Automatic Documentation of Java Code with Formal Specifications using Artificial Intelligence*

Juan Carlos Recio[1], Rubén Saborido[1], and Francisco Chicano[1]

ITIS Software, Universidad de Málaga, Spain
{jcrecio,rsain,chicano}@uma.es

Abstract. We present a system that produces JML (Java Modeling Language) formal specifications of Java programming language code using Higher Order Logic (HOL). The system verifies that the formal specifications are correct using the Isabelle/HOL theorem prover assistant. Our approach combines automated unit test case generation, invariant detection, conversion between formal languages, and specialized Large Language Models (LLMs) for proof inference and formalized code documentation generation. This novel pipeline bridges the gap between practical software engineering and formal verification methods, enabling developers to automatically produce trustworthy documentation backed by mathematical proofs and formal statements that can later be combined in a modularized way to prove the correctness of large software systems.

Keywords: Formal verification · invariant detection · Large Language Models · JML · Isabelle/HOL

1 Introduction

Software verification is only applied to critical software systems (like device drivers or communication protocols) due to the important challenges it poses in modern software development [5], particularly when dealing with high-level programming languages. Although these languages offer powerful abstractions and extensive libraries, formally verifying their behavior requires bridging the gap between programming constructs and mathematical proofs [1]. The fundamental challenges arise from the difficulty of scaling formal verification to industrial-sized codebases and the inherent difference between programming languages and mathematical proofs: the former employs sequential operational semantics to specify procedural execution, whereas the latter establishes relationships through declarative assertions of properties and constraints that must be satisfied.

In the domain of software documentation generation, we observe significant advancements in the last years, leveraged by recent advances in artificial intelligence (AI) models and software engineering good practices. LLM-based tools

* This work has been partially funded by RED2022-134647-T (AI4Software) and PLEC2023-010266 (SOFIA).

like GitHub Copilot can produce contextually relevant doc-strings and sophisticated frameworks such as Swagger/OpenAPI streamline API documentation processes. Despite these advances, the field faces persistent challenges, including accuracy issues where LLMs may hallucinate functionality [10], limited context understanding beyond immediate files, documentation maintenance difficulties as code evolves, inadequate capture of technical debt rationale, challenges in adapting to domain-specific conventions, immature documentation testing capabilities, and accessibility concerns. These systems excel at summarizing code functionality and converting natural language into formal documentation, but struggle with maintaining synchronization between documentation and changing codebases.

2 Proposal

We present a comprehensive pipeline (see Figure 1) that aims to generate code documentation for Java methods that is exceptionally rigorous and trustworthy, expressed in terms of formal specifications. Our approach combines multiple tools and techniques to achieve this goal. Given a target Java project (stage 1 in

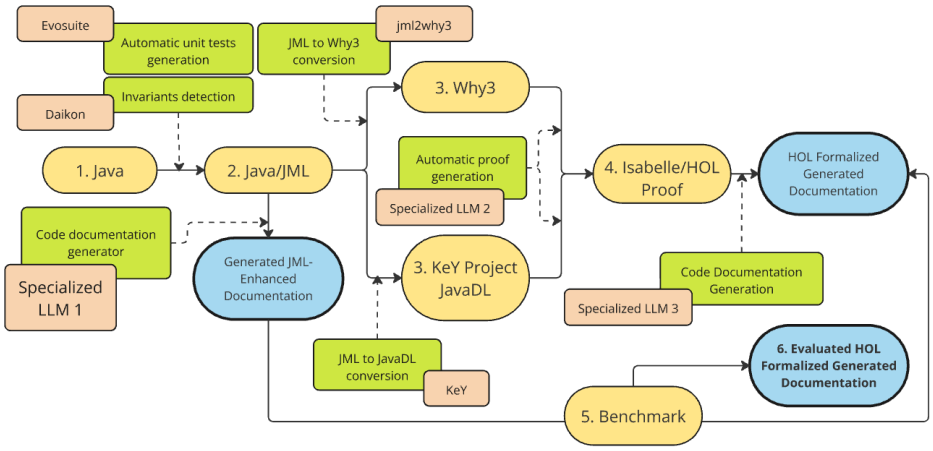

Fig. 1. Overview of the pipeline stages to generate code documentation. The yellow boxes represent the different stages for the original code until generating its documentation. The green boxes represent the actions to move between stages. The blue boxes represent the generated documentation. The orange boxes are the tools used. The stage 3 is *duplicated* to illustrate the two different approaches under investigation for translating JML into Isabelle/HOL.

