Automated Synthesis of Hardware Designs using Symbolic Feedback and Grammar-Constrained Decoding in Large Language Models

Sumit Kumar Jha
Computer Science Department
Florida International University
Miami, FL, USA
jha@cs.fiu.edu

Susmit Jha
Computer Science Laboratory
SRI International
Menlo Park, CA, USA
jha@csl.sri.com

Muhammad Rashed ul Rashed
Computer Science and Engineering
University of Texas at Arlington
Arlington, TX, USA
muhammad.rashed@uta.edu

Rickard Ewetz
Electrical and Computer Engineering
University of Central Florida
Orlando, FL, USA
rickard.ewetz@ucf.edu

Alvaro Velasquez
Computer Science Department
University of Colorado Boulder
Boulder, CO, USA
alvaro.velasquez@colorado.edu

***Abstract*—Large language models (LLMs) are capable of creating small programs including those in hardware description languages. However, there are no guarantees on the correctness of such generated programs. Our approach seeks to create correct-by-construction hardware designs using LLMs by employing formal verification to verify the designs and by using counterexamples to guide the synthesis of such hardware designs in a counterexample-guided refinement loop. Grammar-constrained decoding is used to ensure that the generated code always satisfies the grammar of the hardware description language. We demonstrate the capability of our automated synthesis approach by generating a multiplier using LLMs and their assurance artifacts using model checking. Our approach provides a step in the direction of high-assurance synthesis of hardware artifacts using LLMs and formal methods.**

***Index Terms*—LLM, grammar-constrained decoding, automated synthesis, circuits**

I. INTRODUCTION

Neural representations learned using backward gradient propagation and optimization over large corpora of data sets have led to great leaps in the capabilities of AI models over the last decade. In particular, the rise of transformers and associated models coupled with the idea of pre-training on internet-scale data sets and instruction-tuning to allow postponing of the definition of the downstream task to inference time has led to an inflexion point where pre-trained models are becoming the dominant technology choice across areas as diverse as automated theorem proving and robotics [1]–[4].

Despite the astounding success of large language models (LLMs) on multiple tasks such as translation and code generation, the current generation of large language models have difficulty synthesizing a simple 4-bit multiplier. One solution to the problem would be to train customized LLMs on large corpora of Verilog and other hardware definition languages. However, even such future LLMs trained on large Verilog and other HDL corpora are not guaranteed to produce correct responses for all design queries.

In this paper, we propose to resolve this problem by (i) using grammar-constrained decoding [5]–[7] to enforce syntax constraints on generated hardware designs, and (ii) employing formal methods [8]–[13] to verify the correctness of the generated design against formal specifications. If the generated design is correct, we present a formal assurance artifact together with the design. On the other hand, if a counterexample is produced, we provide the counterexample as a symbolic feedback to the LLM and ask it to produce another LLM design. The process continues until a design is produced and verified, or resources such as time and energy are exhausted.

Our approach brings together the inductive generation capabilites of LLMs with deductive capabilities of formal methods, such as model checking. If a design is produced and verified, it is bound to be correct. If the LLM produces a correct design, it will indeed be verified by the model checker. However, it is possible that the LLM continues to produce incorrect designs without ever yielding the correct design.

Our approach leverages neuro symbolic reasoning allowing the LLM to benefit from the purely data-driven learning at training time and still providing it with symbolic guidance during inference based on its interactions with a formal deductive reasoning system, such as a model checker or a theorem prover, as well as the constrained generation of tokens using grammars. Such a procedure has the advantage of creating provably correct circuit designs.

However, LLMs often create free-flowing text in their output which is not readily parsed into artifacts for formal deductive reasoning systems, and may not even follow the syntax of

