AI for Program Verification

Cristian Cadar[a], **Abhik Roychoudhury**[b]

[a] *Imperial College London* c.cadar@imperial.ac.uk

[b] *National University of Singapore* abhik@comp.nus.edu.sg

1. Introduction

Our society runs on code, with all aspects of life affected by software. Scientific discovery is equally reliant on software, with all branches of science depending on software to collect and process data, build computational models, run simulations, and validate scientific hypotheses. Errors in such software can have a significant impact, ranging from retractions from scientific journals [1] to the public losing trust in scientific models (e.g., the climate skepticism campaign after the ClimateGate incident [2]) to flight disasters [3, 4].

The science of program verification has existed for almost as long as the field of computing [5], straddling both computer science and mathematics. Program verification involves proving facts about the program, in particular that the code correctly follows its *specification*. For program verification to be possible, a formal mathematical specification that describes the expected behaviour of the program must be provided.[1]

A common and well-known program verification approach is *interactive deductive verification*. In this approach, specification and code are used to create a set of mathematical *proof obligations*, which are then discharged to an *interactive theorem prover*. In recent years, several influential interactive theorem provers (also called *proof assistants*) have been created, such as Coq [6], Isabelle [7] and Vampire [8]. These systems can develop formal proofs via computer-human collaboration, where some steps are provided by the computer, and some by the human. Typically, the human guides the computer toward the proof whenever it cannot make automatic progress by itself.

Program verification has seen tremendous success in the last couple of decades, with several machine-verified software systems being created, such as the CompCert verified compiler [9] and the seL4 verified microkernel [10]. However, such verified software is extremely expensive to build, with manually-written specifications often significantly larger than the code itself, and with the overall verification process often requiring years of PhD-level expertise. For example, the seL4 developers estimate that it took over 200,000 lines of Isabelle proof script and 20 person-years of effort to build the initial version of seL4 [11].

[1] Some implicit specifications, such as the lack of memory errors, can be used, but the ambition of program verification is to prove full functional correctness.

2. AI for Program Verification

Recent developments in generative AI (GenAI) have opened the possibility to revolutionise the scalability of the program verification process. In this extended abstract, we discuss two promising interconnected directions which we have recently started to investigate: AI-driven specification inference and AI-driven proof search.

2.1 AI-driven Specification Inference

One of the most expensive and difficult stages in program verification is writing the formal specifications. Even if the system-level intended behavior is known and documented, program verification requires writing formal specifications for individual units of code. As discussed above, these specifications are often enormous in size: the initial version of the seL4 verified microkernel required 200,000 lines of Isabelle code [11], while the specification for the CompCert verified compiler was eight times larger than the code itself [12]. In addition to the huge amount of effort involved, the current approach to verified systems is based on building them from scratch, as inferring intent from an existing system would be impractical.

Our recent work on SpecRover [13] shows the power of *LLM agents* for intent extraction from existing codebases. LLM agents [14] exploit large language models (LLMs) but do so in an autonomous way, with access to a variety of external tools. We believe that a similar agentic approach as in SpecRover can be used for specification inference for code units in a complex software system for the purpose of program verification. Roughly speaking, this would amount to inferring the intended pre-conditions and post-conditions for every function or method in a large codebase.

Furthermore, GenAI makes it possible to take advantage of artifacts which have previously been largely inaccessible to automated program verification, such as documentation, developer discussions, and requests for comments. While imperfect, such artifacts are rich sources that capture program intent, which can be used to infer specifications.

