Evaluating the Ability of GPT-4o to Generate Verifiable Specifications in VeriFast

Wen Fan
Purdue University
West Lafayette, IN, USA
fan372@purdue.edu

Marilyn Rego
Purdue University
West Lafayette, IN, USA
mrego@purdue.edu

Xin Hu
University of Michigan - Ann Arbor
Ann Arbor, MI, USA
hsinhu@umich.edu

Sanya Dod
Purdue University
West Lafayette, IN, USA
sdod@purdue.edu

Zhaorui Ni
Purdue University
West Lafayette, IN, USA
ni134@purdue.edu

Danning Xie
Purdue University
West Lafayette, IN, USA
xie342@purdue.edu

Jenna DiVincenzo
Purdue University
West Lafayette, IN, USA
jennad@purdue.edu

Lin Tan
Purdue University
West Lafayette, IN, USA
lintan@purdue.edu

Abstract—Static verification is a powerful method for enhancing software quality, but it demands significant human labor and resources. This is particularly true of static verifiers that reason about heap manipulating programs using an ownership logic. LLMs have shown promise in a number of software engineering activities, including code generation, test generation, proof generation for theorem provers, and specification generation for static verifiers. However, prior work has not explored how well LLMs can perform specification generation for specifications based in an ownership logic, such as separation logic. To address this gap, this paper explores OpenAI's GPT-4o model's effectiveness in generating specifications on C programs that are verifiable with VeriFast, a separation logic based static verifier. Our experiment employs three different types of user inputs as well as basic and Chain-of-Thought (CoT) prompting to assess GPT's capabilities. Our results indicate that the specifications generated by GPT-4o preserve functional behavior, but struggle to be verifiable. When the specifications are verifiable they contain redundancies. Future directions are discussed to improve the performance.

Index Terms—formal verification, large language models, prompt engineering, separation logic

I. INTRODUCTION

Auto-active (Hoare-logic styled [1], static) verifiers, such as Viper [2], Verus [3], Dafny [4], Gillian [5], and VeriFast [6], are powerful as they can prove the absence of large classes of bugs in code. Ideally, users of such tools need only specify the intended behavior of their code on the code itself (as pre- and postconditions), and the tool will automatically provide feedback on whether or not the code is provably correct with respect to this behavior. In reality, auto-active verifiers require many more auxiliary specifications (e.g. loop invariants, lemmas, opens, closes) to achieve this goal, burdening users.

In recent years, large language models (LLMs) have been effective in generating code [7], [8], test-cases [9]–[14], and proofs in proof assistants [15]–[19]. LLMs have also been shown to be effective at generating specifications supported by auto-active verifiers [20]–[25]. However, related work has not explored whether or not off-the-shelf LLMs can generate specifications based on a permissions logic, like *separation logic* [26], that can be verified by auto-active verifiers such

as VeriFast, Gillian, and Viper. Thanks to such specifications, these verifiers do well at verifying programs that manipulate the heap for both memory safety and functional properties. But, permissions logic based specifications are particularly cumbersome to write, because they must specify the shape of the heap alongside functional constraints. This leads to specifications containing a number of predicates that hide heap details; and as a result, numerous lemmas, folds, unfolds, and special loop invariants that are used to connect the content of these predicates. While such specifications are difficult to reason about, they are written in a patterned way that may be amenable to generation via LLMs.

Therefore, this paper evaluates how effective OpenAI's GPT-4o model [27] is at generating specifications that can be verified by VeriFast [6], which supports separation logic-based verification of C and Java code. We selected OpenAI's GPT-4o model due to its strong text comprehension and generation capabilities [28]. We ask three main research questions regarding whether or not the LLM's output specifications and code preserve functional behavior, are verified, and are conventional. To answer these questions, we developed 21 input-output pairs from 150 publicly available VeriFast programs as test cases and ground truth, and employed two prompt engineering methods (Basic Prompting and CoT Prompting) to instruct GPT-4o. Then, we manually inspected the outputs generated by the LLM by comparing them with the ground truth in a qualitative analysis. Results show that GPT-4o generates specifications and code that preserves functional behavior expressed in input files. We also found that both prompting methods result in tons of verification errors, and the small number of LLM output files that do verify contain redundant specifications. This shows a need for better prompting techniques, which we intend to explore in follow-up work.

