Evaluating the Effect of LLMs on Program Invariant Synthesis*

Javier Gutierrez
Department of Computer Science
Amherst College
Amherst, MA, United States
jgutierrezbach26@amherst.edu

Tyler McCord
Department of Computer Science
Amherst College
Amherst, MA, United States
tmccord26@amherst.edu

Michael Perales
Department of Computer Science
Amherst College
Amherst, MA, United States
mperales24@amherst.edu

Theodore Woodward
Department of Computer Science
Amherst College
Amherst, MA, United States
twoodward25@amherst.edu

***Abstract*—Program invariants play a crucial role in formal verification, optimization, and automated reasoning. Yet, it remains challenging and computationally intensive to synthesize a program's invariants. The recent advances in large language models (LLMs) have shown their capabilities to be useful for a variety of programming tasks, such as generating code for a specified task. In this work, we build on previous works regarding LLMs and program invariants, offering our own analysis and evaluation. We investigate the capabilities and limitations of LLMs, using GPT-4o in generating program invariants for various benchmark programs. Additionally, we analyze the impact of various prompt engineering techniques on the accuracy of the invariants. We analyze the correctness, generality, and usefulness of these verification statements. Our findings reveal that while GPT-4o can generate a high volume of correct invariants, their usefulness varies significantly, with many being trivial. Prompt engineering techniques, particularly Few-Shot prompting, can enhance the generation of useful invariants, though no single technique consistently outperforms others across all benchmarks, and challenges remain for complex algorithmic properties.**

***Index Terms*—Large Language Models, Program Invariant Synthesis, Formal Verification, Prompt Engineering, Automated Reasoning, CBMC**

I. Introduction

Program invariants are essential to software development and verification, helping to ensure the correctness and safety of code through the entirety of execution. Synthesizing these invariants requires significant computational power and time, especially for large-scale programs. However, the rise of large language models allows us to potentially automate parts of this process. LLMs have demonstrated proficiency in solving various programming tasks such as code generation, bug fixing, and potentially program invariant synthesis. This project aims to evaluate the effectiveness of LLMs in generating these invariants for a variety of C program benchmarks. We investigate how different prompting techniques can affect the quality and correctness of the invariants. We use a formal verifier, CBMC, to assess the validity of invariants produced.

II. Related Work

There has been an increase in work towards building tools that exploit LLMs to generate invariants for program verification. Initial observations show that GPT models have the capability to synthesize invariants in a zero-shot setting, but require several samples and many calls to a verifier to be produced. To tackle this issue, a re-ranking tool was developed that can distinguish between inductive invariants and incorrect ones (Chakraborty et al., 2023) [1]. In a similar vein, LEMUR is a framework that uses LLMs to generate invariants and leverages formal reasoners to verify programs soundly (Wu et al., 2024) [2]. Further work was done to improve loop invariant synthesis in programs with more complex data structures and memory manipulations. The LLM-SE framework coordinates LLM with symbolic execution and uses fine-tuned models to generate invariants on these harder benchmarks (Liu et al., 2023) [3]. Another type of invariant that has been investigated is class invariants. ClassInvGen is a method that generates executable class invariants of high quality by leveraging LLMs' ability to synthesize pure functions, outperforming other pure LLM methods or data-driven inference techniques (Sun et al., 2025) [4].

III. Methodology

Generating and verifying program invariants using LLMs followed a distinct process. First, program benchmarks were collected by sourcing C programs from previously existing algorithms. The algorithms were collected to handle a variety of different data structures, such as traversing trees or graph structures.

With these algorithms, we used GPT-4o to prompt the LLM to generate invariants for the given program in the form of standard C `assert` statements. For each algorithm, the program is given to the LLM along with a specific prompt. The prompts used can be found in Appendix A. In this project, we considered:

- Traditional prompting

- Chain-of-thought prompting
- Few-shot prompting
- Role prompting

Thus, each algorithm is prompted to the LLM four times, once for each prompting method. The LLM returns code for a new program that should be nearly identical to the input code, just with various `assert()` statements throughout the program. It's important to note that we don't specify the type or quantity of invariants, but rather let the LLM determine what's important to verify.

CBMC is a Bounded Model Checker for C/C++ programs. For this project, CBMC serves as a symbolic verifier for the benchmarks and programs that we are testing. In a way, we are trying to formally verify the programs that we are testing on. CBMC played a crucial role in verifying the invariants produced by the LLM. CBMC is first used to verify that the program is valid before any assertions are made. In other words, the verifier checks that the program executes without serious errors. After the LLM outputs a new program with invariant assertion statements, the CBMC is used to verify that program as well. If the program is verified, then the invariants produced by the LLM are true. If the program fails, the incorrect invariants are identified and recorded.

