_mem_write_ data;
    reg [15:0] next_pc;

    // Instantiate components
    alu_16bit alu_inst (
        .operand_a(reg_file_read_data_a),
        .operand_b(reg_file_read_data_b),
        .alu_op(ir[3:0]),
        .result (alu_result)
    );

    register file 16bit reg_file_inst (
        .read_addr_a(ir[7:4]),
        .read_addr_b(ir[11:8)]),
...
```

Figure 3. A simple 16-bit CPU generated with top down creation

C. Design Understanding and Visualization

Another LLM application experiment is design understanding. The LLM’s capability in describing Verilog code in a text format is tested. The control_operation example code generated from the previous test is recognized as an FSM

by the LLM, which generates the Graphviz dot language representation of an FSM diagram with the prompt "I need to visualize this control_operation module using graphviz as FSM diagram". The generated FSM visualization is almost correct from the first attempt as in Figure 4, except for the logic of the loop-back transitions and the logic operators. One additional prompt to the LLM successfully fixed the errors. It should be noted that without specifying the type of diagram, the LLM defaulted to generating a block diagram representation.

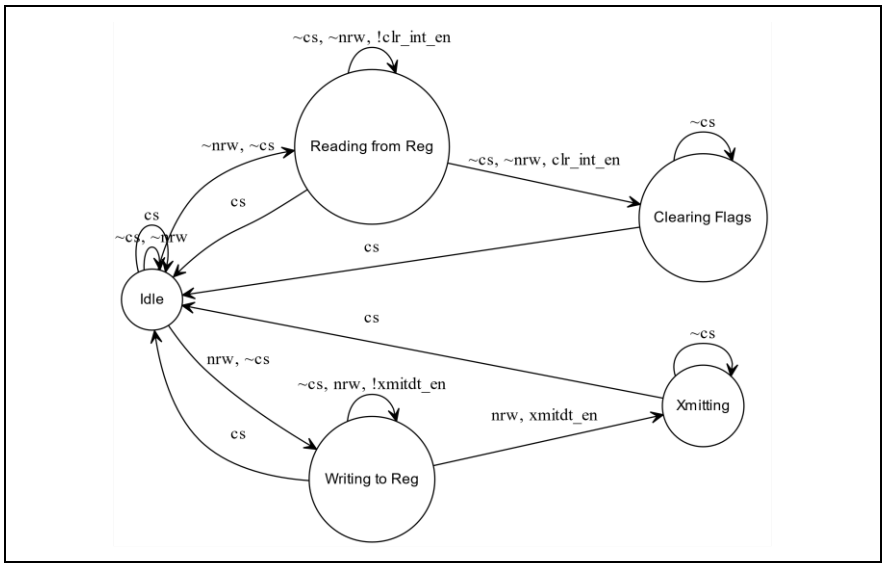

Figure 4. Graphviz visualization of LLM generated FSM visualization of control_operation module

IV. PARADIGMS OF LLM APPLICATION FOR VERIFICATION

The Universal Approximation Theorem has demonstrated that Artificial Neural Networks (ANNs) can approximate the behavior of any continuous function within a specific range with arbitrary accuracy. However, the theorem also implies that prediction errors are to be expected in most cases. The transformer architecture, which serves as the fundamental architecture for almost all LLMs, is not exempt from these prediction errors [13].

Hence, it is advised that the responses obtained from LLMs should not be directly applied to various verification tasks. Figure 5 demonstrates 4 application paradigms that address the inherent issue of prediction errors. In paradigm (a), additional knowledge that is not accessible to the LLM is used to create a quality gate or guardrail, filtering out unqualified results. This knowledge can be in the form of domain expertise or specific rules related to the domain. Paradigm (b) introduces a quality gate that can produce feedback, such as error messages, to the LLM until it generates qualified results. This continuous feedback loop allows for refinement and improvement over time. In paradigm (c), an external agent capable of performing high-precision tasks is introduced to collaborate with the LLM. The agent assists the LLM, enabling it to focus on task understanding and orchestration while delegating accurate execution tasks to the agent.

As evidenced by most research in Section II, LLMs still struggle with complex design and verification problems. Paradigm (d) combines a programmatic approach together with an LLM to tackle more challenging complex problems. It leverages a set of programmatic rules to address the complex problems presented to the LLM.