II. RELATED WORK

A. Specification Generation for Auto-active Verifiers

Houdini [29] is an annotation assistant that suggests candidate annotations using heuristics, which are then verified

or refuted by ESC/Java [30]. Vogels et al. [31] showed on eight small programs that given preconditions shape analysis can deduce all specifications required to verify programs with VeriFast. Yao et al. [32] combine GPT-4 and static analysis to automate proof synthesis in Verus [3]. Kamath et al. [21] use LLMs to generate inductive loop invariants and then verify the candidates with Frama-C [33]. Misu et al. [22] investigated how well GPT-4 and PaLM-2 can synthesize verified methods in Dafny with various prompting approaches, and found few-shot reasoning is the best. Laurel [34] uses large language models with prompting techniques to generate helper assertions for Dafny programs. Finally, Mukherjee et al. [25], which is most closely related, present a synthesis framework for C code that uses LLMs to generate programs verified by VeriFast. Our work focuses on verifying existing C code with VeriFast rather than generating whole programs. Further, our work considers loops and helper functions; theirs does not.

B. Program Invariant Generation with LLMs or ML

Daikon [35] automatically infers likely invariants from program executions using machine learning for use in testing, debugging, and verification tasks. Pei et al. [36] used fine-tuned large language models to predict program invariants on par with Daikon. Xie et al. [37] evaluated the capabilities of LLMs to generate specifications from comments and documentation, demonstrating few-shot learning and advanced prompt strategies outperform traditional methods by 5.1–10.0%. Only Nimmer et. al [38] tried to integrate generated program invariants with an auto-active verifier pairing Daikon with ESC/Java to achieve over 90% precision and recall. Daikon does not rely on LLMs to generate specifications and ESC/Java does not support separation logic, so our work is novel in comparison.

C. Proof Synthesis for Proof Assistants with LLMs or ML

Lots of work utilize LLMs or neural networks to generate tactics (next proof steps) or whole proofs in proof assistants (interactive theorem provers) such as Coq [39], Isabelle/HOL [40], and Lean [41]. For Coq, Sanchez-Stern et al. [42] improved on existing neural network based proof synthesis tools ASTactic [43] and TacTok [44] by modeling identifiers. Lu et. al [45] combined LLMs with symbolic methods to synthesize whole proofs. For Isabelle/HOL, Jiang et. al. [17] use a fine-tuned language model to synthesize proof steps until a proof is achieved. Thor [46] uses hammers for premise selection and designates the rest of proof synthesis to language models. Similarly, Jiang et al. [47] leverage informal and formal proof sketches that are filled in by LLMs. Finally, First et. al [19] fine-tune LLMs for whole-proof generation and repair. For Lean, Han et al. [48] use LLMs to suggest proof tactics. In contrast, our work focuses on the capabilities of LLMs to generate specifications for auto-active verifiers.

III. VERIFAST & BENCHMARKS

A. VeriFast

VeriFast [6] is a sound and modular, auto-active verifier that reasons with symbolic execution [49] to efficiently verify single and multi-threaded C and Java programs against specifications in separation logic [26]. As a result, VeriFast can verify functional properties and memory safety (e.g., dangling pointer dereference and double free). We chose VeriFast due to its maturity: e.g. verifying four industrial case studies for memory safety [50]. Furthermore, over 150 open-source C programs have been specified and verified with VeriFast.

B. VeriFast Benchmarks used in the Evaluation

We selected a subset of 21 fully specified C files available publicly in VeriFast's Github repository to use in our evaluation. These benchmarks cover diverse programming concepts and verification properties. For example, *filter_stack.c* and *values.c* involve on-heap data structures like stacks and linked lists, while others have file I/O (e.g., *cp.c*), fractional permission (e.g., *fractions-counting.c*) and recursion (e.g., *wc.c*). We extracted the types of specification constructs used in each file (e.g. separation logic arrow and predicate instance) and where they are specified (e.g., in precondition or postcondition). Table I presents the aggregated results of files from this analysis. Of the 398 specification elements observed, predicate calls (44.2%), boolean expressions (32.7%) and separation logic arrows (10.3%) are prevalent. Moreover, the heap-related specifications (e.g., separation logic arrow and fractional permission) occurred the most in preconditons, postconditions and predicate declarations. We also find that the number of specifications varied a lot (e.g. *filter_stack.c* has 49 lines while *typedef.c* has 2 lines).

