SS24 CSE814 Project Report

Sol Zilberman Ayaan Shaik

September 14, 2024

1 Problem Statement

Recent advances in deep learning have shown promising results in using large language models (LLMs) for downstream tasks such as program synthesis [2, 8]. Language models are trained on large corpus of text and use statistical learning to model the distribution of tokens (i.e., basic linguistic units such as characters or words) in an input dataset. When the training corpus includes code, these models are able to take as input program requirements and generate the corresponding programs [7]. However, existing research has shown that code generated by LLMs often contains faulty reasoning (i.e., 'hallucinations') that manifest as bugs in the generated software [11, 6]. To ensure that software is correct, formal verification can provide exhaustive proofs through deductive reasoning that a program meets its specification [10]. However, synthesizing programs from formal specification using symbolic/enumeration-based methods can be computationally expensive. To this end, our study aimed to explore how to leverage the generative capabilities of large language models for program synthesis and then integrate formal methods to ensure the generated programs are correct.

2 Proposed Solution

We have proposed a *neuro-symbolic* approach to program synthesis that synergistically integrates large language models for efficient program generation and formal verification to ground the generated programs in a deductive reasoning framework. Specifically, our work is motivated by counter-example guided inductive program synthesis (CeGIS) [1] and existing work in LLM-based iterative program generation [5, 4]. CeGIS is an iterative approach to program synthesis that takes as input a program specification, generates candidate programs, provides the programs to

a verification engine (e.g., SMT solver), and uses the verification output to inform incremental updates to the generated programs.

3 Implementation

Figure 2 shows an overview of our proposed solution, where rectangles denote processes and curved shapes denote data. A program specification ϕ is provided as input to the framework. First, the `prompt-builder` component generates a natural language prompt that instructs an LLM to generate a program from the specification ϕ (Step 1). Next, an LLM takes as input the constructed prompt and generates a candidate program C (Step 2). Our key novel contribution is the inclusion of the mutation engine (Step 3) that takes the LLM-generated program C and applies fine-grained mutations to provide a greater set of programs in the program space as candidate solutions to the program synthesis task. Finally, an SMT solver uses the specification ϕ as well as candidate programs c_i, where i is an integer from 0 to K, and K is a developer-specified hyperparameter for the mutation engine, and checks if the program satisfies the specification (Step 4). If a program c_i does satisfy ϕ, it is returned to the user as a correct program. Otherwise, the verification results, along with original program C are returned to the prompt builder that constructs a new prompt, now containing a counter-example, and repeats the process until a correct program is found or a termination criteria (e.g., number of attempts) is met.

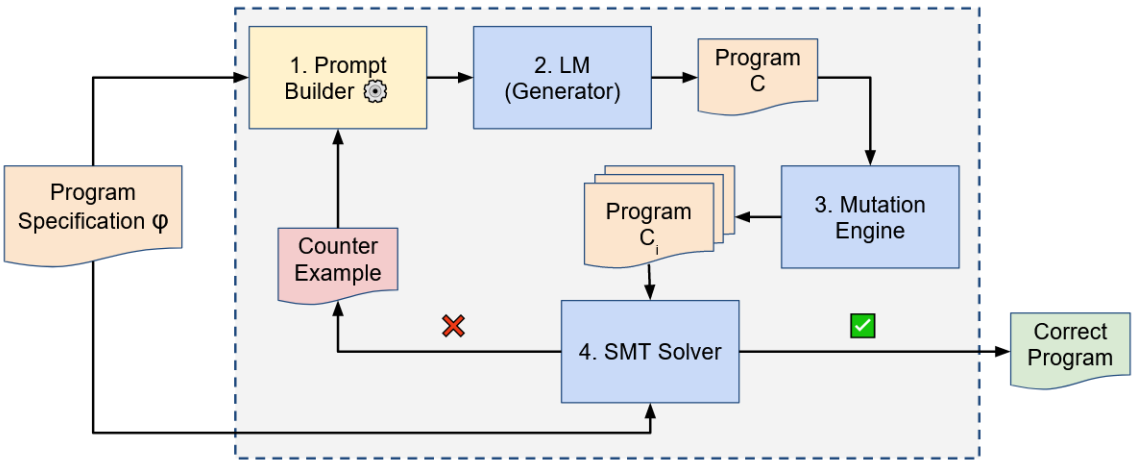


Figure 1: A data-flow diagram of our proposed solution.