A. Evaluation Methodology

Evaluating the capability of LLMs, specifically GPT-4o, on generating program invariants is not a clear cut process. State of the art models are very skillful in both writing and analyzing code and its properties, meaning they tend to synthesize valid assertions most of the time. Thus, quantitative evaluation methods are not as representative of the usefulness of invariants generated, since our experimentation found the model often generated many trivial or unhelpful assertions, which were still valid invariants for all possible inputs. For this reason, we perform qualitative analysis on the usefulness of these invariants in the context of helping programmers fix runtime errors by exposing their root causes. The criteria we came up with in order to categorize each invariant based on its usefulness is as follows:

1. **Useful Invariants**

- *Definition:* An invariant is useful if violating it would signal a semantic or logical error in the program's behavior, i.e., the program would be partially or wholly incorrect with respect to its specification.
- *Rules for Classification:*
  - It constrains functional correctness, such as optimality, consistency, or convergence.
  - It enforces key algorithmic properties (e.g., triangle inequality, sortedness, balance conditions).
  - It is not implied by syntax, variable declarations, or constant assignments.
  - Violating it would affect output, control flow, or data structure integrity.
  - It often relates to postconditions or loop-fixed points.

2. **Trivial Invariants**

- *Definition:* An invariant is trivial if it checks a property that is guaranteed by immediate syntax, language semantics, or initialization logic. These invariants offer no additional verification power and are redundant unless detecting memory corruption.
- *Rules for Classification:*
  - It checks a value immediately after declaration or assignment to a constant.
  - It asserts a condition that is structurally guaranteed (e.g., loop bounds, index ranges).
  - It confirms properties already enforced by static typing or domain constraints.
  - It would always hold unless the code is malformed or violates basic invariants (e.g., segmentation faults).

3. **Sometimes Useful Invariants**

- *Definition:* An invariant is sometimes useful if it does not directly enforce correctness, but can serve as a diagnostic or sanity check, help during debugging or partial implementation, or act as a guard against uncommon side effects or unexpected inputs.
- *Rules for Classification:*
  - It asserts a redundant or derived property, but not guaranteed by syntax.
  - It may help uncover implementation mistakes (e.g., forgotten guards or missed updates).
  - It's not strictly necessary for correctness, but aids program stability or maintainability.
  - Its utility depends on context (e.g., development vs. production).

One main takeaway from our experiments is that when nothing complex is happening in the code, the LLM might generate a trivial and useless invariant to satisfy the user's request for assertions.

Thus, in order to evaluate the invariants that we generated with GPT-4o, we analyze the proportions of the type of invariants for each prompting technique, in terms of their degree of usefulness. Moreover, we check which invariants fail when running the CBMC verifier, and annotate the usefulness of the discarded assertions.

IV. Results

A. Broad Findings

Our findings indicate that Large Language Models, specifically GPT-4o, are capable of generating a significant number of syntactically valid program invariants. These invariants are largely semantically correct, as verified by the CBMC bounded model checker. Failed assertions were infrequent across both algorithms and prompting techniques, demonstrating a consistently high level of validity for the generated invariants. However, the quality of these invariants, measured by their usefulness for verification of program correctness, varied considerably. This observed variation was influenced heavily by both the program invariants were generated for, and the prompt engineering technique employed to generate them.

B. Impact of Prompt Engineering on Invariant Quality

Our experiments did not identify a single prompting technique that consistently outperformed all others across every benchmark. Few-Shot prompting demonstrated the best average performance, and generated the highest overall number of ”Useful” invariants (25 across all algorithms) compared to our Baseline (13), COT (19), and Role prompting (15) prompts. This performance suggests that providing a few high-quality examples significantly improves the ability of LLMs to generate more relevant and insightful assertions.

Each prompting technique (Baseline, Few Shot, COT, Role) managed to be the top performer or tied for top performer in generating ”Useful” invariants for at least one problem instance, underscoring the context-dependent nature of prompt effectiveness.

In the case of the DFS algorithm, the Baseline prompt performed as well (8 useful invariants) as the best engineered prompt (COT, also with 8 useful invariants). This result suggests that for certain problems or aspects of invariant generation, more elaborate prompt engineering might not yield substantial improvements over a simple, direct request.

There were algorithms where all prompting techniques struggled to generate a large number of ”Useful” invariants. For example, in the Bellman-Ford algorithm, the maximum number of ”Useful” invariants generated by any single prompt was only 2 (by Few Shot). This indicates that certain complex algorithms and their properties might be inherently more challenging for current LLMs to capture through invariants, regardless of the prompting strategy tested.