C. Input-Output Pairs

We developed input-output pairs from the benchmarks described in §III-B. The inputs are GPT-4o's starting point and the outputs are GPT-4o's goal. Thus, the output files are defined as the fully verified benchmarks. We experiment with three types of inputs per distinct output file that represent possible user inputs: Natural Language (NL), Functional Behavior (FB), and Functional Behavior Plus (FB+). Each of the input types contain code from the output file but with different partial specifications. An NL input does not contain any formal specifications, but only a natural language description of the intended behavior of each function. A FB input contains formal specifications that specify only the functional behavior of each function (e.g., pre- and postconditions). Finally, a FB+ input contains only formally specified preconditions and postconditions directly from the output file, which specify functional behavior but may also contain other properties. Examples of each input type and the corresponding output for them is shown in Listing 1.

```
// The increment function increments the
// value of the Counter structure by one ...
void increment(struct Counter* c) // NL input
{ int tmp = c->value; c->value = tmp + 1; }

void increment(struct Counter* c) // FB input
//@ requires Counter(c, ?v);
//@ ensures Counter(c, v+1);
{ int tmp = c->value; c->value = tmp + 1; }
```

Specification	Pre.	Post.	Open	Close	Pred. Declar.	Lemma Declar.
Aggregated Result	8/3/59/35/6/3	9/2/47/35/5/5	0/6/0/33/0/0	0/3/0/38/0/0	24/5/15/5/2/10	0/0/9/0/1/0

TABLE I: Aggregated results of specification constructs. Element counts are formatted as "Separation Logic Arrow/Fractional Permissions/Boolean Expression/Predicate Call/Empty Heap/Malloc Block."

```
void increment(struct Counter* c) // FB+ input
//@ requires Counter(c, ?v) &*& v < INT_MAX;
//@ ensures Counter(c, v+1);
{ int tmp = c->value; c->value = tmp + 1; }

void increment(struct Counter* c) // output
//@ requires Counter(c, ?v) &*& v < INT_MAX;
//@ ensures Counter(c, v+1);
{ //@ open Counter(c, v);
  int tmp = c->value;
  c->value = tmp + 1;
  //@ close Counter(c, v+1); }
```

Listing 1: Input-output pairs created for the `increment` function

IV. PROMPT ENGINEERING

We explore two different prompting approaches in our study: Basic and Chain of Thought (CoT) prompting.

Basic Prompting: The basic prompting approach instructs GPT-4o to generate verifiable specifications with VeriFast for an input with a single, few sentence prompt. This prompt provides minimal context and instructions and is available (*link available post review*). Basic prompting establishes a baseline to evaluate CoT and future prompting approaches against.

CoT Prompting: The CoT prompting approach [51] guides an LLM through structured, step-by-step instructions for generating VeriFast specifications. Our version of CoT mirrors the logical progression followed by experts in specification writing. For example, when writing a precondition, the prompt first explains how to capture the input behavior of a function, then details constraints on syntax and positional requirements, and finally addresses additional properties such as memory safety. The CoT prompt, with code, is available (*Link included post review*). Unlike few-shot prompting used by Misu et al. [22], our CoT prompting deconstructs the specification writing process into actionable steps.

V. QUALITATIVE ANALYSIS

We prompt GPT-4o for output files using scripts implementing the prompting approaches (§IV) and our input files (§III-C). We assess the GPT-4o outputs using a qualitative analysis designed to answer the following research questions:

- **RQ1:** How well does GPT-4o preserve functional behavior?
- **RQ2:** How well does GPT-4o generate verifiable specifications?
- **RQ3:** How conventional are the verifiable specifications generated by GPT-4o?

For each RQ, we developed an initial set of qualitative codes from our intuition, comparisons with the ground truth, and a pilot study on a subset of the output files. As we performed our qualitative analysis by applying the codes to parts of the output files as necessary, we refined our codes as well.