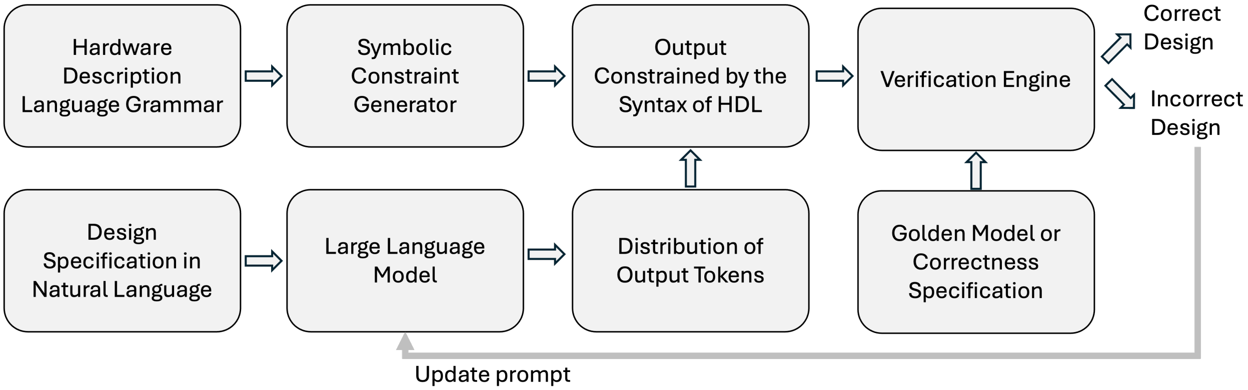


Fig. 1. Overview of our neuro symbolic approach to high-assurance synthesis of hardware circuits using grammar-constrained generation from large language models and feedback from formal deductive reasoning engines.

hardware description languages. Hence, we employ grammar-constrained decoding [5] in our approach to ensure that the emitted text is bound by the syntax of the Verilog language and can be parsed by Verilog simulators as well as model checkers. In this paper, we make the following contributions:

- We propose a neuro symbolic approach to synthesis of circuit designs that leverages both LLMs and feedback from formal deductive reasoning systems, such as model checkers.
- We show that the current generation of commercial and local LLMs, including GPT-4, Claude and Llama 3, are not able to directly produce reliable designs for relatively simple circuits, such as multipliers with small bit widths.
- We highlight that feedback generation in the setting of hardware designs is curtailed by the lack of structured response from models, and suggest the use of grammar-constrained decoding to create syntactically valid responses. Grammar-constrained decoding ensures that the generated response satisfies the syntax of the hardware description language, such as Verilog.
- We suggest that LLMs coupled with feedback from a verification engine are capable of synthesizing correct circuit designs, whose correctness is established by model checking. We demonstrate the success of our effort by automatically synthesizing a 4-bit multiplier.

II. APPROACH

Our approach combines the inductive capabilities of large language models with the deductive capabilities of formal methods, and extends our earlier work [14]–[16] on neuro symbolic reasoning from the planning domain to the arena of design synthesis using hardware description languages. In our methodology, the user provides three inputs:

(i) A high-level specification of the circuit is provided, such as a natural language statement asking the LLM to design a 4-bit multiplier design in Verilog.
(ii) A golden reference model or a specification is provided. It could be as straightforward as an assertion, such as the result of the multiplier with inputs a and b must be the product of a and b. It could also be more complex and involved Boolean or temporal logics in case of synthesis of complex controllers or other hardware models using state machines.
(iii) A grammar defining the language of the hardware description language is provided to aid in the decoding of the tokens from the LLM. This is a one-time effort and the grammar of languages like Verilog is readily available in the standard EBNF [17] syntax.

LLMs conjecture a distribution of tokens that represent the potential space of designs based on the instruction provided by the user, such as a 4-bit multiplier in Verilog. However, we do not greedily decode this distribution to simply find its highest probability member. Instead, we rely on grammar-constrained decoding [5]–[7] to identify a sequence of tokens that both represent high-probability tokens in the output of the LLM and rigorously satisfy the grammar of the hardware definition language. This ensures that the text being generated is syntactically-correct Verilog code and, hence, can be automatically parsed for downstream verification tasks, such as simulation or model checking.

The generated design is then verified by a verification engine against the corresponding specification or the golden model. The fastest incomplete checks can be performed using simulations and the resulting errors can be used to update the prompts to the LLM. This has the benefit of being fast as well as not sacrificing correctness. On the other hand, if simulations do not discover any errors, we pose the verification problem to a formal deductive system such as a model checker. Since circuits often do not terminate and usually execute forever, we employ k-induction [18]–[20] to verify that the circuit indeed satisfies the property at all times by establishing an inductive argument using a satisfiability solver.

If the formal deductive reasoning system establishes correctness of the design, it creates an artifact such as an unsatisfiable core that serves as a formal evidence of the correctness of the design. On the other hand, if the deductive reasoning leads to the discovery of a counterexample that violates the specification, the same is provided to the LLM as a prompt. An instruction-following LLM is unlikely to repeat the same design with the information provided in the prompt. However, it is possible that a sufficiently complex design can challenge the LLM and the generate-and-verify loop may not terminate without exhausting time and memory resources.