Software systems need to be able to change, e.g., to add new features or adapt to new hardware platforms and environments. As a result, specifications needs to grow and evolve as well, to keep in sync with the software system. We believe that LLM agents could make it easier to evolve specifications over time, by adapting the old specifications to include

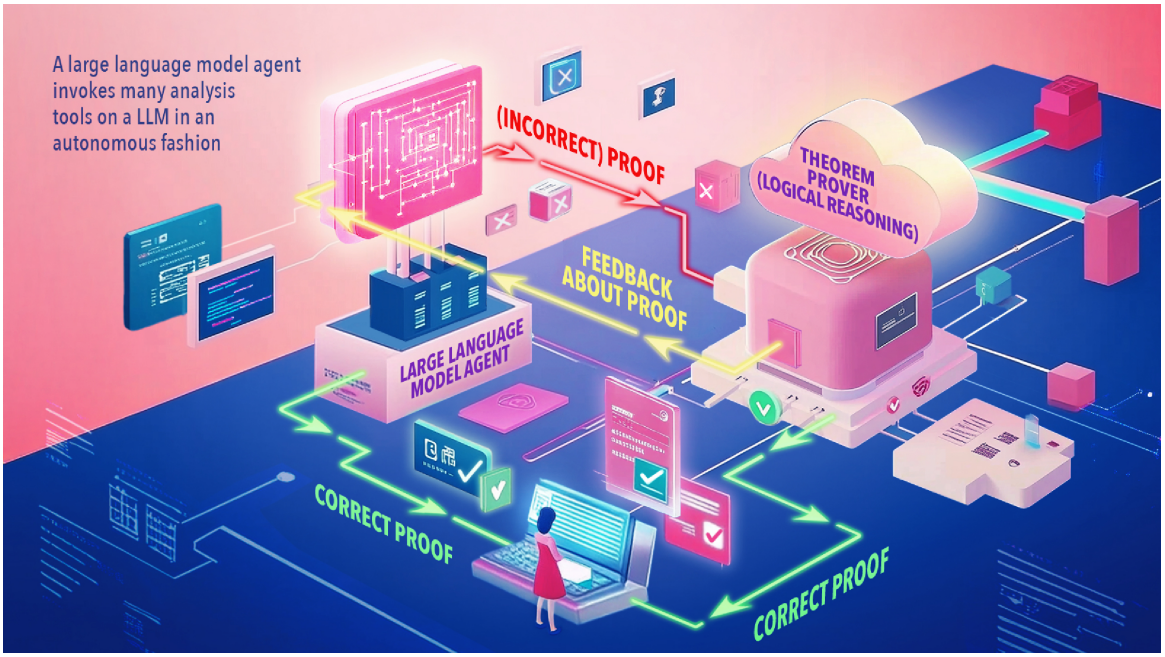

Fig. 1: Architecture of planned Theorem Proving engine for Program Verification

the intent inferred for code changes. Our recent work on *patch specifications* [15] could also be leveraged to write specification for code changes in a modular way.

2.2 AI-driven Proof Search

The other research direction that could have a significant impact on the scalability of program verification is AI-driven proof search. Consider the task of dispensing a proof obligation about a large software system via top-down proof construction. We can reuse and re-purpose the autonomous code search in LLM agents such as AutoCodeRover [16] to discover code elements involved in intermediate lemmas as well as the desiderata of proof obligations about such code elements. Instead of software verification amounting to a pure (and laborious) pushing of proof obligations via constraint propagation, GenAI technology can contribute significant *inventive steps* in discovering useful lemmas about different units of a codebase. These lemmas can prove properties of a large software system.

Inspired by the AlphaProof approach [17] for proving mathematical theorems, we believe a mechanism for program verification using GenAI is timely and feasible. We plan to use a theorem prover as a checker of proofs constructed by an LLM agent. The checked proofs may be provided with feedback based on the prover's proof structure, and this can improve the LLM agent's proof construction capability via a reinforcement learning loop. In time, we expect these verification workflows to be customized for software, hardware, and embedded systems.

3. Perspectives

A rough architecture of our modern AI-driven mathematical proof engine for computer programs can be seen in Figure 1.

While LLM-assisted proof generation has already been leveraged in several recent research projects [18, 19, 20, 21], a key insight that we plan to explore is the virtuous cycle between specification inference and proof search, trying to solve together, rather than separately, the two key challenges of program verification.