For **RQ1**, we assessed how well GPT-4o preserves functional behavior expressed in function contracts (precondition-/postconditions) and in source code. We assigned codes/subcodes categorized as *preserved* when the output is equivalent to the input, *strengthened* when the output implies the input, *weakened* when the output is implied by the input, and *others* (e.g. when the output is unrelated to the input).

For **RQ2**, we ran VeriFast on all outputs from GPT-4o to check their verifiability. If a file was verified, then we analyzed its specifications for how conventional they are (**RQ3**). Otherwise, we assigned codes representing the cause of verification failure(s). To capture all the underlying verifiability issues in the output, we iteratively fixed failures until the output verifies. The codes fall into two categories: *compilation error* or *verification error*. Compilation error codes include *specification out-of-position (spec-OOP)*, *syntax errors* (errors during the parsing stage) and *include & type check errors* (errors during the include or type checking stage). Verification errors are assigned based on the the component of the corresponding fix and include incorrect *precondition/postcondition*, *predicate definition*, *open/close/assert/leak* use or definition, *lemma definition*, *lemma use*, and *loop invariant* and *other* errors (e.g., source code being modified incorrectly).

For **RQ3**, we analyze the conventionality of correctly generated specifications. Our codes for this research question capture redundant specifications that are unnecessary for verification and ambiguous specifications that exhibit naming mismatches with the intended function. The assigned codes are aggregated by occurrence; and, the results given next (§VI).

VI. RESULT & DISCUSSION

A. *RQ1 - Functional Behavior Analysis*

Table II summarizes functional behavior in output files, where most of them preserve the functional behavior. Specifically, in the 126 output files analyzed (21 benchmarks $\times$ 3 input types $\times$ 2 prompting), only 20 show changes in preconditions/postconditions and 3 show changes in source code. For the 7 files with strengthened functional behavior, 6 had extra constraints in postcondition (e.g., "`new_count <= count`"). For 10 files with weakened functional behavior, 9 missed properties in the postconditions or predicates (e.g., not specifying the value of balance in "`acc->balance |-> ?b`"). The modified functional behavior in the source code was due to LLM altering the source code (e.g., changing the argument from -100 to 100). Thus, to answer **RQ1**, we show GPT-4o largely preserves functional behavior in specifications and source code across all input types and prompting techniques.

Functional Behavior	Pre/Postcondition	Source Code
preserved	106	123
strengthened	7	0
weakened	10	1
others	3	2
Total	126	126

TABLE II: Functional Behavior Preservation Analysis

B. RQ2 - VeriFast Specifications Analysis

Table III shows the aggregated number of different errors in GPT-4o's outputs across the 21 benchmarks with different prompting methods and input types. Only 9 out of the 126 output files were directly verifiable, while the remaining files showcased a high number of errors, with 539 errors from basic prompting and 555 from the CoT prompting approach.

1) Impact of Input Type: The NL inputs resulted in more compilation errors (103 and 123) than other inputs (at most 28). Similarly, considering the errors of pre/postcondition and predicate, FB+ inputs resulted in fewer such errors (at most 6 + 2 = 8 in CoT prompt of FB+), compared to other types (at least 15 + 9 = 24). This difference can be attributed to more detailed and precise specifications provided in FB+ inputs. Thus, providing GPT-4o with examples or input containing detailed preconditions and postconditions in VeriFast syntax can improve the correctness of its output. However, even if correct preconditions/postconditions are provided, GPT-4o occasionally modifies them, introducing errors (e.g., removing bound checks in a precondition). Additionally, the three input types produced a similar number of errors for lemmas and loop invariants. For lemmas, this may be because they often require extra information beyond the capabilities of the GPT-4o. For instance, 114 errors were about implicit bound checks for data types like `size_t`, and 27 errors were linked to meeting the preconditions of standard library functions such as `fread`. For loop invariants, the difficulty likely arises from the complexity of ensuring their correctness, as they must hold true both at the start of the loop and after each iteration.