Several challenges remain in our neuro symbolic approach aided by grammar-constrained generation. First, the context of LLMs is often limited in practice and this limits the amount of information that can be provided to the LLM from the formal deductive reasoning system. Recent progress [21]–[24] has led to rapid rise in the size of contexts and there is ongoing work in providing infinite context windows. So, this problem may cease to be a bottleneck in the future. Second, while formal methods for verifying hardware has been well studied for at least the last four decades, the tools may not have the same scalability as LLMs. It is likely that LLMs can generate complex circuit designs that the current generation of model checkers may not be able to analyze. Our approach provides a new neuro symbolic approach to automated hardware circuit synthesis that couples advances in LLMs with the potential of model checkers in finding interesting non-trivial counterexamples as well as establishing formal correctness guarantees of designs.

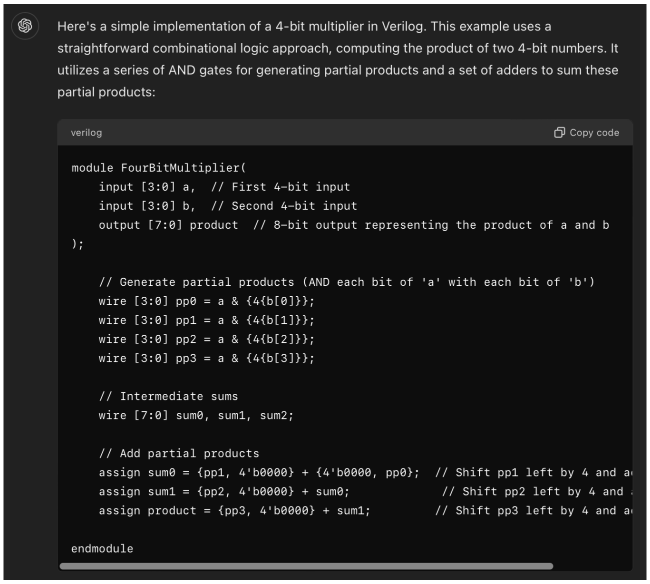
III. RESULTS

We tested our automated circuit synthesis approach on the design of a 4-bit multiplier without any additional constraints. Our analysis used both commercial off-the-shelf systems such as GPT-4 and Claude-3 Opus, as well as locally executed Llama 3 8B models whose output distribution could be further controlled using grammar-constrained decoding.

A. Direct Synthesis using LLMs

1) GPT-4: We prompted the state-of-the-art GPT-4 model to create a 4-bit multiplier in Verilog and received the response in Fig. 2. Our test bench harness showed that the design was not correct. For example, for the input a = 7 and b = 13, the expected result is 91. However, this circuit design produces 231. This is not surprising as the partial products have all been shifted left with the same number of bits, which is a glaring error. It is likely that future versions of GPT trained on additional Verilog models would correctly solve this specific problem. However, our approach of neuro symbolic synthesis using grammar-constrained decoding for LLMs and feedback from symbolic analysis of the generated code using formal methods may still be useful for larger hardware design problems.