3.1 BlocksWorld Environment

For testing and simulation, we created a Python class for the BlocksWorld game which keeps track of states and actions being performed on the blocks. We also have a PyGame[1] version for visualization of specific cases when we have a peculiar output generated by the LLM. Figure 2 shows an image form the PyGame visualization.

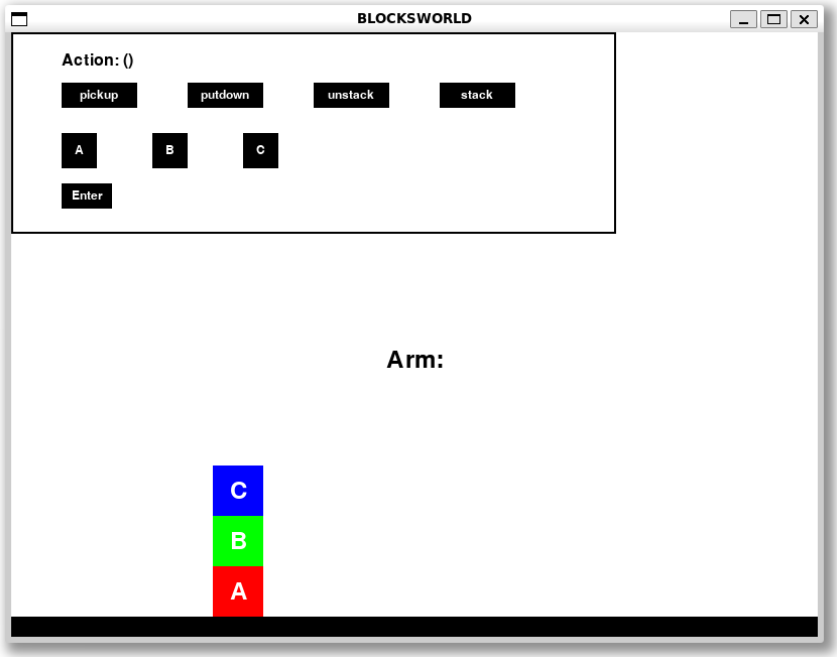


Figure 2: An image from the PyGame visualization.

3.2 CeGIS Loop

Below is the pseudo code for the CeGIS loop in our implementation. In our CeGIS implementation, the learner component is the LLM and the verifier component is a custom pyhton class specifically for BlocksWorld. We run a while loop that iterates over getting the output from the LLM, using the mutation engine to enumerate the search space, input into the verifier component to check completion. For every iteration, we add the incorrect code as a counterexample to the prompt to give to the LLM on the next run. When we reach a solution that satisfies the program specification, we break the loop and return that solution.

[1] https://www.pygame.org

Algorithm 1 CEGIS Loop

```
prompt ← LLM.build_prompt(problem, None)
solved ← False
while ¬solved do
    solution ← LLM.solve(prompt)
    solved, counter_example ← verifier.verify(solution)
    if ¬solved then
        prompt ← LLM.build_prompt(problem, counter_example)
    else
        print('Solution found!', solution)
        break
    end if
end while
```

3.3 Verification Engine

We use first-order logic to encode the constraints of the BlocksWorld environment. Specifically, a program in BlocksWorld is a sequence of instructions that update the state of the world. For each instructions, there are specific constraints that must be true about the current state of the world for that action to be performed. For example, for instruction `pickup(x)`, where x is a block, it must be the case that the 'arm' is currently free (i.e., not holding any other block). Listing 1 overviews the formal constraints for each of the four possible instructions in BlocksWorld.