2) Impact of Prompting Method: The basic prompting and CoT prompting resulted in similar performance. On NL, FB, and FB+ inputs, the CoT prompt did not reduce the number of compilation errors compared to basic prompting. In fact, for NL inputs, the CoT prompting resulted in more *spec-OOP* errors (72 compared to 54). This is surprising because the CoT prompt explicitly told GPT-4o to put preconditions and postconditions at the right position. Similarly, CoT is not significantly better than basic prompting on verification errors, except errors about open/close/assert/leak statements on FB+ input (47 compared to 77). This shows that in almost all cases adding more instructions steps in a prompt does not improve GPT-4o's performance. However, when preconditions, postconditions, and their predicate dependencies are specified as needed for verification, CoT works well for generating dependent auxiliary specifications. Thus, to answer **RQ2**, GPT-4o generates VeriFast specifications with limited success across the input types and prompting strategies

Error Code	Basic Prompt			CoT Prompt		
Sub-code	NL	FB	FB+	NL	FB	FB+
Compilation Error						
spec-OOP	54	1	0	72	8	0
syntax	16	10	5	12	7	9
include & type check	33	8	9	39	13	10
Total	**103**	**19**	**14**	**123**	**28**	**19**
Verification Error						
pre/postcondition	28	18	2	25	15	6
predicate definition	10	12	3	14	9	2
open/close/assert/leak	49	65	77	57	65	47
lemma definition	1	2	2	2	2	5
lemma use	24	28	28	27	27	27
loop invariant	9	10	9	9	10	12
others	8	7	11	9	8	7
Total	**129**	**142**	**132**	**143**	**136**	**106**

TABLE III: Error Code Analysis for Basic and CoT Prompts

assessed. Improvements to the prompting approaches or new prompting approaches should be explored.

C. RQ3 - Convention Analysis

In the verifiable output, there were 10 cases of redundant specifications that, while verifiable, could be simplified. This includes 4 specifications with unnecessary encapsulation of a predicate or lemma, 5 with redundant conditions, and 1 with an unused predicate. For example, in the `person` predicate definition in Listing 2, removing `p != 0` doesn't affect correctness since other clauses imply it.

```
predicate person(struct person *p, ...) =
    p != 0 &*& malloc_block_person(p) &*& ...
```

Listing 2: Predicate with a redundant condition

Thus, to answer **RQ3**, GPT-4o-generated verifiable specifications occasionally include redundant elements, suggesting the need to make GPT-4o aware of conventions.

D. Discussion and Suggestions for Future Work

Future work can resolve errors by a more fine-grained prompt engineering method, such as refining prompt or input granularity (e.g., granularity of function as in [22] [32] [21]), or providing verifier's feedback to LLM to guide the fix (e.g., [21]). Further, the performance differences among input types show that providing example specifications to an LLM as part of the prompting process may improve performance. Few-shot prompting showed promise in related work [22] [24], and we plan to evaluate this approach in future work. Fine-tuning (e.g., [52]) may also be an interesting avenue for exploration.

Threats to Validity: The results of our analysis is limited with only 21 benchmarks, 1 verifier, and 1 LLM, but provides useful results to build on. The benchmarks were open-sourced years ago, so GPT-4o may have been trained on them biasing our results; however, GPT-4o's poor performance signals that

this bias is not impactful. Finally, while the analysis is methodical, it remains subjective; efforts to mitigate bias include thorough review, discussion, and consensus among conductors.

VII. CONCLUSION

This work is the first attempt to evaluate LLMs' ability to generate separation logic specifications for verification with an auto-active verifier. We develop 21 benchmarks and design three types of user input and two prompting methods to prompt GPT-4o with. The results show that while the output generated by GPT-4o preserves functional behavior in the input, it suffers from significant errors that block automated verification. Despite more negative results, our work provides guidance for the development of more effective prompt engineering approaches.

REFERENCES

[1] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[2] P. Müller, M. Schwerhoff, and A. 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*. Springer, 2016, pp. 41–62.

[3] A. Lattuada, T. Hance, C. Cho, M. Brun, I. Subasinghe, Y. Zhou, J. Howell, B. Parno, and C. Hawblitzel, "Verus: Verifying rust programs using linear ghost types," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA1, pp. 286–315, 2023.

[4] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *International conference on logic for programming artificial intelligence and reasoning*. Springer, 2010, pp. 348–370.

[5] J. Fragoso Santos, P. Maksimović, S.-É. Ayoun, and P. Gardner, "Gillian, part i: a multi-language platform for symbolic execution," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 927–942.

