Towards LLM-support for Deductive Verification of Java Programs

Samuel Teuber[0000−0001−7945−9110] and Bernhard Beckert[0000−0002−9672−3291]

Karlsruhe Institute of Technology, Germany
{teuber,beckert}@kit.edu

1 Introduction & Background

Along with the success of Large Language Models (LLMs) in other domains [5], a quickly developing body of research is investigating the use of Large Language Models to analyze or generate program code [6, 8, 10, 12, 13, 15, 16, 18, 19]. Our work investigates using LLMs for generating (auxiliary) *specifications* for a real-world programming language (namely Java). In contrast to pure code generation, where LLMs could produce buggy code, specification generation allows us to rigorously check consistency between pre-existing source code and generated annotations through the use of theorem provers. This extended abstract summarizes our efforts [3, 17] to couple the deductive verification tool KeY with GPT to automate the process of annotating Java code with JML specifications.

KeY and JML. KeY [1] is an interactive theorem prover for Java Dynamic Logic [1, 2] (JavaDL) allowing the deductive verification of Java programs w.r.t. specifications written in the Java Modeling Language (JML) [11]. Given a piece of Java code and a JML specification, the artifacts are translated into a JavaDL formula which must be proven valid for successful verification. To this end, KeY implements a sequent-style calculus. Consider the example in Listing 1.1: Here, the method `f` is annotated with a JML specification which states that the method's return value must be `-2*x`. However, proving statements about `f` requires assumptions about `g`. To admit modular verification, KeY will use a JML specification on the behavior of `g` as a lemma to prove the correctness of `f`. Then, we also have to prove that `g` adheres to its assumed JML specification. This modular, auto-active [14] approach has been used in numerous real-world case studies [4, 7, 9] that even discovered a bug in OpenJDK's sort method [7]. However, while modular verification is very desirable to reduce complexity, it does require that we annotate submethods with suitable JML specifications. Similarly, we also have to annotate loops with invariants and variants to prove their correctness and termination. We call the specification of `f` a *top-level*specification. *Auxiliary* specifications (such as loop invariants or the specification of a method like `g`) are classically written manually by experts.

```
//@ ensures \result == -2*x;
int f(int x) {
    return g(-x);
}
int g(int x) {
    return x+x;
}
```

Listing 1.1. Callee method `g` lacks an annotation [3, 17]

2 Automated Specification Generation using LLMs

To decrease the user effort for deductively verifying Java programs, we have implemented a prototype that, given a Java program and its top-level specification, generates the required auxiliary specifications (in particular submethod specifications and loop invariants). Our approach is visualized in Figure 1: Given a partially annotated Java program, we use an LLM to generate missing annotations. Subsequently we call a verification tool, in this case KeY, which either successfully verifies the top-level specification or returns an error. In the latter case, we ask the LLM for a new/refined annotation. Our prototype either returns a basic error description from the verifier to the LLM (feedback-based approach) or just samples a different solution from the LLM without feedback (sampling-based approach).

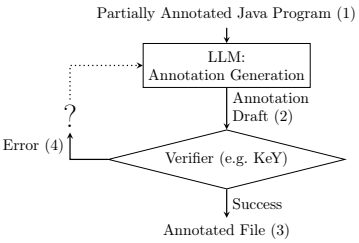

Fig. 1. Our LLM integration for KeY [17]

Evaluation. As an initial evaluation of our approach, we constructed a benchmark set of 27 benchmark instances for invariant generation and 14 benchmark instances for submethod specification generation. Each instance consists of a partial JML specification which misses one (auxiliary) submethod or loop specification that must be generated by the LLM. The benchmarks cover a wide range of Java and JML features supported by KeY including quantifiers, arrays, and field access [3]. The results for the feedback-based approach can be found in Table 1 and are quite promising given they were achieved with minimal prompt engineering.

Table 1. Overview of experimental results: mean (μ) and standard deviation (σ) of success rate across 5 runs [3]

Category	$\mu \pm \sigma$ of success rate (%)	
	GPT 3.5	GPT 4o
Submethods	19.3 ± 12.1	$\mathbf{40.5 \pm 4.1}$
Invariants	37.0 ± 7.4	$\mathbf{67.9 \pm 5.7}$

We also compared the sampling and feedback-based approaches [17]. Here, we compare the percentage of successfully solved instances for the two approaches for one to ten iterations (see Figure 2 for invariants benchmark set): KeY's current feedback seems insufficient to improve upon sampling. Moreover, our feedback approach sometimes gets "stuck" by minimally changing a wrong solution instead of pivoting to a correct one. The behavior for submethods is similar [17].