C. Quality of Generated Invariants

The generated invariants classified as ”useful” by our evaluation metric would significantly aid programmers in determining if a program performed as expected, or in the case of runtime errors, where those errors occurred. For instance, when `malloc` is used to allocate memory, invariants are often generated to verify that the allocation and references are valid. More complex invariants exhibit understanding of the objective of the program, such as when invariants check partial sorting in Insertion Sort, or after the termination of the Floyd Warshall algorithm, verify that triangle inequality holds for all nodes in a graph.

While our experiments demonstrated LLMs were capable of generating useful invariants, we observed that our prompts tended to generate as many or more ”Sometimes useful” and ”Trivial” invariants than ”Useful” invariants, as defined in our Evaluation Methodology.

These trivial invariants included assertions about loop counter bounds that are syntactically guaranteed (e.g., `assert(i>=0 && i<v)` inside a `for (int i=0; i<v; i++)` loop) or properties of boolean variable relationships (e.g., `assert(visited[i] == true || visited[i] == false);`) which must by definition be true.

TABLE I
BELLMAN-FORD ASSERTION CATEGORIES AND FAILURES BY PROMPT TECHNIQUE

Prompt Technique	Failed Assertions	Useful	Sometimes Useful	Trivial
Baseline	0	0	8	4
Few Shot	1	2	3	4
COT	0	0	6	8
Role	0	1	2	4

TABLE II
CYCLE SORT ASSERTION CATEGORIES AND FAILURES BY PROMPT TECHNIQUE

Prompt Technique	Failed Assertions	Useful	Sometimes Useful	Trivial
Baseline	0	0	5	4
Few Shot	0	3	3	1
COT	1	1	3	14
Role	0	4	0	0

TABLE III
DFS ASSERTION CATEGORIES AND FAILURES BY PROMPT TECHNIQUE

Prompt Technique	Failed Assertions	Useful	Sometimes Useful	Trivial
Baseline	0	8	10	1
Few Shot	0	5	10	4
COT	0	8	14	5
Role	0	2	2	4

TABLE IV
INSERTION SORT ASSERTION SUMMARY BY PROMPT TECHNIQUE

Prompt Technique	Failed Assertions	Useful	Sometimes Useful	Trivial
Baseline	0	2	0	3
Few Shot	0	4	1	2
COT	0	4	2	1
Role	1	3	1	0

TABLE V
FLOYD-WARSHALL ASSERTION SUMMARY BY PROMPT TECHNIQUE

Prompt Technique	Failed Assertions	Useful	Sometimes Useful	Trivial
Baseline	1	2	3	0
Few Shot	1	5	1	3
COT	0	1	5	6
Role	1	3	0	1

TABLE VI
PRIM’S ALGORITHM ASSERTION SUMMARY BY PROMPT TECHNIQUE

Prompt Technique	Failed Assertions	Useful	Sometimes Useful	Trivial
Baseline	0	1	3	4
Few Shot	1	6	2	7
COT	1	5	0	12
Role	0	2	4	14

V. Conclusion

In essence, we discovered that there is some promise in using LLMs to synthesize program invariants for run-time bug detection. Performance of different prompting techniques seemed to be irregular, with a slightly better performance for more complex prompts in comparison to the baseline. Additionally, we observed that there are some invariants generated that are invalid, and many that are trivial in the sense that they do not convey helpful information to detect any issues with a given program. However, some useful invariants are created, and we believe that LLMs coupled with formal verifiers can make the debugging process for programmers less tedious.

References

[1] S. Chakraborty, S. K. Lahiri, S. Fakhoury, M. Musuvathi, A. Lal, A. Rastogi, A. Senthilnathan, R. Sharma, and N. Swamy, "Ranking LLM-Generated Loop Invariants for Program Verification," in *Findings of the Association for Computational Linguistics: EMNLP 2023*, Singapore, Dec. 2023, pp. 8937–8952. Available: https://aclanthology.org/2023.findings-emnlp.614.pdf

[2] H. Wu, C. Barrett, and N. Narodytska, "Lemur: Integrating large language models in automated program verification," in *Proc. Twelfth Int. Conf. Learn. Represent. (ICLR)*, Vienna, Austria, May 2024. Available: https://doi.org/10.48550/arXiv.2310.04870

[3] C. Liu, X. Wu, Y. Feng, Q. Cao, and J. Yan, "Towards general loop invariant generation: A benchmark of programs with memory manipulation," in *Proc. Thirty-eighth Conf. Neural Inf. Process. Syst. Datasets and Benchmarks Track (NeurIPS)*, New Orleans, LA, USA, Dec. 2024. Available: https://doi.org/10.48550/arXiv.2311.10483

[4] C. Sun, V. Agashe, S. Chakraborty, J. Taneja, C. Barrett, D. Dill, X. Qiu, and S. K. Lahiri, "ClassInvGen: Class Invariant Synthesis using Large Language Models," *arXiv preprint arXiv:2502.18917*, Feb. 2025. Available: https://doi.org/10.48550/arXiv.2502.18917

Appendix

A. Baseline Prompt

Generate valid program invariants helpful for bug detection for the following code in C. Invariants should be added into the code in standard C assertion format.