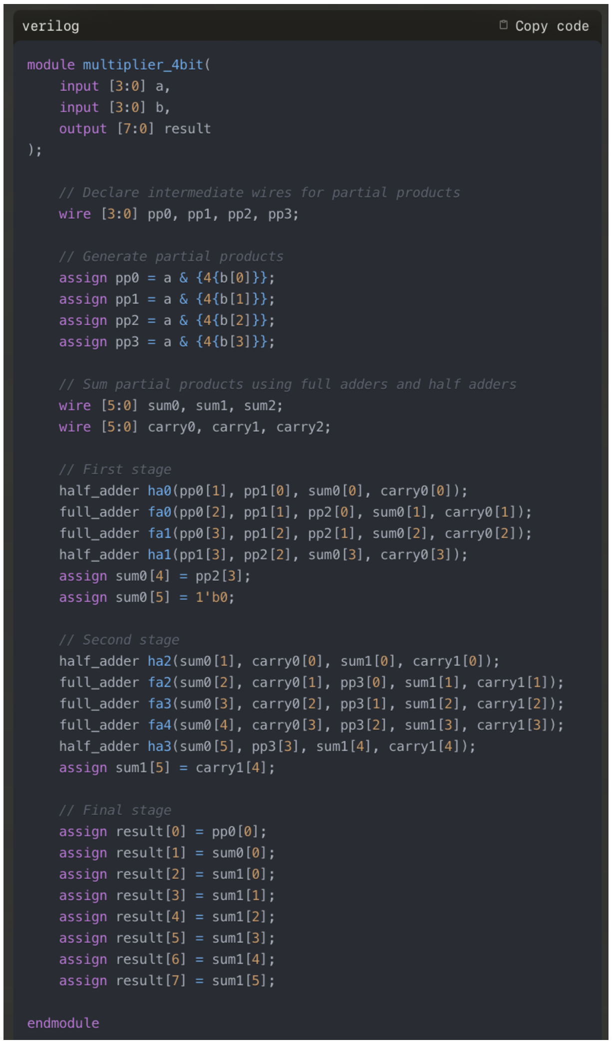
2) Claude 3: We prompted another state-of-the-art commercial AI model, Claude 3 Opus, to produce a 4-bit multiplier. Its response is shown in Fig. 3. Our test bench harness determines that the design is not correct. In particular, it does not produce the correct output 117 while provided with inputs a = 13 and b = 9.



Here's a simple implementation of a 4-bit multiplier in Verilog. This example uses a straightforward combinational logic approach, computing the product of two 4-bit numbers. It utilizes a series of AND gates for generating partial products and a set of adders to sum these partial products:

```verilog
module FourBitMultiplier(
    input [3:0] a,  // First 4-bit input
    input [3:0] b,  // Second 4-bit input
    output [7:0] product  // 8-bit output representing the product of a and b
);

    // Generate partial products (AND each bit of 'a' with each bit of 'b')
    wire [3:0] pp0 = a & {4{b[0]}};
    wire [3:0] pp1 = a & {4{b[1]}};
    wire [3:0] pp2 = a & {4{b[2]}};
    wire [3:0] pp3 = a & {4{b[3]}};

    // Intermediate sums
    wire [7:0] sum0, sum1, sum2;

    // Add partial products
    assign sum0 = {pp1, 4'b0000} + {4'b0000, pp0};  // Shift pp1 left by 4 and a
    assign sum1 = {pp2, 4'b0000} + sum0;             // Shift pp2 left by 4 and
    assign product = {pp3, 4'b0000} + sum1;         // Shift pp3 left by 4 and a

endmodule
```

Fig. 2. GPT-4's attempt at the design of a 4-bit multiplier is not correct.



```verilog
module multiplier_4bit(
    input [3:0] a,
    input [3:0] b,
    output [7:0] result
);

    // Declare intermediate wires for partial products
    wire [3:0] pp0, pp1, pp2, pp3;

    // Generate partial products
    assign pp0 = a & {4{b[0]}};
    assign pp1 = a & {4{b[1]}};
    assign pp2 = a & {4{b[2]}};
    assign pp3 = a & {4{b[3]}};

    // Sum partial products using full adders and half adders
    wire [5:0] sum0, sum1, sum2;
    wire [5:0] carry0, carry1, carry2;

    // First stage
    half_adder ha0(pp0[1], pp1[0], sum0[0], carry0[0]);
    full_adder fa0(pp0[2], pp1[1], pp2[0], sum0[1], carry0[1]);
    full_adder fa1(pp0[3], pp1[2], pp2[1], sum0[2], carry0[2]);
    half_adder ha1(pp1[3], pp2[2], sum0[3], carry0[3]);
    assign sum0[4] = pp2[3];
    assign sum0[5] = 1'b0;

    // Second stage
    half_adder ha2(sum0[1], carry0[0], sum1[0], carry1[0]);
    full_adder fa2(sum0[2], carry0[1], pp3[0], sum1[1], carry1[1]);
    full_adder fa3(sum0[3], carry0[2], pp3[1], sum1[2], carry1[2]);
    full_adder fa4(sum0[4], carry0[3], pp3[2], sum1[3], carry1[3]);
    half_adder ha3(sum0[5], pp3[3], sum1[4], carry1[4]);
    assign sum1[5] = carry1[4];

    // Final stage
    assign result[0] = pp0[0];
    assign result[1] = sum0[0];
    assign result[2] = sum1[0];
    assign result[3] = sum1[1];
    assign result[4] = sum1[2];
    assign result[5] = sum1[3];
    assign result[6] = sum1[4];
    assign result[7] = sum1[5];

endmodule
```

Fig. 3. Claude 3 Opus does not design a correct 4-bit multiplier.