Listing 1: Formal BlocksWorld constraints.

$$\texttt{pickup(x)} \rightarrow (\neg\texttt{Arm(x)} \wedge \texttt{ArmFree} \wedge (\forall\, y, y \neq x, \neg\texttt{Stacked(y,x)}) \wedge \texttt{Table(x)})$$

$$\texttt{putdown(x)} \rightarrow (\texttt{Arm(x)} \wedge \neg\texttt{ArmFree} \wedge \neg\texttt{Table(x)})$$

$$\texttt{stack(x, y)} \rightarrow (\texttt{Arm(x)} \wedge \neg\texttt{ArmFree} \wedge \neg\texttt{Table(x)} \wedge (\forall\, z, z \neq y, \neg\texttt{Stacked(z,y)}))$$

$$\texttt{unstack(x, y)} \rightarrow (\neg\texttt{Arm(x)} \wedge \texttt{ArmFree} \wedge \texttt{Stacked(x,y)} \wedge \neg\texttt{Table(x)} \wedge (\forall\, z, z \neq x, \neg\texttt{Stacked(z,x)}))$$

4 Validation Studies

This section overviews our experimental setup and the empirical results.

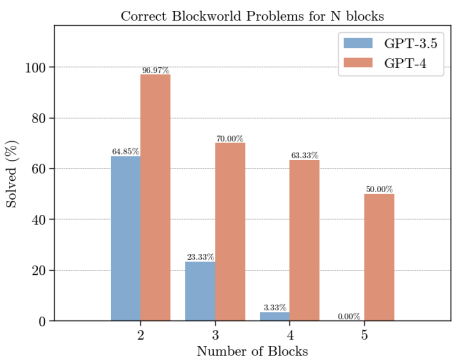
4.1 Experiment Setup

We ran experiments for different numbers of BlocksWorld problems ranging from 2 to 5 blocks with an average of 8 problems per blocks size. We run these experiments on two LLM models, GPT-3.5-Turbo [3] and GPT-4 [9]. We also ran experiments that have fuzzing (the mutation engine) enabled. We use the Z3 theorem prover[2] as the SMT solver for our experiments.

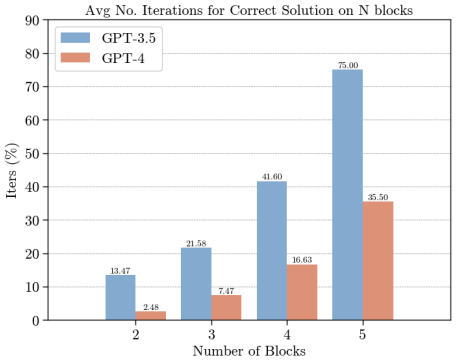
4.2 Results

Figure 3a shows the average accuracy of the proposed approach for solving ten BlocksWorld problems in worlds of size $N \in [2, 5]$. First, we note that as the number of blocks increased, the solve rate of both models (i.e., GPT-3.5, and GPT-4) decreased. Next, Figure 3b overviews the average number of iterations required to find a solution for problems in each world size. We observe that the number of iterations increased as the number of blocks considered increased. This is likely because the complexity of the problem increases as more blocks are considered, thus resulting in a lower solve rate and increased number of iterations needed when finding the correct solution. Importantly, we see that for all world sizes, the LLM took (on average) more than one iteration to arrive at the correct solution. This indicates that LLMs alone exhibit faulty reasoning and the addition of the our proposed framework could guide them towards the correct solution and improve their accuracy. Finally, Figure 3c compares the performance of each model within our framework, with and without the mutation engine enabled for each of the world sizes. We note that, on average, the addition of the mutation (i.e., fuzzing) engine yielded an improved solve rate. Our explanation for this is that the LLMs are good at approximating what a correct solution may *look like* (i.e., generating syntactically correct programs), but may lack the reasoning abilities to 'understand' the problem. Thus, the LLM-generated solutions are often close to the expected solution but have swapped a few instructions do to some stochastic variations/uncertainty. The mutation engine took the approximate solution from the LLM, which was quite close to the correct solution (i.e., in the space of all possible text sequences), and performed a fine-grained local search that enabled the framework to discover the correct program.

