Can LLMs Enable Verification in Mainstream Programming?

Aleksandr Shefer$^{1,2[0000-1111-2222-3333]}$, Igor Engel$^{1,2[1111-2222-3333-4444]}$, Stanislav Alekseev$^{1,3[2222--3333-4444-5555]}$, Daniil Berezun$^{1[3333-4444-5555-6666]}$, and Ekaterina Verbitskaia$^{1,2[0000-0002-6828-3698]}$

[1] JetBrains Research, Amsterdam, the Netherlands
[2] Constructor University, Bremen, Germany
[3] Neapolis University, Pafos, Cyprus

Abstract. Despite being able to produce reliable software, formal methods have hardly been adopted in mainstream programming. With the advent of large language models, it becomes more feasible to automatically generate code along with verification guarantees. This research explores whether LLMs can produce verified code from a textual description and a partial specification. We were able to achieve 63% success rate in Nagini and this in Viper on the HumanEval benchmark.

Software is notoriously difficult to get right. Off-by-one errors, null dereferencing, and infinite loops are among the most common avoidable mistakes developers tend to make. Formal methods aim to prevent these and more complicated errors by providing a programmer with means to reason about a program and prove its correctness. Such tools are especially valued in critical domains such as healthcare and finance, where software failures can have severe consequences. However, adopting formal verification requires significant additional effort and expertise, which limits their use beyond high-stakes applications.

SMT-powered software verification systems such as Dafny and F* partially automate proof search but remain standalone tools. This implies that to introduce verification into an existing project, one needs to make a tough decision of adopting a new language, which often comes with worse developer tools and a higher entrance barrier for the engineers. One way to overcome this drawback is to use an intermediate verification language such as Viper[4], with frontends in mainstream languages. The last hurdle to clear is to make it easy for developers to specify properties of their programs, as well as to prove that they hold.

Prior research!!! In this project, we explore whether large language models are capable of generating verified code in mainstream languages from a text description and a partial specification. We focus on Nagini[5] and Verus[6] – the extensions of the popular programming languages Python and Rust.

[4] Viper system: `https://www.pm.inf.ethz.ch/research/viper.html`

[5] Nagini, an automatic verifier for Python programs: `https://github.com/marcoeilers/nagini`

[6] Verus, a tool for verifying code correctness in Rust: `https://github.com/verus-lang/verus`

In addition to a function signature and a body, verified code contains a specification of its behavior. It includes preconditions that describe assumptions held before the evaluation of the function begins and guarantees ensured after execution, called postconditions. Sometimes these are enough to establish correctness, bun in the majority of non-trivial cases, additional statements should be proven, such as loop invariants or lemmas.

This provides a spectrum of data which can be exposed to a model in code synthesis tasks ranging from a textual description of the problem at hands to everything but loop invariants. In this extended abstract, we only focus on a scenario where the user describes the problem in a natural language and supplies a function signature with pre- and postconditions, as we view it as the most precise way to express the user intent. An example of a query is "Checks if given string is a palindrome" along with the following code.

```
def is_palindrome(text : List[int]) -> bool:
  Requires(Acc(list_pred(text))) # precondition
  Ensures(Acc(list_pred(text)))  # postconditions
  Ensures(Result() == Forall(int, lambda i:
    (i >= 0 and i < len(text))
    implies (text[i] == text[len(text) - i - 1])))
```

The model is them prompted to generate the function body as well as any necessary additional conditions necessary to finish the proof. We use few-shot The complete prompt can be found in the Appendix(ref). If the produced code verifies, it is accepted and passed to the user. Otherwise, the verifier feedback is sent to the model for further revision of the suggestion; this process is repeated up to 5 times.

In order to evaluate the abilities of the model, we created a benchmark based on HumanEval[1]. We manually implemented a subset of the problems in Nagini[7] and contributed in a collaborative effort to create it for Verus[8]. Not every problem in the original dataset suits well for verification. For some of them, specification duplicates the implementation (numbers), while unsupported language features are needed for others (numbers). In total, our benchmarks contain 106 problems for Nagini and this many for Verus.

We ran the described experiment on Claude-3.5-Sonnet, which successfully produced verified code for 67 problems (63%) in Nagini and this many in Verus. We conclude that using large language models can enable verification in mainstream programming languages.

References

1. Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brock-

[7] HumanEval dataset in Nagini: https://github.com/JetBrains-Research/HumanEval-Nagini/

[8] HumanEval dataset in Verus: https://github.com/secure-foundations/human-eval-verus

man, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.