Both GPT-4 and Claude-3 Opus represent state-of-the-art commercial systems that are significantly larger in size and complexity than the other local models explored in this paper. Failure to generate correct code by prompting shows the gap that currently exists between the capability of these models in software synthesis and in creating hardware designs.

While it is possible that larger models trained on more hardware data can solve this specific problem – the design of multipliers, our focus is to enable LLMs to synthesize provably correct hardware circuit designs by using feedback from a verification engine in the form of counterexamples.

3) Llama 3: We queried the Llama 3 8B and CodeLlama 7B models and directly tested their responses using the following prompt:

> Create a Verilog module for a n-bit binary multiplier. Please ensure the code is well-commented to explain each step of the process. Do not use this line in the code: assign product = a * b; The test suite will instantiate this module using:
> // Instantiate the multiplier module
> multiplier uut (
> .a(a),
> .b(b),
> .product(product)
>);
> Only output the code starting with module.

Both Llama 8B and CodeLlama 7B did not succeed in producing a response directly in our experiments. Llama 8B priduced the following response:

"Here is a 4-bit binary multiplier in Verilog:

` ` `verilog

```
module multiplier ( input [3:0] a, // 4-bit input A input [3:0] b, // 4-bit input
B output [7:0] product // 8-bit output product );
// Declare the necessary signals
wire [3:0] p0, p1, p2, p3;
// Partial products
wire [7:0] sum; // Accumulator for partial products
// Generate partial products
assign p0 = a[0] & b;
assign p1 = a[1] & b;
assign p2 = a[2] & b;
assign p3 = a[3] & b;
// Accumulate partial products
assign sum = p0 << 4|p1 << 3|p2 << 2|p3;
// Shift the sum to the left and add the next partial product
assign product = sum[7], sum[6], sum[5], sum[4], sum[3] & a[0], sum[2] &
a[0], sum[1] & a[0], sum[0] & a[0];
endmodule ` ` `
```

This code generates the partial products of the multiplication and then accumulates them. The partial products are shifted to the left and added to the next partial product. The result is the 8-bit product of the 4-bit inputs."

The generated code is not correct and our test bench harness identifies this. In particular, we find that the circuit does not correctly multiple 13 and 9 to produce 117.

Both Llama 3 8B and CodeLlama 7B failed to generate correct multiplier designs. This is not surprising given the failures of the larger GPT-4 and Claude 3 Opus models.

B. Synthesis with Feedback using LLMs

Inspired by earlier work on use of feedback in planning, we seek to use the instruction-tuned model and provide it feedback to improve the design. In particular, we provide the inputs that led to the incorrect design, the expected correct output, as well as the output produced by the LLM's previous design suggestion. A prompt with the following format is used:

> Create a Verilog module for a n-bit binary multiplier. Please ensure the code is well-commented to explain each step of the process. Do not use this line in the code: assign product = a * b; The test suite will instantiate this module using:
> // Instantiate the multiplier module
> multiplier uut (
> .a(a),
> .b(b),
> .product(product)
>);
> The following error {error message} was obtained on your last design: {design}. So, try a new idea. Only output the code starting with module.

The LLM does not always produce output in a structured manner and is hence difficult to parse. The parsing often fails after a few iterations. Hence, feedback by itself may not be adequate to establish a feedback refinement loop for the automated synthesis of circuits using LLMs.

```
module multiplier(
    input  [3:0] a,  // 4-bit input A
    input  [3:0] b,  // 4-bit input B
    output [7:0] product  // 8-bit output product
);

// Declare wires for partial products
wire [7:0] p0, p1, p2, p3;

// Declare wires for intermediate results
wire [7:0] temp0, temp1;

// Generate partial products for each bit of B
assign p0 = (a & {4{b[0]}}) << 0;  // No shift needed for
     the least significant bit
assign p1 = (a & {4{b[1]}}) << 1;  // Shift left by 1
assign p2 = (a & {4{b[2]}}) << 2;  // Shift left by 2
assign p3 = (a & {4{b[3]}}) << 3;  // Shift left by 3

// Generate intermediate results by adding partial
    products
assign temp0 = p0 + p1;
assign temp1 = p2 + p3;
assign product = temp0 + temp1;

endmodule
```