[6] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens, "Verifast: A powerful, sound, predictable, fast verifier for c and java," in *NASA formal methods symposium*. Springer, 2011, pp. 41–55.

[7] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J.-G. Lou, and W. Chen, "Codet: Code generation with generated tests," *arXiv preprint arXiv:2207.10397*, 2022.

[8] S. Sarsa, P. Denny, A. Hellas, and J. Leinonen, "Automatic generation of programming exercises and code explanations using large language models," in *Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 1*, 2022, pp. 27–43.

[9] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models," in *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*, 2023, pp. 423–435.

[10] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, "Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 919–931.

[11] N. Rao, K. Jain, U. Alon, C. Le Goues, and V. J. Hellendoorn, "Catlm training language models on aligned code and tests," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 409–420.

[12] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," *IEEE Transactions on Software Engineering*, 2023.

[13] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language models: Survey, landscape, and vision," *IEEE Transactions on Software Engineering*, 2024.

[14] C. S. Xia, M. Paltenghi, J. Le Tian, M. Pradel, and L. Zhang, "Fuzz4all: Universal fuzzing with large language models," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[15] C. Zheng, H. Wang, E. Xie, Z. Liu, J. Sun, H. Xin, J. Shen, Z. Li, and Y. Li, "Lyra: Orchestrating dual correction in automated theorem proving," *arXiv preprint arXiv:2309.15806*, 2023.

[16] K. Yang, A. Swope, A. Gu, R. Chalamala, P. Song, S. Yu, S. Godil, R. J. Prenger, and A. Anandkumar, "Leandojo: Theorem proving with retrieval-augmented language models," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[17] A. Q. Jiang, W. Li, J. M. Han, and Y. Wu, "Lisa: Language models of isabelle proofs," in *6th Conference on Artificial Intelligence and Theorem Proving*, 2021, pp. 378–392.

[18] S. Welleck and R. Saha, "Llmstep: Llm proofstep suggestions in lean," *arXiv preprint arXiv:2310.18457*, 2023.

[19] E. First, M. N. Rabe, T. Ringer, and Y. Brun, "Baldur: Whole-proof generation and repair with large language models," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1229–1241.

[20] L. Ma, S. Liu, Y. Li, X. Xie, and L. Bu, "Specgen: Automated generation of formal program specifications via large language models," *arXiv preprint arXiv:2401.08807*, 2024.

[21] A. Kamath, A. Senthilnathan, S. Chakraborty, P. Deligiannis, S. K. Lahiri, A. Lal, A. Rastogi, S. Roy, and R. Sharma, "Finding inductive loop invariants using large language models," *arXiv preprint arXiv:2311.07948*, 2023.

[22] M. R. H. Misu, C. V. Lopes, I. Ma, and J. Noble, "Towards ai-assisted synthesis of verified dafny methods," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 812–835, 2024.

[23] F. He, J. Zhai, and M. Pan, "Beyond code generation: Assessing code llm maturity with postconditions," *arXiv preprint arXiv:2407.14118*, 2024.

[24] E. Mugnier, E. A. Gonzalez, R. Jhala, N. Polikarpova, and Y. Zhou, "Laurel: Generating dafny assertions using large language models," 2024. [Online]. Available: https://arxiv.org/abs/2405.16792

[25] P. Mukherjee and B. Delaware, "Towards automated verification of llm-synthesized c programs," 2024. [Online]. Available: https://arxiv.org/abs/2410.14835

[26] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 2002, pp. 55–74.

[27] OpenAI, "Hello gpt-4o," 2024. [Online]. Available: https://openai.com/index/hello-gpt-4o/

[28] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[29] C. Flanagan and K. R. M. Leino, "Houdini, an annotation assistant for esc/java," in *International Symposium of Formal Methods Europe*. Springer, 2001, pp. 500–517.

[30] K. R. M. Leino, G. Nelson, and J. B. Saxe, "Esc/java user's manual," *ESC*, vol. 2000, p. 002, 2000.

[31] F. Vogels, B. Jacobs, F. Piessens, and J. Smans, "Annotation inference for separation logic based verifiers," in *International Conference on Formal Methods for Open Object-Based Distributed Systems*. Springer, 2011, pp. 319–333.