B. Chain of Thought Prompt

You are given a C program. Your task is to read and understand the code and generate valid program invariants in the form of C `assert(...)` statements. These assertions should represent conditions that are guaranteed to be true at specific points during the execution of the program. Follow these guidelines:

- Identify logical invariants before and after loops, function calls, and conditionals.
- Express invariants as valid C `assert(...)` statements.
- Ensure that each invariant is semantically correct and holds across all executions, not just for specific inputs.
- Include variable bounds, relations (e.g., `x>=0`, `a[i]==i*2`), and loop invariants where appropriate.
- Do not make assumptions based on undefined behavior.
- Output the original code, and annotate it with your generated assertions in appropriate places (as if inserting them into the source).

Here's the C code:

C. Role Prompting

You are a Distinguished Professor of Formal Methods, specializing in the automated synthesis of precise program invariants for safety-critical systems. Your goal is to identify the strongest and most complete set of loop invariants necessary for full program verification, particularly those that would be useful as input for an SMT solver like Z3. Focus on mathematical rigor and invariants that capture the core properties of the loop. Generate the invariants in the form of C assertions and give me back the full original code.

D. Few Shot Prompting

Your goal is to identify all non-trivial invariants that capture the essential relationships between variables and their bounds. Generate the invariants in the form of C assertions and give me back the full original code.

High-Quality Examples of Invariants in the format of C assertions:

Example 1:

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

// Displays the array
void display(int *arr, int n)
{
    int i;
    for (i = 0; i < n; i++)
    {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

// Swaps two integers
void swap(int *first, int *second)
{
    int temp = *first;
    *first = *second;
    *second = temp;
}

// Cycle sort with invariants for formal verification
void cycleSort(int *arr, int n)
```

```
{
    int writes = 0;
    for (int cycle_start = 0; cycle_start <= n - 2; cycle_start++)
    {
        // Invariant: arr[0..cycle_start-1] is sorted
        // and contains the smallest elements
        for (int i = 0; i < cycle_start - 1; i++) {
            assert(arr[i] <= arr[i + 1]);
        }

        int item = arr[cycle_start];
        int pos = cycle_start;

        // Count elements less than item to find correct position
        for (int i = cycle_start + 1; i < n; i++) {
            if (arr[i] < item)
                pos++;
        }

        // Invariant: pos equals number of elements < item
        // in arr[cycle_start+1..n-1]
        int count = 0;
        for (int i = cycle_start + 1; i < n; i++) {
            if (arr[i] < item) count++;
        }
        assert(pos == cycle_start + count);

        if (pos == cycle_start)
            continue;

        while (item == arr[pos]) pos += 1; // Corrected from PDF's '=' to '=='

        if (pos != cycle_start)
        {
            swap(&item, &arr[pos]);
            writes++;
        }

        while (pos != cycle_start)
        {
            pos = cycle_start;
            for (int i = cycle_start + 1; i < n; i++)
                if (arr[i] < item)
                    pos += 1;

            while (item == arr[pos]) pos += 1; // Corrected from PDF's '=' to '=='

            if (item != arr[pos]) // This condition seems to always be true if item was swappe
            {
                swap(&item, &arr[pos]);
                writes++;
            }
            // Invariant: no element is lost or duplicated;
            // item is being placed into correct final position
            // Item will eventually reach its correct location
        }
    }
    // Post-loop invariant: arr[0..cycle_start] is now sorted
    // This looks like it should be outside the outer loop, or refer to n-1
    // For the last cycle_start, it would be arr[0...n-2]
    // The PDF has this inside the loop, which might be a typo.
    // For now, keeping as per PDF structure.
    // for (int i = 0; i < cycle_start; i++) { // cycle_start here might be an issue after loo
    //    assert(arr[i] <= arr[i+1]);
    // }


    // Final invariant: array is sorted
    for (int i = 0; i < n - 1; i++) {
        assert(arr[i] <= arr[i + 1]);
    }
}

int main()
{
    int arr[] = {20, 40, 50, 10, 30}; // Sample input array
    int n = sizeof(arr) / sizeof(arr[0]); // Compute size of array
    printf("Original array: ");
    display(arr, n);
    cycleSort(arr, n); // Corrected from PDF's cycleSort(arr,  );
    printf("Sorted array: ");
    display(arr, n);
    return 0;
}
```

end of examples. Beginning of C code for task

The full prompt with examples 2 and 3 can be seen in the Github Repository at: https://github.com/TH30DOR3/Program-Invariant-Synthesis-Using-Large-Language-Models/blob/main/Prompts/Standard.txt