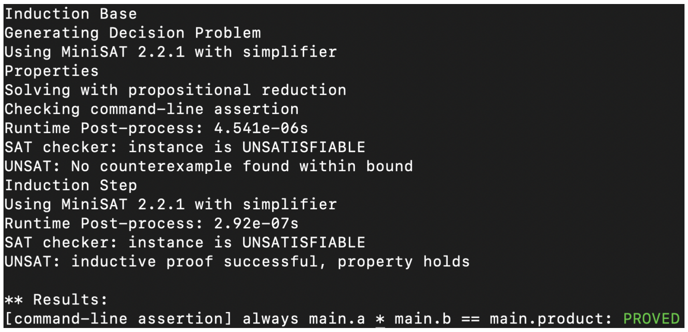
Fig. 4. Verilog code for a 4-bit multiplier generated by our approach.

C. Synthesis with Grammar Decoding and Feedback

In order to automatically execute Verilog models generated by the LLM, we enforce an approximate variant of a Verilog grammar on the output of the LLM. In grammar-based decoding, we do not simply emit the highest probability token that is produced by the LLM. Instead, we ensure that the token being emitted is a high-probability token that will fit within the formal grammar of the artifact being synthesized.

This enables us to automatically parse the output and invoke simulation as well as deduction tools on the hardware design being generated.

Using neuro symbolic feedback, we were able to synthesize the 4-bit multiplier design in Fig. 4. We ran a test bench harness of 1,000 random inputs using the Icarus Verilog simulation engine to gain confidence in the correctness of our design. We then employ the EBMC model checker with k-induction to verify the property "a $\times$ b $=$ product". With a bound of 1, the property could be established at all times, as shown in Fig. 5.



```
Induction Base
Generating Decision Problem
Using MiniSAT 2.2.1 with simplifier
Properties
Solving with propositional reduction
Checking command-line assertion
Runtime Post-process: 4.541e-06s
SAT checker: instance is UNSATISFIABLE
UNSAT: No counterexample found within bound
Induction Step
Using MiniSAT 2.2.1 with simplifier
Runtime Post-process: 2.92e-07s
SAT checker: instance is UNSATISFIABLE
UNSAT: inductive proof successful, property holds

** Results:
[command-line assertion] always main.a * main.b == main.product: PROVED
```

Fig. 5. The model checker EBMC formally proves that the multiplier design produced by the LLM is correct for all time.

IV. RELATED WORK

A. Large Language Models

The transformative potential of Large Language Models (LLMs) has been felt across numerous domains, including text generation, code synthesis, and even hardware design. These models, built on transformer architecture and its variations, have been pre-trained on vast internet-scale datasets, enabling them to perform a wide range of tasks without needing task-specific training [1]–[4]. However, high-assurance applications of LLMs in synthesizing hardware designs such as Verilog code are often plagued by outputs that lack correctness, requiring the need for formal verification and subsequent refinement.

B. Grammar Based Decoding

Grammar-based decoding involves constraining the output of LLMs to adhere to the specific syntax of target languages, such as Verilog. This method [5]–[7] ensures that the outputs are syntactically correct and can be parsed by subsequent tools like simulators and model checkers. It addresses the challenge of LLMs generating syntactically incorrect or incoherent code, which is particularly critical in high-assurance domains that require further processing of the response from the LLM.

C. Neuro Symbolic Feedback in LLMs

Integrating neuro symbolic feedback within the framework of LLMs offers a promising approach [14], [15], [25] to refine the outputs based on specific domain knowledge or feedback loops. This approach leverages the strengths of LLMs in generating potential solutions while employing symbolic methods to guide and correct the model's outputs in a targeted manner. By using feedback from verification processes, such as counterexamples from model checking, LLMs can iteratively improve towards generating correct and reliable designs. While earlier work has pursued such a neuro symbolic approach in the context of planning [14]–[16], we present a new application in this paper exploring the neuro symbolic approach in context of hardware design synthesis.