Fig. 1) we use Evosuite [8], a tool based on evolutionary computation, to generate high-coverage unit tests. These tests provide diverse inputs for Daikon [4], the invariant detection tool we use in our system. The invariants discovered by

Daikon are used to annotate the original Java code with JML annotations (stage 2 in Fig. 1). The goal of the JML annotations is to serve as an intermediate formal specification that accurately captures the behavior and properties of Java programs. This specification becomes the base of our subsequent transformation steps. At this point, we use the Java/JML specifications to directly generate code documentation with an LLM to compare it later with the final generated documentation as a control step. Next, we check if the JML annotations are accurate, that is, we prove their correctness using the Isabelle/HOL proof assistant. To bridge the gap between JML specifications and Isabelle/HOL theories, we explore two promising approaches. The first utilizes the Why3 platform [6] (stage 3 up in Fig. 1), which acts as a natural intermediary due to its ability to understand both Java and HOL semantics, facilitated by the *jml2why3* tool [2]. The second approach leverages The KeY Project (stage 3 down in Fig. 1) [3], which offers an alternative path for connecting Java JML to Isabelle/HOL. Both approaches are currently under active investigation to determine their relative strengths and applicability. Once the formal specifications are expressed as Isabelle/HOL theorems (stage 4 in Fig. 1) we need to prove them. Inspired by Baldur [7], we use specialized LLMs for proof inference. Our models are based on math and Chain of Thought reasoning (CoT) [11]. We fine-tuned these models using theorems and proof data to enhance their ability to generate correct proofs efficiently. We alternatively use Retrieval Augmented Generation (RAG) [9], utilizing a vectorized database of Isabelle/HOL content to provide relevant context for proof inference. Once a proof is generated and verified with an Isabelle/HOL Checker, we feed both the proven theorem and the original JML-annotated code into another LLM, specifically trained for code documentation. This will generate comprehensive and mathematically-verified documentation for the original Java code to get better results than using a general approach (stage 5 in Fig. 1).

3 Conclusion

Although we still did not conduct an exhaustive experimentation using our proposal, we have tested it with tens of Maven-based open-source Java projects collected using the GitHub API. The translation of Java/JML specifications to Isabelle/HOL through Why3 as an intermediary faces substantial technical challenges stemming from the foundational differences between these systems. Java object-oriented model with inheritance hierarchies, null references, and generic types does not map directly to Why3ML-inspired type system. This process requires utilizing tools such as OpenJML and specialized Abstract Syntax Tree manipulations to effectively bridge the semantic gap between these formally distinct languages. The translation must preserve not only basic language constructs but also ensure the accurate representation of JML specification semantics throughout the multi-stage translation pipeline. For proof inference, we analyzed the performance using LLMs, since we think it is a critical point of our approach. Our preliminary experiments show that by fine-tuning the models we can reach 29% success rate in proof inference, while RAG only achieves 19% success rate.

Our claim is that using Reasoner fine-tuned LLMs like DeepSeek, might increase the performance drastically to infer correct proofs. For the final stage also we are experimenting with RAG and fine-tuned approaches for code documentation generation, based on Java, JML and an Isabelle/HOL proved theorem. However, we require a decent number of results from the previous stages. We might probably need some human evaluation of the generated documentation to assess the results.

References

1. Arceri, V., Cortesi, A., Ferrara, P., Olliaro, M.: Challenges of Software Verification, vol. 238. Springer Nature (2023)
2. Becker, B., Filliâtre, J.C., Marché, C.: Rapport d'avancement sur la vérification formelle des algorithmes de ParcourSup. Technical report, Université Paris-Saclay (Jan 2020), https://inria.hal.science/hal-02447409
3. Beckert, B., Hähnle, R., Schmitt, P.H.: Verification of object-oriented software. the key approach - foreword by k. rustan m. leino. In: The KeY Approach (2007), https://api.semanticscholar.org/CorpusID:37721082
4. Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D.: Dynamically discovering likely program invariants to support program evolution. In: Proceedings of the 23rd International Conference on Software Engineering. pp. 213–224. ICSE '01, IEEE Computer Society, Washington, DC, USA (2001). https://doi.org/10.1109/ICSE.2001.919095
5. Ferrara, P., Arceri, V., Cortesi, A.: Challenges of software verification: The past, the present, the future. International Journal on Software Tools for Technology Transfer . https://doi.org/10.1007/s10009-024-00765-y, https://doi.org/10.1007/s10009-024-00765-y
6. Filliâtre, J.C., Paskevich, A.: Why3 – where programs meet provers. In: Proceedings of the 22nd European Symposium on Programming (ESOP 2013). pp. 125–128. Springer (2013). https://doi.org/10.1007/978-3-642-37036-6_8
7. First, E., Rabe, M.N., Ringer, T., Brun, Y.: 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. p. 1229–1241. ESEC/FSE 2023, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3611643.3616243
8. Fraser, G., Arcuri, A.: EvoSuite: Automatic test suite generation for object-oriented software. Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering pp. 416–419 (2011). https://doi.org/10.1145/2025113.2025179
9. Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.t., Rocktäschel, T., Riedel, S., Kiela, D.: Retrieval-augmented generation for knowledge-intensive nlp tasks. In: Proceedings of NIPS. NIPS '20, Curran Associates Inc., Red Hook, NY, USA (2020)
10. Liu, F., Liu, Y., Shi, L., Huang, H., Wang, R., Yang, Z., Zhang, L., Li, Z., Ma, Y.: Exploring and evaluating hallucinations in llm-powered code generation (2024), https://arxiv.org/abs/2404.00971
11. Wei, J., Wang: Chain-of-thought prompting elicits reasoning in large language models. In: Proceedings of the 36th International Conference on Neural Information Processing Systems. NIPS '22, Curran Associates Inc., Red Hook, NY, USA (2022)