Our experiments in Section III have extensively adopted the chain-of-though paradigm illustrated in (d) in either top-down or bottom-up creation experiments to achieve satisfactory performance. CodeChain [15] proposes a novel chain-of-thought approach to break the problem down into smaller problems and generate corresponding submodules with self-revisions iteratively. The submodules are then clustered together to generate more generic and reusable implementations for each cluster. The last step involves reimplementing the originally-generated submodules with these generic implementations. It reports significant improvements in code modularity and correctness. For example, it posts 61.5% overall correctness, much higher than GPT4's 34.75%, or GPT4+human feedback self-repair's 52.60%.

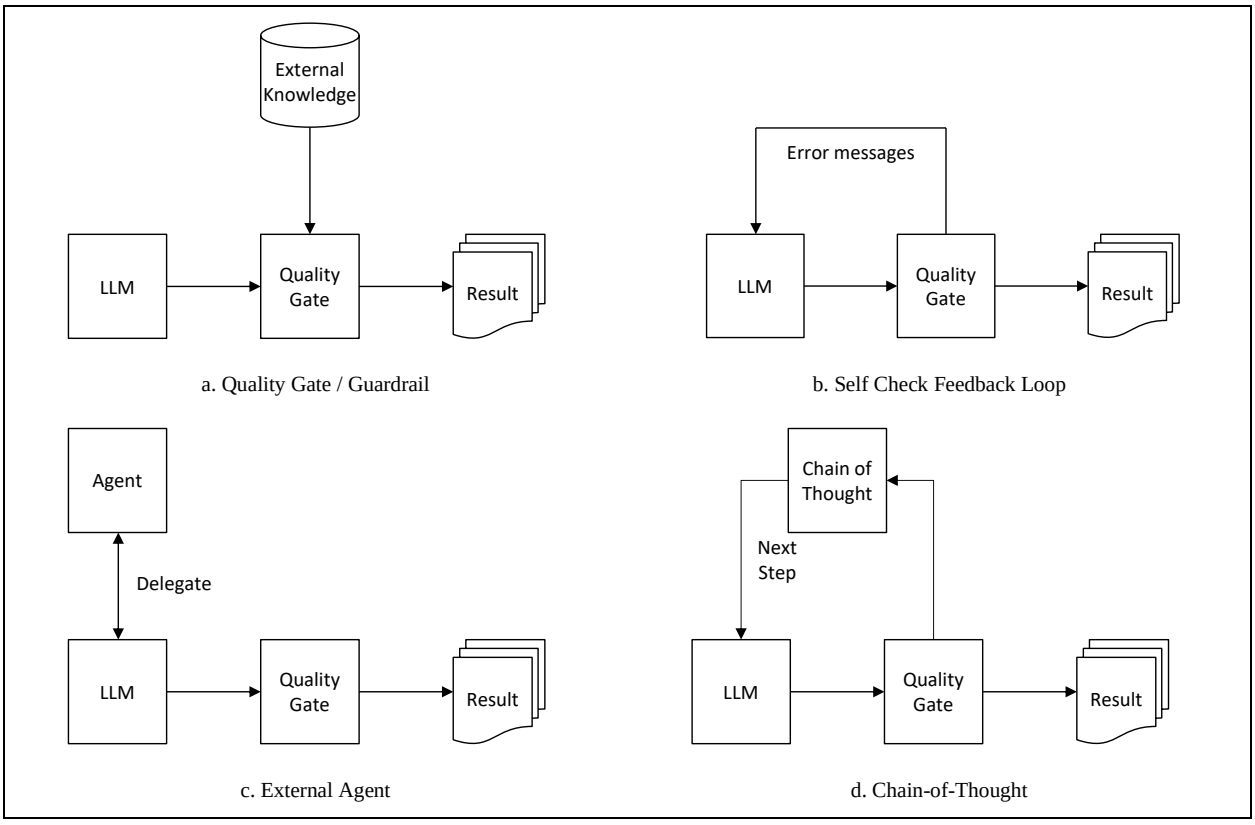

Figure 5. LLM Application Paradigms

V. CONCLUSION AND LLM TRENDS

Although recent progress of LLMs for hardware design and verification (DV) has demonstrated promising progress, the reported performance and usability still lags behind the best achievable with classic algorithms and methodologies. As most hardware design languages (HDLs) are less represented in almost all open datasets, most LLMs trained on them cannot achieve the best performance for hardware design and verification tasks.