D. Counterexample Guided Synthesis

Counterexample Guided Synthesis (CEGIS) has emerged as a powerful paradigm [26]–[28] in automated program synthesis [29], where the synthesis process is iteratively refined based on counterexamples provided by a verification engine. In the context of hardware design, employing CEGIS allows for the automated generation of hardware descriptions that meet formal specifications, leveraging both the generative capabilities of LLMs and the rigor of formal verification methods to ensure the correctness of designs.

These related works collectively motivate our approach to high-assurance synthesis of hardware artifacts using LLMs, emphasizing the need for integrating diverse methodologies to address the limitations of current generation of LLMs in hardware design synthesis.

V. CONCLUSIONS AND FUTURE WORK

This work has demonstrated a new approach to the synthesis of hardware designs using Large Language Models (LLMs) integrated with formal verification methods. By combining the generative capabilities of LLMs with the rigorous verification provided by model checking, we have successfully synthesized correct-by-construction design for hardware circuit, such as 4-bit multiplier.

Our method leverages grammar-based decoding to ensure syntactic correctness and employs counterexample-guided synthesis to iteratively refine the designs based on feedback from the verification process. This approach not only enhances the reliability of the synthesized designs but also has the potential to significantly reduce the manual effort involved in hardware design and verification.

Several directions for future work remain open. We plan to investigate scaling the approach to handle more complex hardware designs and larger circuits, which will require improved context handling by LLMs to manage longer synthesis sessions. We aim to develop domain-specific optimizations in the training or fine-tuning process of the model to generate artifacts tailored for specific industries like automotive or aerospace.

Another longer-term goal is to achieve deeper integration with electronic design automation (EDA) tools, streamlining the transition from high-level synthesis to physical design. We will also explore expanding the methodology to areas such as analog circuit design, which remains predominantly manual and heuristic-based. Approaches such as attribution analysis [30] may be used to further enhance causal reasoning to create better feedback between LLMs and symbolic reasoning systems. Lastly, we plan to enhance the feedback mechanisms used in our process, integrating more symbolic approaches to highlight potential flaws, thereby improving the overall robustness of the automatically synthesized design.

We did not create a feedback loop with commercial tools such as GPT-4 and Claude Opus, as they currently do not provide the ability to control the generation of the tokens using grammar-constrained decoding. Hence, we would have to rely on ad hoc approaches for parsing the free-flowing text responses from these commercial systems.

REFERENCES

[1] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[2] Danny Driess, Fei Xia, Mehdi SM Sajjadi, Corey Lynch, Aakanksha Chowdhery, Brian Ichter, Ayzaan Wahid, Jonathan Tompson, Quan Vuong, Tianhe Yu, et al. Palm-e: An embodied multimodal language model. *arXiv preprint arXiv:2303.03378*, 2023.

[3] Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J Prenger, and Animashree Anandkumar. Leandojo: Theorem proving with retrieval-augmented language models. *Advances in Neural Information Processing Systems*, 36, 2024.

[4] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[5] Saibo Geng, Martin Josifoski, Maxime Peyrard, and Robert West. Grammar-constrained decoding for structured nlp tasks without fine-tuning. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 10932–10952, 2023.

[6] Saibo Geng, Martin Josifosky, Maxime Peyrard, and Robert West. Flexible grammar-based constrained decoding for language models. *arXiv preprint arXiv:2305.13971*, 2023.

[7] Chufan Shi, Haoran Yang, Deng Cai, Zhisong Zhang, Yifan Wang, Yujiu Yang, and Wai Lam. A thorough examination of decoding methods in the era of llms. *arXiv preprint arXiv:2402.06925*, 2024.

[8] Edmund Clarke, Ansgar Fehnker, Sumit Kumar Jha, and Helmut Veith. Temporal logic model checking. *Handbook of Networked and Embedded Control Systems*, pages 539–558, 2005.

[9] Sumit K Jha, Edmund M Clarke, Christopher J Langmead, Axel Legay, André Platzer, and Paolo Zuliani. A bayesian approach to model checking biological systems. In *Computational Methods in Systems Biology: 7th International Conference, CMSB 2009, Bologna, Italy, August 31-September 1, 2009. Proceedings 7*, pages 218–234. Springer, 2009.

[10] Edmund M Clarke, E Allen Emerson, and Joseph Sifakis. Model checking: algorithmic verification and debugging. *Communications of the ACM*, 52(11):74–84, 2009.

[11] Orna Grumberg and Helmut Veith. *25 years of model checking: history, achievements, perspectives*, volume 5000. Springer Science & Business Media, 2008.

