Capabilities of Generative Artificial Intelligence for Circuit Design Automation

Andey Robins, Jenna Goodrich, Mike Borowczak

DRACO, Electrical and Computer Engineering, University of Central Florida, Orlando, Florida, USA

STATEMENT OF PROBLEM

Large language models (LLMs) have rapidly become embedded within academic and commercial spaces as they achieve greater precision and accuracy in tasks such as natural language processing and code synthesis. The specific patterns for interaction with an LLM system can have a direct effect on the quality of the system's output. This work examines multiple models for interacting with a generative artificial intelligence (GAI) system built upon LLMs across multiple dimensions. This work explores the frequency of interaction (one-shot vs few-shot) as well as the impacts of prompt patterning on the output of Verilog code synthesized from GAI systems. Implementation code is generated and the feasibility of adapting LLMs to each form of coding is assessed to frame discussion over where LLM backed GAI tools fit into the circuit design landscape and design automation pipelines, as well as what barriers must be overcome for these tools to be applicable to domains in which they currently struggle.

BACKGROUND

LLMs are systems constructed from a multitude of transformers that function as a predictor of the next token given a specific prompt. The practice, known as auto-regression, takes as input some prompt, often text, and breaks the prompt into a series of tokens. These tokens are sequences of characters that may correspond to words, short phrases, symbols, or other small pieces of information. The LLM then takes the steam of tokens and predicts the next token in the sequence [5]. A multitude of applications have been built upon this technology, and one of the questions that arise is: how effective are these tools at generating code?

General purpose LLM systems, such as Bard [3] and ChatGPT [2] as well as purpose-built applications for code completion [6] and specifically Verilog code completion [7], are all capable of generating Verilog code as output. However, "LLMs have limited abilities in comprehending complex logic and reasoning tasks, often experiencing confusion or making errors in intricate contexts" [1], raising the question of how most effectively to represent that context to generate the desired output. The templating of prompts provided to an LLM has been referred to as "prompt patterning" [8]. A unique challenge posed by interacting with LLMs using prompt patterns is that "LLMs [are] sensitive to prompts, especially adversarial prompts, which trigger new evaluations and algorithms to improve its robustness" [1]. As a result, determining the efficacy of specific prompts will empower users of LLM systems to leverage the maximum power contained within these tools.

Prior works have centered on determining the raw performance of LLMs. The problems are framed as AI problems and the accuracy and loss are the most important metrics. While the accuracy of the system is undoubtedly important, determining which prompt, or in other words which version of a problem specification, is the most effective has not received the same breadth of study. One foundational work by Thakur et al. demonstrates that performing fine-tuning on a relevant LLM can increase answer accuracy from 1% to 27%. The authors mention though that "a better prompt might yield a correct result. This indicates the importance of creating the best prompt" [7].

This work, in contrast to many other works which seek to answer which is most important between two of the three options: prompt, pattern, and model, seeks to answer what impact each of the three have on each other.

METHODS

Three distinct phases of methods were needed for this work. Each of these phases was repeated for one of three LLM applications. The models of Bard, GenV, and CodeGen were evaluated in this study.

A. Specification Design

Building on prior works, we utilize the same problem set presented by Thakur et al. [7]. Each specification is repeated either from the HDLBits website (where Thakur et al. gathered their problems from) [4] verbatim or from the specifications utilized in the previous work.

B. Prompting

Prompts were created from the prompt patterns enumerated by White et al. [8]. These prompts identified the tasks as generating Verilog code to satisfy the provided specification while providing whatever additional context the prompt pattern specified.

C. Evaluation

Each generated Verilog code file was compiled with the test benches provided with the dataset by Thakur et al. Each prompt for each pattern was assessed based on the percentage of test cases the design under test (DUT) passed. A solution which failed to compile was evaluated with an accuracy score of "0."

