FACULTY OF ENGINEERING
AND ARCHITECTURE

DEPARTMENT OF ELECTRONICS AND INFORMATION SYSTEMS

EXPLORING LARGE LANGUAGE MODELS FOR HDL VERILOG

Ewout Danneels, Karel-Brecht Decorte, Senne Loobuyck, Arne Vanheule, Ian Van Kets and Erik H. D'Hollander

Ghent University, Ghent, Belgium

Abstract

Due to the latest advances in Deep Learning, **Large Language Models (LLMs)** have the potential to automate and simplify code writing tasks. One of the emerging applications of LLMs is **hardware design**, where natural language is used to generate, annotate, and correct code in a **Hardware Description Language (HDL),** such as **Verilog**. This work provides an **overview of the current state of using LLMs to generate Verilog code**, highlighting their capabilities, accuracy, and techniques to improve the design quality. It also reviews the existing benchmarks to evaluate the correctness and quality of generated HDL code, enabling a fair comparison of different models and strategies. The major topics are:

1. What are the potential **applications** of Large Language Models (LLMs) in hardware design?
2. **How** do LLMs contribute to automating and simplifying code writing tasks in a Hardware Description Language (HDL) like Verilog?
3. What **benchmarks** are available to evaluate the correctness and quality of generated HDL code using LLMs?

Applications of Large Language Models

1. **Code Generation**: LLMs can be utilized to generate code in Hardware Description Languages based on natural language descriptions. This can streamline the process of creating complex hardware designs by allowing engineers to describe the desired functionality in natural language, which is then translated into HDL code by the LLM.
2. **Bug Detection and Correction**: LLMs can be employed to detect and correct bugs and security flaws in hardware code. By generating prompts that detail the bug and ask for a repair, LLMs can assist in addressing security vulnerabilities in hardware designs.
3. **Benchmarking**: LLMs can be evaluated for their ability to generate HDL code using benchmarks specifically designed for hardware design. These benchmarks assess the syntax, functionality, and quality of the generated HDL code, providing a means to compare different models and strategies.

Types of Large Language Models uses

Pre-trained LLMs

A pre-trained Large Language Model has undergone an initial training phase on a diverse and extensive dataset. This initial training allows the model to capture general language patterns, syntactic structures, and semantic relationships.. Examples are the ChatGPT-x (Generative Pre-trained Transformer) models..

Fine-tuning

Fine-tuning involves **adjusting the pre-trained weights of the LLM** to improve its performance in generating Verilog code. By fine-tuning, LLMs are optimized to accurately capture the nuances and subtleties of hardware design, such as timing constraints, optimization goals, and verification methods. Fine-tuning only applies to open source LLMs.

Prompt engineering

Prompt engineering **enhances natural language descriptions provided to the LLM, by offering more guidance or details to the model**. This can help the LLM better understand and interpret the requirements for generating Verilog code, leading to more accurate and tailored outputs. Prompt engineering doesn't change the weights of the LLM, it leverages the existing knowledge to improve the desired outcome.

Approaches to adapt LLMS for Verilog

DAVE: Fine-tuning ChatGPT-2 (11/2020) [Pearce]

- Fine-tuned, based on ChatGPT-2
- Training dataset in the form TASK: <English Text> RESULT: <Verilog Code>
- Training templates for assignment, register, sequence, multi-task
- Validation on non-trained templates yields 94.8 % accuracy.
- Example:

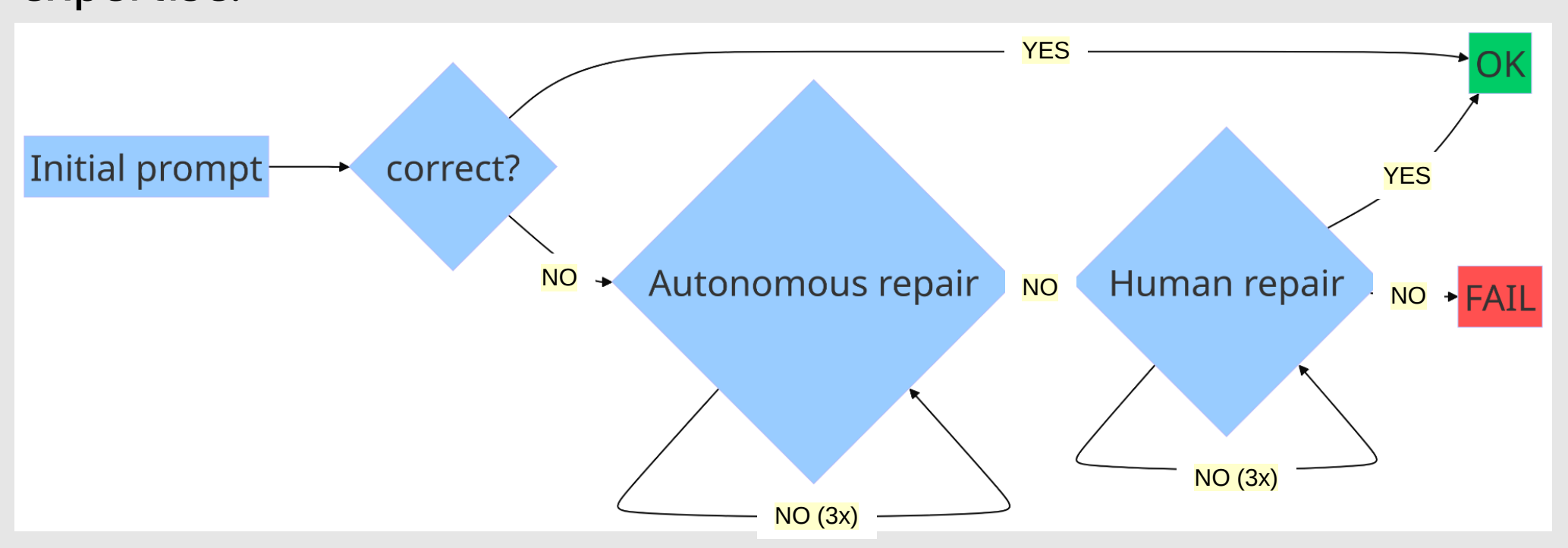
`Given 'a' and 'b', check if these are both true and return the result in 'c'.`	`assign c = a & b`

In-Context-Learning and Chain-of-Thought(7/2023) [Du]

- **ICL** presents predefined Q&A pairs as examples to the LLM.
- This enables the LLM to learn from these examples.
- **CoT** prompting gives extra details on how the examples can be dissected and tackled
- This enables the LLM with extra reasoning capabilities.
- Example: Du et al [2] used CoT to derive the FFT twiddle factors
  $W_N^k = e^{-j(2\pi k/N)}, k \in \{0,1,\dots,N/2\}$
  for N=16, 32, 64 given the code for N=8.

Reinforcement Learning with Human Feedback (5/2023) [Blocklove]

- Prompt is used to create an initial design, code and testbench.
- If there are errors, the LLM tries to fix them **autonomously**, up to 3 times. If the error persists, **simple, moderate and advanced human feedback** is given.
- The **conversational method** is used to build an 8-bit accumulator-based microprocessor, **using ChatGPT-4.**
- Sticky points: requires extra LLM time and prompt engineering expertise.



Self-planning (9/2023) [Lu]

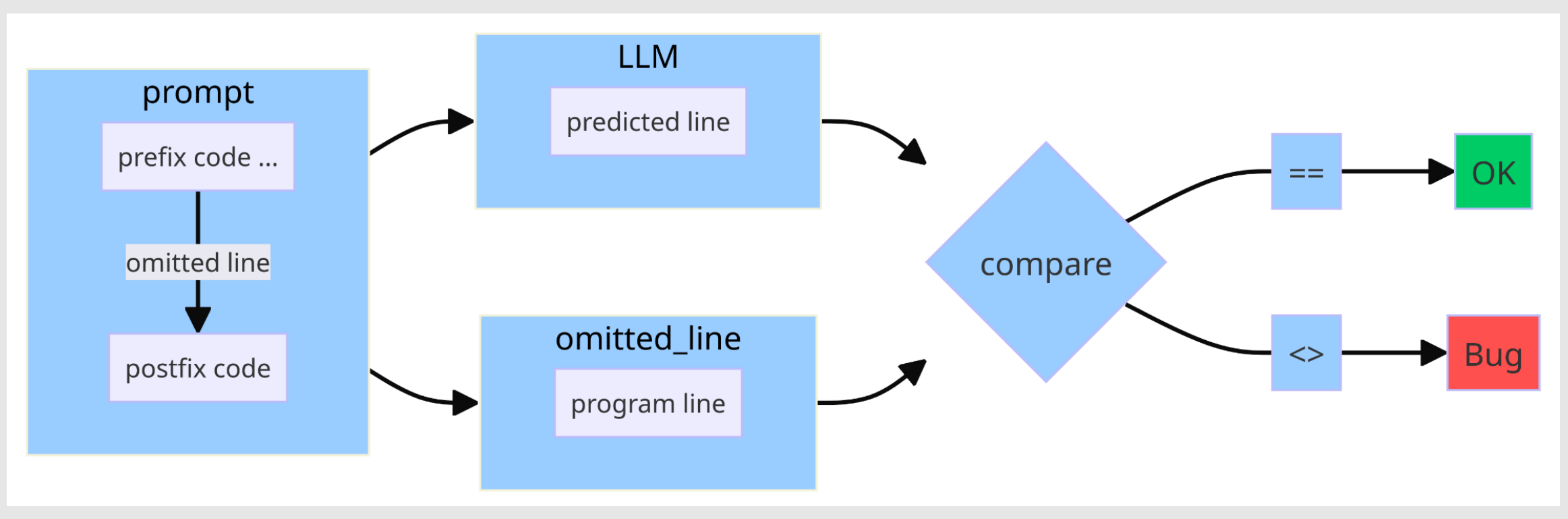
- Two-step process: planning and coding
- **Planning step**: the prompt requests a coding strategy in natural language. This is translating the prompt into **natural-language analysis and reasoning steps**, thereby preparing a better prompt.
- The planning step includes the contextual knowledge Verilog to avoid syntax errors during code generation.
- The **coding step** is based on the optimized prompt and language information.
- Self-planning increases the percentage of syntactical and functional correct codes to resp. 73% and 47% with ChatGPT-3.5
- Example of a self-planning prompt [Lu]:

```
#Implement the design of unsigned 16bit multiplier
based on shifting and adding operation.
module multi_16bit(
// ...I/O details omitted...
);
#Please try to understand the requirements above and
  give reasoning steps in natural language to achieve it.
#In addition, try to give advice to avoid syntax errors
```