Furthermore, while constructing mathematical proofs, it is hard to move from a "state" of an unfinished proof attempt to make progress. Why? This is because a mathematician may have some intuitions about inventive steps about which lemmas may be fruitful to consider, which can be hard to encode via reward functions. This is a key planned innovation in our work. Instead of seeking to devise proof construction tactics capturing general mathematical intuition, our focus will be on program verification. For dispensing proof obligations about computer programs, the code search and program analysis capabilities in agentic AI approaches [13, 16] will be leveraged to capture intuitions about proving program properties.

Acknowledgments

We would like to thank David Lo and Wei Gao (from SMU), Martin Rinard (from MIT), Michael Shieh (from NUS), Peter Müller (from ETH) and Anil Bharath (from Imperial College) for discussions around AI for Program Reasoning.

References

[1] Greg Miller. A scientist's nightmare: Software problem leads to five retractions. *Science*, 314(5807):1856–1857, 2006.

[2] Zeeya Merali. Computational science:... error. *Nature*, 467(7317), 2010.

[3] Phillip Johnston and Rozi Harris. The Boeing 737 MAX saga: lessons for software organizations. *Software Quality Professional*, 21(3):4–12, 2019.

[4] Jean Yves Henrion and Thierry Vallée. V88 Ariane 501, 1997.

[5] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[6] Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. The Coq proof assistant: a tutorial. *Rapport Technique*, 178:113, 1997.

[7] Makarius Wenzel, Lawrence C Paulson, and Tobias Nipkow. The Isabelle framework. In *International Conference on Theorem Proving in Higher Order Logics*, pages 33–38. Springer, 2008.

[8] Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In *International Conference on Computer Aided Verification*, pages 1–35. Springer, 2013.

[9] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[10] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220, 2009.

[11] Gerwin Klein, June Andronick, Gabriele Keller, Daniel Matichuk, Toby Murray, and Liam O'Connor. Provably trustworthy systems. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 375(2104):20150404, 2017.

[12] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 42–54, 2006.

[13] Haifeng Ruan, Yuntong Zhang, and Abhik Roychoudhury. SpecRover: Code intent extraction via LLMs. *arXiv preprint arXiv:2408.02232*, 2024.

[14] Anthropic. Building effective agents. https://www.anthropic.com/research/building-effective-agents, 2024.

[15] Cristian Cadar, Daniel Schemmel, and Arindam Sharma. Patch specifications via product programs. In *2023 International Conference on Formal Methods in Software Engineering (FormaliSE 2023)*, pages 39–43, 5 2023.

[16] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. AutoCodeRover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1592–1604, 2024.

[17] Google DeepMind. AI achieves silver-medal standard solving International Mathematical Olympiad problems. https://deepmind.google/discover/blog/ai-solves-imo-problems-at-silver-medal-level/, 2024.

[18] Albert Qiaochu Jiang, Wenda Li, Jesse Michael Han, and Yuhuai Wu. LISA: Language models of ISAbelle proofs. In *6th Conference on Artificial Intelligence and Theorem Proving*, pages 378–392, 2021.

[19] Albert Qiaochu Jiang, Wenda Li, Szymon Tworkowski, Konrad Czechowski, Tomasz Odrzygóźdź, Piotr Miłoś, Yuhuai Wu, and Mateja Jamnik. Thor: Wielding hammers to integrate language models and automated theorem provers. *Advances in Neural Information Processing Systems*, 35:8360–8373, 2022.

[20] Kyle Thompson, Nuno Saavedra, Pedro Carrott, Kevin Fisher, Alex Sanchez-Stern, Yuriy Brun, João F. Ferreira, Sorin Lerner, and Emily First. Rango: Adaptive retrieval-augmented proving for automated software verification. In *Proceedings of the 47th International Conference on Software Engineering (ICSE)*, April 2025.

[21] Gabriel Poesia, David Broman, Nick Haber, and Noah D. Goodman. Learning formal mathematics from intrinsic motivation. In *Thirty-Eighth Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2024.