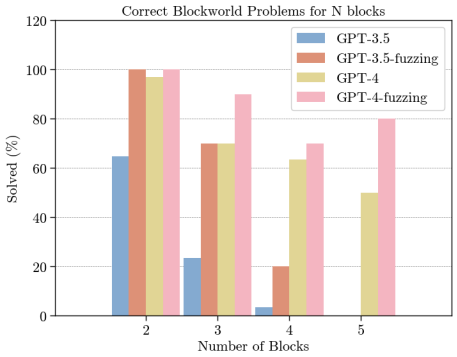
[2] https://z3prover.github.io/api/html/z3.html



(a) Average solve rate.



(b) Average iterations to find solution.



(c) Average solve rate with mutation engine.

Figure 3: Overview of results for average solve rate, average iterations, and average solve rate with mutation engine.

5 Threats to Validity

LLMs generate text based on stochastic algorithms, leading to slight variations in outcomes across repeated trials. To mitigate this variability, we averaged our results over several trials. It is important to note that we cannot assert that our instruction prompt is the definitive best for this task. Our approach to prompting LLMs was guided by commonly used techniques in the field, as evidenced by the literature.

6 Conclusion

In this work, we have proposed a framework that integrates LLMs and formal verification to address the task of program synthesis. We conducted several validation studies that indicated that the proposed approach improved the output of LLMs in generating code for BlocksWorld problems. By integrating a Counterexample-Guided Inductive Synthesis (CeGIS) loop and a mutation engine, we significantly improve the accuracy and efficiency of the code generated. Future research could extend to more complex code generation tasks (e.g., python3 code) and explore more sophisticated mutation engines (e.g., syntax-guided mutations, genetic algorithms) that could further improve the performance of the proposed framework.

References

[1] A. Abate, C. David, P. Kesseli, D. Kroening, and E. Polgreen. Counterexample guided inductive synthesis modulo theories. In *International Conference on Computer Aided Verification*, pages 270–288. Springer, 2018.

[2] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

[3] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[4] A. Desai, S. Gulwani, V. Hingorani, N. Jain, A. Karkare, M. Marron, and S. Roy. Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering*, pages 345–356, 2016.

[5] S. K. Jha, S. Jha, P. Lincoln, N. D. Bastian, A. Velasquez, R. Ewetz, and S. Neema. Neuro Symbolic Reasoning for Planning: Counterexample Guided Inductive Synthesis using Large Language Models and Satisfiability Solving, Sept. 2023.

[6] Z. Ji, N. Lee, R. Frieske, T. Yu, D. Su, Y. Xu, E. Ishii, Y. J. Bang, A. Madotto, and P. Fung. Survey of hallucination in natural language generation. *ACM Computing Surveys*, 55(12):1–38, 2023.

[7] G. Melis, C. Dyer, and P. Blunsom. On the state of the art of evaluation in neural language models. *arXiv preprint arXiv:1707.05589*, 2017.

[8] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

[9] OpenAI, J. Achiam, S. Adler, S. Agarwal, and et. al. Gpt-4 technical report, 2024.

[10] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal methods: Practice and experience. *ACM computing surveys (CSUR)*, 41(4):1–36, 2009.

[11] Z. Xu, S. Jain, and M. Kankanhalli. Hallucination is inevitable: An innate limitation of large language models. *arXiv preprint arXiv:2401.11817*, 2024.