LLMs for Verification (CS521 Project Report)

AMAN SINGH and ASHWIN LAMANI

1 INTRODUCTION

This report summarizes the work we did in the project. While working on the project we changed the initial problem statement a couple of times according to the observations from our experiments and the papers we read. The content of the report is divided into different sections as follows: first, we discuss the most general and useful problem in Section 2. Due to several limitations, we realize that this is a very difficult problem to solve and hence look at a simpler problem of synthesising assert statements from natural language in Section 3. As this simpler problem is tractable, we move on to a more involved problem of synthesizing postconditions in Section 4. After this, we change gears and talk about a serious problem with evaluation of LLMs on code generation and our attempts to fix it in Section 5. In Section 6, we end the report with how we would like to extend this work given more time and resources.

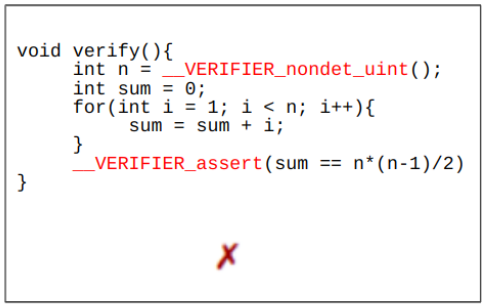
All the code for the project can be viewed at https://github.com/amansingh-13/cs521-project.

2 LLMS FOR FORMAL VERIFICATION

2.1 Problem Description

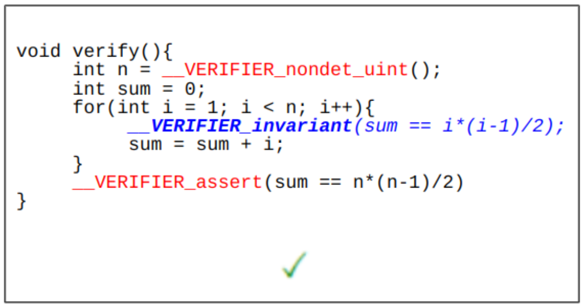
Formal verification of programs is a challenging task. A significant challenge in formal verification is to prove that a bad state, i.e. division by zero, assertion failure, etc. is never reached. This problem can be solved by computing all the possible states the program can be in at each line of the source code. However, computing how the state of the program changes across a loop requires knowing an *invariant* of the loop.

Verification tools like CPAchecker [1] have invariant synthesis algorithms but they are not very effective. In the code presented below, the one on the left can't be verified by CPAchecker. However, if given a good invariant like in the one on the right, the verification is successful. As we can observe, the invariant needed is not very complicated but it is still difficult to generate. Most verification tools rely on the creativity of the programmer to come up with invariants. This presents a good use case for LLMs to exploit its creativity in generating invariants to beat state-of-the-art verification softwares.



```
void verify(){
     int n = __VERIFIER_nondet_uint();
     int sum = 0;
     for(int i = 1; i < n; i++){
          sum = sum + i;
     }
     __VERIFIER_assert(sum == n*(n-1)/2)
}
```

✗



```
void verify(){
     int n = __VERIFIER_nondet_uint();
     int sum = 0;
     for(int i = 1; i < n; i++){
          __VERIFIER_invariant(sum == i*(i-1)/2);
          sum = sum + i;
     }
     __VERIFIER_assert(sum == n*(n-1)/2)
}
```

✓

2.2 Observations

In a recent paper [9], the authors discuss the performance of LLMs in generating loop invariants. The benchmark used for this study is [2]. The main observation from the paper is that the list of candidate invariants generated

.