[32] J. Yao, Z. Zhou, W. Chen, and W. Cui, "Leveraging large language models for automated proof synthesis in rust," *arXiv preprint arXiv:2311.03739*, 2023.

[33] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c: A software analysis perspective," *Formal aspects of computing*, vol. 27, no. 3, pp. 573–609, 2015.

[34] E. Mugnier, E. A. Gonzalez, R. Jhala, N. Polikarpova, and Y. Zhou, "Laurel: Generating dafny assertions using large language models," *arXiv preprint arXiv:2405.16792*, 2024.

[35] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Science of computer programming*, vol. 69, no. 1-3, pp. 35–45, 2007.

[36] K. Pei, D. Bieber, K. Shi, C. Sutton, and P. Yin, "Can large language models reason about program invariants?" in *International Conference on Machine Learning*. PMLR, 2023, pp. 27 496–27 520.

[37] D. Xie, B. Yoo, N. Jiang, M. Kim, L. Tan, X. Zhang, and J. S. Lee, "Impact of large language models on generating software specifications," *arXiv preprint arXiv:2306.03324*, 2023.

[38] J. W. Nimmer and M. D. Ernst, "Automatic generation of program specifications," in *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 229–239. [Online]. Available: https://doi.org/10.1145/566172.566213

[39] B. Barras, S. Boutin, C. Cornes, J. Courant, J.-C. Filliatre, E. Gimenez, H. Herbelin, G. Huet, C. Munoz, C. Murthy *et al.*, "The coq proof assistant reference manual: Version 6.1," Ph.D. dissertation, Inria, 1997.

[40] T. Nipkow, M. Wenzel, and L. C. Paulson, *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002.

[41] L. De Moura, S. Kong, J. Avigad, F. Van Doorn, and J. von Raumer, "The lean theorem prover (system description)," in *Automated Deduction-CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings 25*. Springer, 2015, pp. 378–388.

[42] A. Sanchez-Stern, E. First, T. Zhou, Z. Kaufman, Y. Brun, and T. Ringer, "Passport: Improving automated formal verification using identifiers," *ACM Trans. Program. Lang. Syst.*, vol. 45, no. 2, 2023. [Online]. Available: https://doi.org/10.1145/3593374

[43] K. Yang and J. Deng, "Learning to prove theorems via interacting with proof assistants," in *International Conference on Machine Learning*. PMLR, 2019, pp. 6984–6994.

[44] E. First, Y. Brun, and A. Guha, "Tactok: semantics-aware proof synthesis," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, 2020. [Online]. Available: https://doi.org/10.1145/3428299

[45] M. Lu, B. Delaware, and T. Zhang, "Proof automation with large language models," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 1509–1520.

[46] A. Q. Jiang, W. Li, S. Tworkowski, K. Czechowski, T. Odrzygóźdź, P. Miłoś, Y. Wu, and M. Jamnik, "Thor: Wielding hammers to integrate language models and automated theorem provers," *Advances in Neural Information Processing Systems*, vol. 35, pp. 8360–8373, 2022.

[47] A. Q. Jiang, S. Welleck, J. P. Zhou, T. Lacroix, J. Liu, W. Li, M. Jamnik, G. Lample, and Y. Wu, "Draft, sketch, and prove: Guiding formal theorem provers with informal proofs," in *The Eleventh International Conference on Learning Representations*, 2023.

[48] J. M. Han, J. Rute, Y. Wu, E. Ayers, and S. Polu, "Proof artifact co-training for theorem proving with language models," in *International Conference on Learning Representations*, 2022.

[49] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[50] P. Philippaerts, J. T. Mühlberg, W. Penninckx, J. Smans, B. Jacobs, and F. Piessens, "Software verification with verifast: Industrial case studies," *Science of Computer Programming*, vol. 82, pp. 77–97, 2014.

[51] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

[52] T. Chen, S. Lu, S. Lu, Y. Gong, C. Yang, X. Li, M. R. H. Misu, H. Yu, N. Duan, P. Cheng *et al.*, "Automated proof generation for rust code via self-evolution," *arXiv preprint arXiv:2410.15756*, 2024.