Correcting Verilog code with LLMs

- **CWEs [5]**, Common Weakness Enumeration lists, contain well-known vulnerability issues. Ahmad used CWEs as part of the prompt to let the LLM repair the identified security flaws in the code.

- **FLAG [6]**, finding line anomalies using generative AI, uses the lines before and after a particular code line to regenerate this line and check anomalies. FLAG also uses comments to infer the meaning of the code. Errors are flagged based on the Levenshtein distance between the generated and the original code line.



FLAG strategy to single out each code line for anomaly detection

Evaluating the LLM performance

Pass@k metric (7/2021) [Chen]

The pass@k value is number of "at least 1 correct solution in k trials" over the total number of problems presented. Each problem is allowed k passes through the LLM to achieve a correct solution. This allows for the variability in the LLM responses.

Vgen (9/2023) [Thakur]

- Data benchmarks taken from GitHub and Verilog source books
- LLMs: Megatron-LM, CodeGen-2/6/16B, J1-Large-7B
- Compares **Fine-Tuned vs Pre-Trained LLMs.**
- Result: FT increases correct code pass@10 values from 1% to 26%.

VerilogEval (10/2023)[Liu]

- Data benchmarks: 156 problems from HDLBits
- LLMS: Codegen-2/16B, Codegen-16B-Verilog
- Uses **Supervised Fine Tuning (SFT)** by optimizing the prompt for the LLM training. The **human prompt** adapts the original problem description to a wording amenable to the LLM. The synthetic **machine prompt** is obtained by asking the LLM to describe the golden solution in natural language.
- Result: the machine prompt outperforms the human prompt, generating functional correct codes with pass@10 > 70% and pass@10 > 50% respectively.