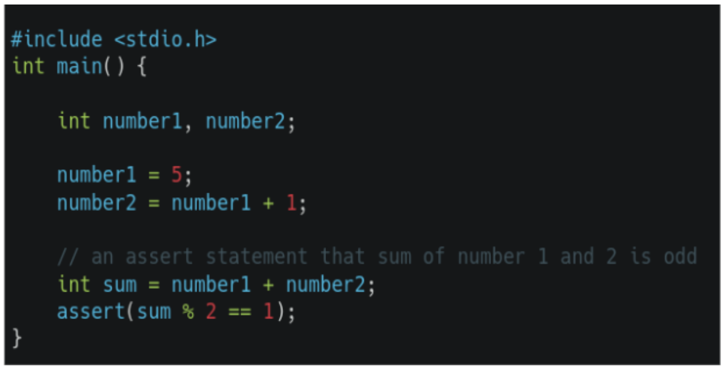
by LLMs is not good enough. For GPT4, the pass@10 accuracy, i.e., the probability that you get at least 1 correct invariant given 10 tries is 50.6%. To improve this, they create a ranking mechanism for invariants by training a model M such that $M(problem)$ is close to $M(candidate\_invariant)$ for correct invariants. Doing this increases the pass@10 accuracy for GPT to 81.9%. However, it still does not beat state-of-the-art invariant synthesis tools like LoopInvGen [3] (which does not use deep learning methods) by a big margin.

This paper provides evidence that generating useful invariants is difficult. This task is more like theorem proving than simple code generation (which seems doable by current LLMs) in the sense that it requires more creativity and trial-and-error. Hence a novel way to tell an LLM that a solution is incorrect and ask it to improve it would be needed.

3 LLMS FOR ASSERT STATEMENT SYNTHESIS

3.1 Problem Description

We now study a much simpler problem. If useful invariants are tough to generate, can simple assert statements be generated given some code and a comment describing the assertion. For example, in the image below, everything up to and including the comment is passed on to an LLM which is prompted to finish the code.


```
#include <stdio.h>
int main() {

    int number1, number2;

    number1 = 5;
    number2 = number1 + 1;

    // an assert statement that sum of number 1 and 2 is odd
    int sum = number1 + number2;
    assert(sum % 2 == 1);
}
```

3.2 Approach

We use the opensource model `codeLLaMa-7B` as it was the only model we could run on our laptops, and `gpt-3.5`. A problem we faced is that there are no well known benchmarks for this problem. We searched public repositories and even though we could find code which extensively used assert statements, none of them described the assertions in natural language. To solve this problem, we created a small benchmark of 20 C/C++ files. An example of a benchmark file is the image above. As this benchmark is small enough, we could verify the correctness of generated assert statements manually.

3.3 Results

We observed that the LLMs are good at generating correct assert statements. `gpt-3.5` could generate 17/20 assertions that were correct. Even a smaller model like `codeLLaMa-7B` could get 13/20 assertions correct. Another thing to note is that this performance is what we get out of the box - we did not provide any special prompts, for example, teaching the LLM what an assert statement means or giving examples in the prompt.

Although 20 examples is not enough to conclude anything significant, we can say that LLMs understand what assertions are with this experiment. They can understand natural language descriptions of assertions and figure out which variables to use and how to use them in generating the assert statement.

3.4 Code

The code for this part can be viewed at https://github.com/amansingh-13/cs521-project/tree/master/llm_assert/. The important files for this part are as follows:

```
llm_assert/
|-- gpt_test.py      - code to test using gpt-3.5
|-- llama_test.py    - code to test using codeLLaMa-7B
`-- abench/          - contains the benchmark files for testing
```

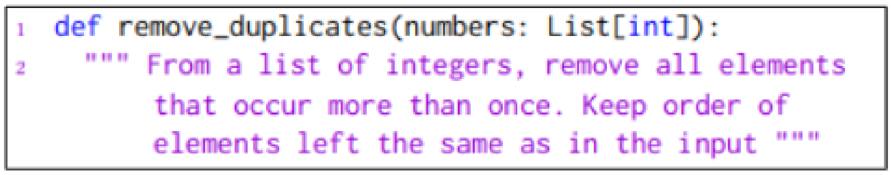
4 LLMS FOR POSTCONDITION SYNTHESIS

4.1 Problem Description

Once we found that LLMs are capable of generating assert statements by interpreting natural language in the form of comments, we went on further to check their ability to synthesize high-quality postconditions from natural language intent in a surrounding context as discussed in [10]. We define a postcondition for a method to be an assertion that holds true for any successful execution of the method given the input and output states of the method.