During our experiments, we also encountered a few quirks in the LLM's behavior, including its struggle in following instruction to capitalize keywords, to generate synchronous reset, and inconsistency in the outputs. This behavior is expected as HDLs only represent a very small subset of programming languages used in training most LLMs. Our analysis in May 2022 indicates that among the available GitHub repositories, where most LLMs acquire their coding training data, only 39,5k projects are written in major HDLs (Verilog, SystemVerilog and VHDL). However 6,8m, 5.6m and 3.4m projects are written in more popular programming languages JavaScript, Java and Python, respectively. Underrepresentation of HDLs in the training data also makes LLMs more susceptible to hallucinations and bad instruction-following capabilities observed in the experiments.

Much research has stressed the importance of data. For example, VerilogEval [16] introduces an open dataset that extracts 156 problems from HDLBits with automatic tests to validate the functional correctness of LLM-generated answers. As reported by much of the research on domain-specific LLMs, a smaller LLM trained on a specific domain's datasets may outperform much larger generic models in the domain knowledge QA [17].

The availability of more open-source LLMs in the ongoing technology race against commercial LLMs has opened possibilities for finetuning high-quality LLMs using proprietary datasets. As a result, the design verification community has begun to leverage LLMs to solve verification problems. This paper provides an overview of the current applications of LLMs in verification. These versatile LLMs, trained on general knowledge, demonstrate flexibility and emerging capability in solving complex verification problems within the studied applications. They complement the classic verification solutions and, in some cases, even outperform them. Our experiments have also shown a significant increase in productivity for verification engineers during the design creation phase. The survey and experiments enable us to summarize typical application paradigms. Rather than directly relying on the responses

generated by LLMs, various application paradigms should be employed to ensure the strict quality demands of the design and verification tasks in EDA.

VI. REFERENCES

[1] Brown, Tom, et al. "Language models are few-shot learners." *Advances in neural information processing systems 33* (2020): 1877-1901.

[2] Sun, Chuyue, Christopher Hahn, and Caroline Trippel. "Towards Improving Verification Productivity with Circuit-Aware Translation of Natural Language to SystemVerilog Assertions." *First International Workshop on Deep Learning-aided Verification*. 2023.

[3] Kande, Rahul, et al. "LLM-assisted Generation of Hardware Assertions." *arXiv preprint arXiv:2306.14027* (2023).

[4] Tufano, Michele, et al. "Predicting Code Coverage without Execution." *arXiv preprint arXiv:2307.13383* (2023).

[5] First, Emily, et al. "Baldur: whole-proof generation and repair with large language models." *arXiv preprint arXiv:2303.04910* (2023).

[6] Kang, Sungmin, et al. "A Preliminary Evaluation of LLM-Based Fault Localization." *arXiv preprint arXiv:2308.05487* (2023).

[7] Kang, Sungmin, et al. "Large language models are few-shot testers: Exploring llm-based general bug reproduction." 2023 *IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023.

[8] Zhang, Zixi, et al. "LLM4DV: Using Large Language Models for Hardware Test Stimuli Generation." *arXiv preprint arXiv:2310.04535* (2023).

[9] Paria, Sudipta, et al. "DIVAS: An LLM-based End-to-End Framework for SoC Security Analysis and Policy-based Protection." *arXiv preprint arXiv:2308.06932* (2023).

[10] Roziere, Baptiste, et al. "Code llama: Open foundation models for code." *arXiv preprint arXiv:2308.12950* (2023).

[11] Y. Serrestou, V. Beroulle and C. Robach,"Functional Verification of RTL Designs Driven by Mutation Testing Metrics", In *Proceedings of Digital System Design Architectures, Methods and Tools(DSD)* , 2007, PP 222 - 227

[12] Thakur, Shailja, et al. "Benchmarking Large Language Models for Automated Verilog RTL Code Generation." *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2023.

[13] Chulhee Yun, Srinadh Bhojanapalli, Ankit Singh Rawat, Sashank J. Reddi, Sanjiv Kumar. "Are transformers universal approximators of sequence-to-sequence functions?." *arXiv preprint arXiv:1912.10077* (2019).

[14] White, Jules, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C. Schmidt. "A prompt pattern catalog to enhance prompt engineering with chatgpt." *arXiv preprint arXiv:2302.11382* (2023).

[15] Sub, Resentative. "CodeChain: Towards Modular Code Genera-Tion Through Chain Of Self-Revisions With Rep." *Arxiv Preprint Arxiv:2310.08992* (2023).

[16] Liu, Mingjie, et al. "VerilogEval: Evaluating Large Language Models for Verilog Code Generation." *arXiv preprint arXiv:2309.07544* (2023).

[17] Wu, Shijie, et al. "Bloomberggpt: A large language model for finance." *arXiv preprint arXiv:2303.17564* (2023).