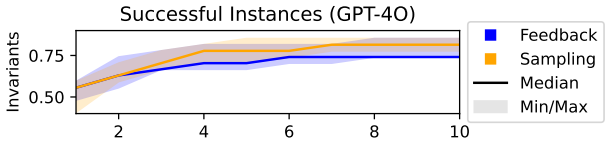

Fig. 2. Success of feedback-based and sampling-based invariant generation approach for different numbers of iterations (min/max across 5 runs)

References

1. Ahrendt, W. (ed.): Deductive Software Verification - The KeY Book - From Theory to Practice. Springer (2016)
2. Beckert, B.: A Dynamic Logic for the Formal Verification of Java Card Programs. In: Attali, I., Jensen, T.P. (eds.) Java on Smart Cards: Programming and Security, First International Workshop, JavaCard 2000, Cannes, France, September 14, 2000, Revised Papers. LNCS, vol. 2041, pp. 6–24. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-45165-X_2
3. Beckert, B., Klamroth, J., Pfeifer, W., Röper, P., Teuber, S.: Towards Combining the Cognitive Abilities of Large Language Models with the Rigor of Deductive Progam Verification. In: Margaria, T., Steffen, B. (eds.) ISoLA 2024, Crete, Greece, October 27-31, 2024, Proceedings, Part IV. LNCS, vol. 15222, pp. 242–257. Springer, Heidelberg (2024). https://doi.org/10.1007/978-3-031-75387-9_15
4. Beckert, B., Sanders, P., Ulbrich, M., Wiesler, J., Witt, S.: Formally Verifying an Efficient Sorter. In: Finkbeiner, B., Kovács, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part I. LNCS, vol. 14570, pp. 268–287. Springer, Heidelberg (2024). https://doi.org/10.1007/978-3-031-57246-3_15
5. Brown, T.B. *et al.*: Language Models are Few-Shot Learners. In: Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., Lin, H. (eds.) Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual (2020)
6. Chakraborty, S. *et al.*: Ranking LLM-Generated Loop Invariants for Program Verification. (2023). https://doi.org/10.18653/V1/2023.FINDINGS-EMNLP.614
7. de Gouw, S. *et al.*: Verifying OpenJDK's Sort Method for Generic Collections. J. Autom. Reason. **62**(1), 93–126 (2019). https://doi.org/10.1007/S10817-017-9426-4
8. Granberry, G., Ahrendt, W., Johansson, M.: Specify What? A Case-Study using GPT-4 and Formal Methods For Specification Synthesis. In: AI for Math Workshop @ ICML 2024 (2024). https://openreview.net/forum?id=ZRTcPkNl7v
9. Hiep, H.A. *et al.*: Verifying OpenJDK's LinkedList using KeY. In: Biere, A., Parker, D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 26th Intl. Conf. TACAS, Dublin, Ireland, Part II. LNCS,vol. 12079, pp. 217–234. Springer, Heidelberg (2020). https://doi.org/10.1007/978-3-030-45237-7_13
10. Janßen, C., Richter, C., Wehrheim, H.: Can ChatGPT support software verification? In: Beyer, D., Cavalcanti, A. (eds.) Fundamental Approaches to Software Engineering - 27th International Conference, FASE 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings. LNCS, vol. 14573, pp. 266–279. Springer, Heidelberg (2024). https://doi.org/10.1007/978-3-031-57259-3_13
11. Leavens, G.T. *et al.*: *JML Reference Manual*. Draft revision 2344. May 2013. http://www.eecs.ucf.edu/~leavens/JML//OldReleases/jmlrefman.pdf.
12. Kamath, A. *et al.*: Finding Inductive Loop Invariants using Large Language Models. CoRR **abs/2311.07948** (2023). arXiv: 2311.07948
13. Lathouwers, S., Huisman, M.: Survey of annotation generators for deductive verifiers. Journal of Systems and Software **211**, 111972 (2024). https://doi.org/10.1016/j.jss.2024.111972

14. Leino, K.R.M., Moskal, M.: Usable auto-active verification. In: Ball, T., Zuck, L., Shankar, N. (eds.) Usable Verification Workshop (2010). https://fm.csl.sri.com/UV10
15. Pei, K., Bieber, D., Shi, K., Sutton, C., Yin, P.: Can Large Language Models Reason about Program Invariants? In: Krause, A. (ed.) Proceedings of the 40th International Conference on Machine Learning. Proceedings of Machine Learning Research, pp. 27496–27520. PMLR (2023). https://proceedings.mlr.press/v202/pei23a.html
16. Sun, C., Sheng, Y., Padon, O., Barrett, C.W.: Clover: Closed-Loop Verifiable Code Generation. In: Avni, G. (ed.) AI Verification - First International Symposium, SAIV 2024, Montreal, QC, Canada, July 22-23, 2024, Proceedings. LNCS, vol. 14846, pp. 134–155. Springer, Heidelberg (2024). https://doi.org/10.1007/978-3-031-65112-0_7
17. Teuber, S., Beckert, B.: Next Steps in LLM-Supported Java Verification (Short Paper). In: 1st International Workshop on Neuro-Symbolic Software Engineering (NSE 2025). IEEE (2025). Forthcoming
18. Wu, H., Barrett, C., Narodytska, N.: Lemur: Integrating Large Language Models in Automated Program Verification. In: The Twelfth International Conference on Learning Representations (2024). https://openreview.net/forum?id=Q3YaCghZNt
19. Yao, J., Zhou, Z., Chen, W., Cui, W.: Leveraging Large Language Models for Automated Proof Synthesis in Rust. CoRR **abs/2311.03739** (2023). arXiv: 2311.03739