D. Execution

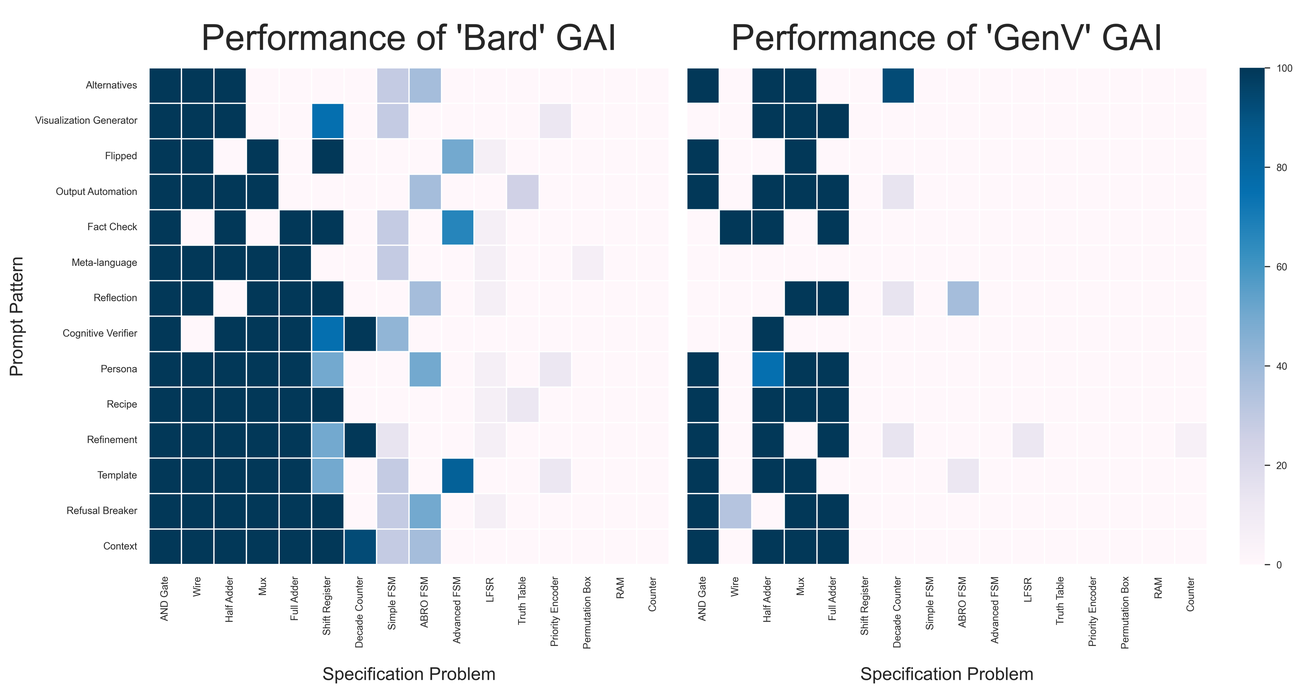
For the case of Bard, the prompting was conducted through their web portal as no formal API access currently exists [3]. For the other models, specifically "`fine-tuned-codegen-2B-Verilog`" for GenV and "`codegen2-1B`" for CodeGen, the latest version was downloaded from HuggingFace and queried using the suggested query methods.

RESULTS

Initial experiments have demonstrated dramatic differences in the results of different models at the high level. Bard produces code which is more accurate as measured by test coverage than GenV, and GenV produced code more accurate than CodeGen. As a matter of fact, the CodeGen model produced zero non-trivial programs which were syntactically correct Verilog code. The comparison of the accuracy of Bard and GenV is visualized in the figure below. The accuracy of the generated solution for each pattern and problem pair is shaded to the degree of accuracy (darker is more accurate). Problems are sorted from problems Bard produced the most passing test cases for to the least passing test cases left-to-right. Patterns are sorted similarly from top-to-bottom.

The lack of a clearly similar shape to the data from the left figure to the right begins to suggest that the patterns of prompting the GAI system which are most effective may change from system to system. In other words, the transferability of prompt pattern paradigms my be limited. Numerous factors could influence and change this effect, from the form of training data to the preprocessing and tokenization of the system, and further study on the transferability of prompt patterns will be necessary to more explicitly determine the limitations they pose.

CodeGen produced several intriguing results which, while not being Verilog code and therefore unrepresented in the data, are worthy of note. Multiple proprietary copyright statements were produced from "Wintermute Engine" to "*김동현*" and "The Intel Corporation." Furthermore, a wide array of languages were present in the output of CodeGen including Swift, C#, Java, C++, C, Python, and Ruby. Similar to the Verilog code generated, it was rarely syntactically correct.



CONCLUSIONS

Some initial generalizations about the applicability of current state-of-the-art generative AI tools can be made:

1. Task difficulty is a major predictor of accuracy, but not an exact one. This can lead to uncertainty in generated output as even "simple" solutions cannot be assumed to be done correctly by automated GAI systems.
2. GAI systems lack a more generalized training set for RTL and Verilog code [7], requiring domain expertise to fine-tune to any degree of quality. The difference in accuracy between GenV and CodeGen is a clear demonstration of this fact.
3. At present, organizations wishing to utilize GAI for solving novel problems in circuit design automation must either invest heavily into model fine-tuning and specialization or look at other levels of interaction below the RTL level.

Future work will be qualitatively analyzing the generated code to determine to what degree it could be modified by a human designer and to what degree this generated code is helpful in the design process. Furthermore, additional models will be evaluated to determine the transferability of these findings to code-generating GAI in general.

REFERENCES

[1] Chang, Yupeng, et al. "A survey on evaluation of large language models." *arXiv preprint arXiv:2307.03109* (2023).

[2] ChatGPT. "Conversations with ChatGPT." *ChatGPT,* OpenAI, Jan. 2022. Web. Accessed 18 Oct. 2023.

[3] Google Bard. "Conversations with Bard." Google AI Blog, 18 Jan. 2023. Web. Accessed 18 Oct. 2023.

[4]"HDLBits Practice Problems." *HDLBITS*, hdlbits.01xz.net/wiki/Main_Page. Accessed 1 Oct. 2023.

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

[6] Nijkamp, Erik, et al. "Codegen: An open large language model for code with multi-turn program synthesis." *arXiv preprint arXiv:2203.13474* (2022).

[7] 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.

[8] White, Jules, et al. "A prompt pattern catalog to enhance prompt engineering with chatgpt." *arXiv preprint arXiv:2302.11382* (2023).

CONTACT & ACKNOWLEDGEMENTS

Andey Robins — Andey.Robins@ucf.edu
Jenna Goodrich — Jenna.Goodrich@ucf.edu
Mike Borowczak — Mike.Borowczak@ucf.edu

This work was supported in part through the ORCGS Fellowship from the University of Central Florida.