Are LLMs Ready for Software Verification?

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 40% in Verus on the HumanEval benchmark.

Software is notoriously difficult to get right, and the cost of errors can be astronomical [8]. Formal methods aim to prevent avoidable mistakes 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 cryptography [17], finance [2], and aerospace [14], 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 require using specialized languages for both programs and verification primitives. 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 [11], 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.

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, but in the majority of non-trivial cases, additional statements should be proven, such as loop invariants or lemmas.

Most prior research has explored generation of either complete proofs [16,12,4] or their parts, such as invariants and assertions [10,13,7], for existing implementations. Other works produce postconditions [6] from textual descriptions, as well as complete verified code in Dafny [15] and in Rust [1]. This approach has a possibility of misinterpreting the user intent and thus producing code which works incorrectly despite being formally verified. In this project, we explore whether large language models (LLMs) alone are capable of generating verified code in mainstream languages from a text description and pre- and postconditions provided by a programmer. Specifically, we focus on Nagini [5] and Verus [9] — the verifiers of subsets of Python and Rust.

In order to evaluate the abilities of the model, we created a benchmark based on HumanEval [3]. We manually implemented a subset of the problems in Nagini[1] and contributed in a collaborative effort to create it for Verus[2]. Notice that our benchmarks contain fewer problems than the original dataset in Python, since not every problem in it is well-suited for verification. For some of them, specification duplicates the implementation (e.g. task 67), while unsupported language features are needed for others (e.g. tasks 2 and 4). In total, our benchmarks include 106 handwritten, verified programs in Nagini and 55 in Verus, along with accompanying textual descriptions.