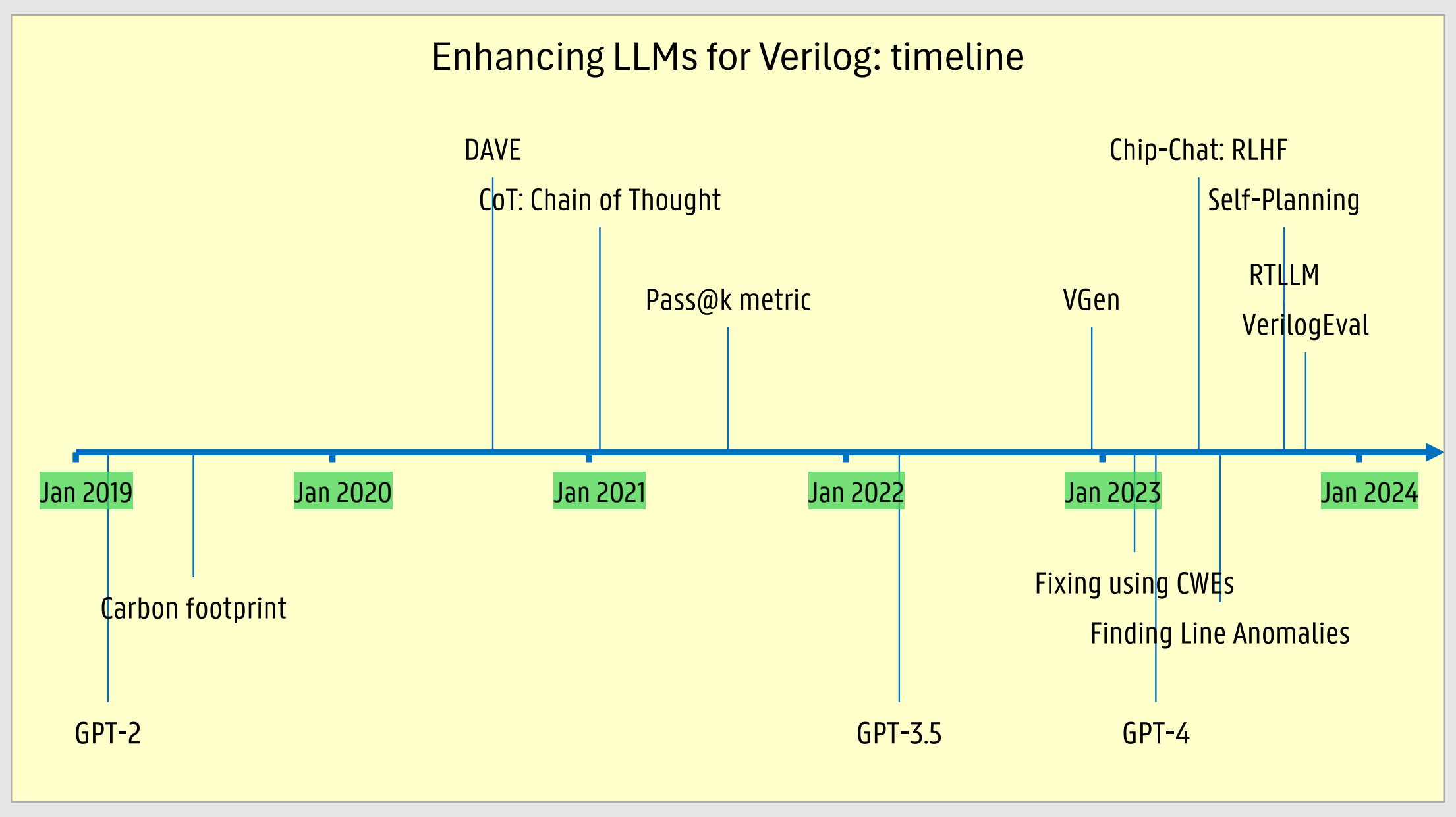
RTLLM benchmark(9/2023)[Lu]

- Compares 4 LLMs on 30 simple and complex designs.
- Improves the ChatGPT-3.5 prompt using self-planning (SP):

Correct	GPT-3.5	GPT-3.5 + SP	GPT-4
Syntax	55%	73%	81%
Functional	33%	47%	50%

Conclusion

- AI-assisted HDL design is a hot research topic, see the timeline.
- Fine-tuning and prompt engineering improve LLMs.
- Closed source large LLMs may outperform fine-tuned open LLMs.
- Training general LLMs is expensive in time and carbon footprint [10].
- Domain specific LLMs may help to close the gap with e.g. GPT-4.



References

[1] H. Pearce et al., "DAVE: Deriving Automatically Verilog from English," ACM/IEEE Workshop Machine Learning for CAD, Nov 2020, pp. 27–32.
[2] Y. Du et al., "The Power of Large Language Models for Wireless Communication System Development: A Case Study on FPGA Platforms." arXiv, Jul. 14, 2023.
[3] J. Blocklove et al., "Chip-Chat: Challenges and Opportunities in Conversational Hardware Design," in 2023 ACM/IEEE Workshop on Machine Learning for CAD, Sep. 2023, pp. 1–6.
[4] Y. Lu et al., "RTLLM: An Open-Source Benchmark for Design RTL Generation with Large Language Model." arXiv, Sep. 26, 2023.
[5] B. Ahmad et al., "Fixing Hardware Security Bugs with Large Language Models." arXiv, Feb. 02, 2023.
[6] B. Ahmad et al., "FLAG: Finding Line Anomalies (in code) with Generative AI." arXiv, Jun. 21, 2023..
[7] M. Chen et al., "Evaluating Large Language Models Trained on Code." arXiv, Jul. 14, 2021.
[8] S. Thakur et al., "Benchmarking Large Language Models for Automated Verilog RTL Code Generation," in 2023 DATE Conference, Dec. 2022, pp. 1–6.
[9] M. Liu et al., "VerilogEval: Evaluating Large Language Models for Verilog Code Generation," in Proc. IEEE/ACM ICCAD, Oct. 2023, pp. 1–8.
[10] E. Strubell et al., "Energy and Policy Considerations for Deep Learning in NLP." arXiv, Jun. 05, 2019.

GHENT
UNIVERSITY