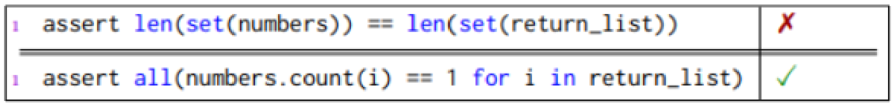
4.2 Motivation

Consider an example where the programmer intends to remove all the duplicates from a given list of integers. Given that the programmer documents the function intention as in the comments shown below, it can be seen that the natural language definition is ambiguous. It can be interpreted in two ways - (1) remove all occurrences of numbers that appear more than once (2) remove all but one occurrences of numbers that appear more than once.



```
def remove_duplicates(numbers: List[int]):
  """ From a list of integers, remove all elements
      that occur more than once. Keep order of
      elements left the same as in the input """
```

The below image shows two postconditions generated by the LLM of which the second one matches with the user intent but the first one is an incorrect assertion. Hence, LLMs can be leveraged to generate postconditions in



`assert len(set(numbers)) == len(set(return_list))`	✗
`assert all(numbers.count(i) == 1 for i in return_list)`	✓

the form of assert statements that can used to verify the user's intent or to prove buggy programs as incorrect.

4.3 Research questions

Having seen that LLMs can be prompted to generate assert statements for a program given natural language description, we based our experiments to answer the following two questions:

(1) Given a natural language description of a method, can the LLM generate postconditions to check the correctness of the method's implementation?
(2) Given a natural language description of a method and a candidate postcondition, how good is the given post condition?

4.4 Approach

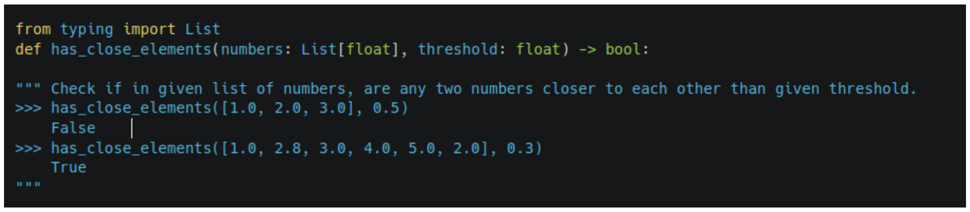
We used the EvalPlus dataset [11] for our experiments. The steps followed during this experiment are as follows:

(1) Generate postconditions using an LLM by providing the prompts and base test cases.
(2) Check if the reference implementation along with the generated postconditions pass all the test cases.
(3) Generate buggy code for the same prompt using an LLM and inserting bugs. Then check if these buggy implementations can be identified by the postconditions through some test case.

We generate buggy code to test the quality of generated postconditions by asking an LLM to give us 10 different implementations of the same function and then performing operations like - replacing $\geq$ with $>$, $\leq$ with $<$, etc.

"Okay" postconditions are the ones for which the reference implementation plugged in with the postcondition passes all the given test cases. "Good" postconditions are the ones which can falsify a program consisting of bugs when plugged in with the postcondition.