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 a 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:
    Implies(i >= 0 and i < len(text),    # ==>
            text[i] == text[len(text) - i - 1]))
```

The model is then prompted to generate the function body as well as any necessary additional conditions necessary to finish the proof. The prompt includes explanations of some aspects of Nagini and Verus which LLMs tend to struggle with and employ the few-shot strategy, showcasing an example of successfully verified code. 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 iterated over up to five times.

We ran the described experiment on Claude Sonnet 3.5 five times and computed the number of unique problems, verified in at least one run. The model successfully produced verified code for 67 problems (63%) in Nagini and 22 programs (40%) in Verus. Our results demonstrate that LLMs have the potential to lower the barrier to formal verification by automating the generation of correct-by-construction code. It is worth mentioning that state-of-the-art code generation tools are considering more complicated benchmarks, such as SWE-bench. As a community, we should strive to develop similar datasets in verified settings, which is an enormous task on its own, given the effort poured into HumanEval.

References

1. Pranjal Aggarwal, Bryan Parno, and Sean Welleck. Alphaverus: Bootstrapping formally verified code generation through self-improving translation and treefinement.

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

[2] HumanEval-Verus: `https://github.com/secure-foundations/human-eval-verus`

arXiv preprint arXiv:2412.06176, 2024.
2. Franck Cassez, Joanne Fuller, and Aditya Asgaonkar. Formal verification of the ethereum 2.0 beacon chain. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 167–182. Springer, 2022.
3. Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.
4. Tianyu Chen, Shuai Lu, Shan Lu, Yeyun Gong, Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Hao Yu, Nan Duan, Peng Cheng, et al. Automated proof generation for rust code via self-evolution. *arXiv preprint arXiv:2410.15756*, 2024.
5. Marco Eilers and Peter Müller. Nagini: a static verifier for python. In *Computer Aided Verification: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I 30*, pages 596–603. Springer, 2018.
6. Madeline Endres, Sarah Fakhoury, Saikat Chakraborty, and Shuvendu K Lahiri. Can large language models transform natural language intent into formal method postconditions? *Proceedings of the ACM on Software Engineering*, 1(FSE):1889–1912, 2024.
7. Adharsh Kamath, Aditya Senthilnathan, Saikat Chakraborty, Pantazis Deligiannis, Shuvendu K Lahiri, Akash Lal, Aseem Rastogi, Subhajit Roy, and Rahul Sharma. Finding inductive loop invariants using large language models. *arXiv preprint arXiv:2311.07948*, 2023.
8. Herb Krasner. The cost of poor software quality in the us: A 2022 report. *Proc. Consortium Inf. Softw. QualityTM (CISQTM)*, 2022.
9. Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying rust programs using linear ghost types. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):286–315, 2023.
10. Eric Mugnier, Emmanuel Anaya Gonzalez, Ranjit Jhala, Nadia Polikarpova, and Yuanyuan Zhou. Laurel: Generating dafny assertions using large language models. *arXiv preprint arXiv:2405.16792*, 2024.
11. Peter Müller, Malte Schwerhoff, and Alexander J Summers. Viper: A verification infrastructure for permission-based reasoning. In *Verification, Model Checking, and Abstract Interpretation: 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings 17*, pages 41–62. Springer, 2016.
12. Gabriel Poesia, Chloe Loughridge, and Nada Amin. dafny-annotator: Ai-assisted verification of dafny programs. *arXiv preprint arXiv:2411.15143*, 2024.
13. Álvaro F Silva, Alexandra Mendes, and João F Ferreira. Leveraging large language models to boost dafny's developers productivity. In *Proceedings of the 2024 IEEE/ACM 12th International Conference on Formal Methods in Software Engineering (FormaliSE)*, pages 138–142, 2024.
14. Jean Souyris. Industrial use of compcert on a safety-critical software product. *https://projects.laas.fr/IFSE/FMF/J3/slides/P05_Jean_Souyiris.pdf*, 2014.
15. Chuyue Sun, Ying Sheng, Oded Padon, and Clark Barrett. Clover: Closed-loop verifiable code generation. In *International Symposium on AI Verification*, pages 134–155. Springer, 2024.

16. Jianan Yao, Ziqiao Zhou, Weiteng Chen, and Weidong Cui. Leveraging large language models for automated proof synthesis in rust. *arXiv preprint arXiv:2311.03739*, 2023.
17. Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. Hacl*: A verified modern cryptographic library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1789–1806, 2017.

Appendix

In this section, we provide prompts used to assess the ability of an LLM to generate verified code in Nagini. The prompts used for Verus have similar structure. The system prompt is the following.

System prompt. *You are an expert in a Python verification framework, Nagini. You will be given tasks dealing with Python programs, including precise annotations. Do not provide ANY explanations. Don't include markdown backticks. Respond only in Python code, nothing else. You respond only with code blocks.*

The main prompt instructs the model on what its task is, as well as providing additional reminders about the syntax of the language and an example of a complete proof.

Main prompt. *Rewrite the following Nagini code with implementations of some functions missing. While rewriting it, ensure that it verifies. Include invariants and assertions. Don't remove any helper functions (they are marked with @Pure annotation), they are there to help you. Prefer loops to recursion. Use helper functions only in invariants, asserts and conditions (in 'if' or 'while' conditions). Don't use helpers in the plain code. Do not change helper functions. Add code and invariants to other functions. Ensure that the invariants are as comprehensive as they can be. Even if you think some invariant is not totally necessary, better add it than not. Don't add any additional text comments, your response must contain only program with invariants. Do not provide ANY explanations. Don't include markdown backticks. Respond only in Python code, nothing else.*

You remember the following aspects of Nagini syntax:

1. Nagini DOES NOT SUPPORT some Python features as list comprehensions (`k + 1 for k in range(5)`*), as double inequalities (*`a <= b <= c`*). Instead of double inequalities it's customary to use two separate inequalities (*`a <= b`* and* `b <= c`*).*

< 5 more aspects mentioned >

You might need to work with accumulating functions, such as sum, so here's an example of how to do that:

< Example program >

To help you, here's a text description given to a person who wrote this program:

< Text description >

The program:

< Function specification with preconditions and postconditions >

If the verifier failed to check the code provided, its feedback is sent back to the model with the following prompt.

Error prompt. *The following errors occurred during verification:*
< The list of errors >
Please fix the errors by adding, removing or modifying the implementation, invariants or assertions and return the fixed program. You should not modify any helper functions (annotated with @Pure), remember they are here to help you with verification. Don't add any additional text comments, your response must contain only program with invariants. Do not provide ANY explanations. Don't include markdown backticks. Respond only in Python code, nothing else.