[12] Edmund M Clarke, Thomas A Henzinger, and Helmut Veith. Introduction to model checking. *Handbook of Model Checking*, pages 1–26, 2018.

[13] Christopher James Langmead and Sumit Kumar Jha. Predicting protein folding kinetics via temporal logic model checking. In *International Workshop on Algorithms in Bioinformatics*, pages 252–264. Springer, 2007.

[14] Susmit Jha, Sumit Kumar Jha, Patrick Lincoln, Nathaniel D Bastian, Alvaro Velasquez, and Sandeep Neema. Dehallucinating large language models using formal methods guided iterative prompting. In *2023 IEEE International Conference on Assured Autonomy (ICAA)*, pages 149–152. IEEE, 2023.

[15] Sumit Kumar Jha, Susmit Jha, Patrick Lincoln, Nathaniel D Bastian, Alvaro Velasquez, Rickard Ewetz, and Sandeep Neema. Counterexample guided inductive synthesis using large language models and satisfiability solving. In *MILCOM 2023-2023 IEEE Military Communications Conference (MILCOM)*, pages 944–949. IEEE, 2023.

[16] Sumit Kumar Jha, Susmit Jha, Patrick Lincoln, Nathaniel D. Bastian, Alvaro Velasquez, Rickard Ewetz, and Sandeep Neema. Neuro symbolic reasoning for planning: Counterexample guided inductive synthesis using large language models and satisfiability solving, 2023.

[17] Richard Feynman. Ebnf: A notation to describe syntax. *Cited on*, page 10, 2016.

[18] Alastair F Donaldson, Leopold Haller, Daniel Kroening, and Philipp Rümmer. Software verification using k-induction. In *Static Analysis: 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings 18*, pages 351–368. Springer, 2011.

[19] Dirk Beyer, Matthias Dangl, and Philipp Wendler. Boosting k-induction with continuously-refined invariants. In *International Conference on Computer Aided Verification*, pages 622–640. Springer, 2015.

[20] Isil Dillig, Thomas Dillig, Boyang Li, and Ken McMillan. Inductive invariant generation via abductive inference. *Acm Sigplan Notices*, 48(10):443–456, 2013.

[21] Chi Han, Qifan Wang, Hao Peng, Wenhan Xiong, Yu Chen, Heng Ji, and Sinong Wang. Lm-infinite: Zero-shot extreme length generalization for large language models. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 3991–4008, 2024.

[22] Tsendsuren Munkhdalai, Manaal Faruqui, and Siddharth Gopal. Leave no context behind: Efficient infinite context transformers with infini-attention. *arXiv preprint arXiv:2404.07143*, 2024.

[23] Yiran Ding, Li Lyna Zhang, Chengruidong Zhang, Yuanyuan Xu, Ning Shang, Jiahang Xu, Fan Yang, and Mao Yang. Longrope: Extending llm context window beyond 2 million tokens. *arXiv preprint arXiv:2402.13753*, 2024.

[24] Bin Lin, Tao Peng, Chen Zhang, Minmin Sun, Lanbo Li, Hanyu Zhao, Wencong Xiao, Qi Xu, Xiafei Qiu, Shen Li, et al. Infinite-llm: Efficient llm service for long context with distattention and distributed kvcache. *arXiv preprint arXiv:2401.02669*, 2024.

[25] Susmit Jha. Lightning talk: Trinity-assured neuro-symbolic model inspired by hierarchical predictive coding. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–2. IEEE, 2023.

[26] Susmit Kumar Jha. *Towards automated system synthesis using sciduction*. University of California, Berkeley, 2011.

[27] Susmit Jha and Sanjit A Seshia. A theory of formal synthesis via inductive learning. *Acta Informatica*, 54:693–726, 2017.

[28] Alvaro Velasquez, Sumit Kumar Jha, Rickard Ewetz, and Susmit Jha. Automated synthesis of quantum circuits using symbolic abstractions and decision procedures. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2021.

[29] Zohar Manna and Richard Waldinger. Fundamentals of deductive program synthesis. 1992.

[30] Susmit Jha, Tuhin Sahai, Vasumathi Raman, Alessandro Pinto, and Michael Francis. Explaining ai decisions using efficient methods for learning sparse boolean formulae. *Journal of automated reasoning*, 63:1055–1075, 2019.