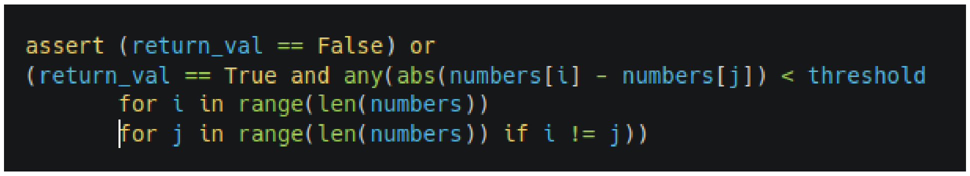
4.4.1 ***Example.*** The below image shows an example of a prompt from the EvalPlus dataset [11].



```
from typing import List
def has_close_elements(numbers: List[float], threshold: float) -> bool:

""" Check if in given list of numbers, are any two numbers closer to each other than given threshold.
>>> has_close_elements([1.0, 2.0, 3.0], 0.5)
    False
>>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
    True
"""
```

On prompting the LLM by plugging in the prompt in the template shown above, the following shows the postcondition that was generated.



```
assert (return_val == False) or
(return_val == True and any(abs(numbers[i] - numbers[j]) < threshold
        for i in range(len(numbers))
        for j in range(len(numbers)) if i != j))
```

The above generated postcondition is an example of an "okay" postcondition since if the *return_val* is true then there is an extra check but if it is false, then the check is passed irrespective of what the program logic is. Since this is somewhat trivial, this is not a "good" postcondition since a program that always returns false always passes this assert statement.

4.5 Formulation

The following formalises the approach to measure:

- Postcondition correctness:

$$\forall\, i \in T : eval(post, (i, P(i))) == true$$

  A postcondition is "okay" if for all the test cases, the correct program P plugged in with the postcondition for all input-output pairs $(i, P(i))$ evaluates to true.
- Quality of postcondition:

$$\exists\, BP \;\; \exists\, i \in T : eval(post, (i, BP(i))) == false$$

An "okay" postcondition is a "good" postcondition if there exists a buggy program (*BP*) and a test case ($i \in T$) such that the buggy program plugged in with the postcondition for an input and generated output evaluates to false.

4.6 Results

Given the limitations on API requests to `gpt-3.5`, we chose only **45** EvalPlus prompts and generated **5** postconditions using the LLM. The following table consists of the results obtained from the experiments:

EvalPlus Prompts	45
No. of generated postconditions	225
"Okay" postconditions	68
"Good" postconditions	36
Percentage of "okay" postconditions	30%
Percentage of "good" postconditions	16%

We found that 30% of the total postconditions generated passed the test cases present in the EvalPlus dataset. Among these, 53% of the postconditions were able to prove buggy programs as incorrect and hence only 16% of the total generated postconditions were "good".

4.7 Conclusion

We can see that although `gpt-3.5` can generate postconditions using the natural language specification, only 16% are actually good to verify program code. It should be noted that no training of LLM was done before postcondition generation and the data sample is substantially small to make significant claims. However, it points us to a direction where the postcondition generation can be improved maybe by providing better prompts, training the LLM before code generation, etc.

4.8 Code

The code for this part can be viewed at https://github.com/amansingh-13/cs521-project/tree/master/llm_post/. The important files for this part are as follows:

```
llm_post/
|-- bugger.py          - code to add bugs
|-- gpt_code_test.py   - code to generate programs for prompts using gpt-3.5
|-- gpt_post_test.py   - code to generate assert statements using gpt-3.5
|-- llama_code_test.py  - code to generate programs for prompts using codeLLaMa-7B
|-- llama_post_test.py  - code to generate assert statements using codeLLaMa-7B
|-- post_accuracy.py   - code to check if postcondition is "okay"
|-- post_efficacy.py   - code to check if postcondition is "good"
`-- benchmarks/        - contains the benchmark files for prompts
```

5 VERIFYING CODE GENERATION BY LLMS

5.1 Problem Description

HumanEval [4] is a popular benchmark for evaluating the code generation ability of LLMs. The benchmark has 4-5 test cases of input-output pairs for each problem, and code generated is claimed to be correct if it passes these test cases. However, a recent paper [11] has shown that the 4-5 test cases are not enough. By cleverly creating test cases, they increase the number of input-output pairs to 320-400 and show that the codegen accuracy drops. However, we know that just using test cases is not enough to verify the correctness of programs. So, we can use formal verification to check if the generated code outputs different values than the reference implementation for all possible inputs.

5.2 Challenges

We do not have any results in this section as the task seemed to be difficult for reasons out of the scope of the project. We document our failed approaches in this section.

(1) We first tried to use the HumanEval [4] benchmark which has reference implementations in Python along with the Nagini [5] verification tool. However, we found that even for correct code, Nagini would find incorrect counterexamples. Upon closer inspection, we learned that counterexample generation in Nagini is experimental which explains the unsoundness.
(2) Next, we tried using the MBJP [6] benchmark which has reference implementation in Java along with the JBMC [7] verification tool. In this case, we found that JBMC is not a very powerful verifier. It could not find counterexamples even for simple programs with loops.
(3) After this, we used the MBCPP [6] benchmark which has reference implementation in C++ with CPAchecker [1] verification tool. CPAchecker is a powerful verification tool but it works only with C programs. So, we removed the programs from the benchmark that used C++ specific features (this got us down to 200 examples from 800). We could find a lot of counterexamples now. But, this attempt failed too as we found that the more than half the reference implementations in MBCPP were plain wrong.

6 FUTURE WORK

6.1 Real-World Applications

The accuracy of postcondition generation, as it currently stands is not satisfactory. One direction in which we can extend this work is by improving the postcondition generation accuracy. This can be done using better prompts and testing on better and *instructed* models. We could not do a lot of tinkering with the prompts, as we did not have powerful machines to rerun the evaluation multiple times for `codeLLaMa-7B` and because of the 200 requests per day limit for `gpt-3.5`.

The reason why generating postconditions is useful is because, theoretically, the generation of good postconditions should be easier than generating complete code (since code for functions is usually longer than the postcondition, it has significantly greater chance of containing a bug). If that is the case, the generated postconditions can be used alongside a verification tool to guide the generation of code by LLMs. It would work by verifying if each block of generated code confirms with a postcondition for the block using a verification tool. Overall, it would increase the accuracy of code generation by LLMs and prevent hallucinations.

6.2 Fair evaluation of LLMs

We know from [11] that the current evaluation methods for code generation by LLMs are not good enough. Completing the work we did in Section 5 would be an important project as a ‘formal’ extension of the work done in [11]. Additionally, we found program synthesis benchmarks PSB1 [11] and PSB2 [8] that were not used by any popular evaluation metrics. These benchmarks are good as they have a extensive test case coverage with corner cases too! We can understand the performance of LLMs better by evaluating on these benchmarks.

REFERENCES

[1] [n. d.]. https://github.com/sosy-lab/cpachecker
[2] [n. d.]. https://github.com/SaswatPadhi/LoopInvGen/tree/master/benchmarks
[3] [n. d.]. https://github.com/SaswatPadhi/LoopInvGen/
[4] [n. d.]. https://github.com/openai/human-eval/
[5] [n. d.]. https://github.com/marcoeilers/nagini

[6] [n. d.]. https://github.com/amazon-science/mxeval
[7] [n. d.]. https://www.cprover.org/jbmc/
[8] [n. d.]. https://cs.hamilton.edu/~thelmuth/PSB2/PSB2.html
[9] Saikat Chakraborty, Shuvendu K. Lahiri, Sarah Fakhoury, Madanlal Musuvathi, Akash Lal, Aseem Rastogi, Aditya Senthilnathan, Rahul Sharma, and Nikhil Swamy. 2023. Ranking LLM-Generated Loop Invariants for Program Verification. arXiv:2310.09342 [cs.PL]
[10] Madeline Endres, Sarah Fakhoury, Saikat Chakraborty, and Shuvendu K. Lahiri. 2023. Formalizing Natural Language Intent into Program Specifications via Large Language Models. arXiv:2310.01831 [cs.SE]
[11] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. arXiv:2305.01210 [cs.SE]