SpecForge: Auto-Formalization of Software Specifications through Symbolic Reasoning and Large Language Models

Juan Zhai, Assistant Professor, juanzhai@umass.edu
Manning College of Information and Computer Sciences, University of Massachusetts Amherst

Abstract

AI-assisted programming tools can generate plausible code from natural language instructions, but without formal specifications their outputs cannot be rigorously verified. Automated reasoning technologies can guarantee correctness only when precise formal semantics are available. SpecForge addresses this gap through a neurosymbolic framework that transforms natural language intent into verified formal specifications. It combines large language models, lightweight program analysis, test-based validation, and symbolic reasoning in a structured, multi-stage pipeline that generates, validates, and iteratively refines candidate specifications.

The system begins with syntax and symbol resolution to ground generated specifications in the program context. It then detects and repairs specification defects, including vacuity, inconsistency, and ambiguity, through test-based and symbolic validation, where LLMs interpret reasoning feedback to guide repair. SpecForge aligns formal semantics with developer intent by interpreting solver feedback, repairing faulty specifications, and generating clarification questions for ambiguous cases. SpecForge establishes a transparent pipeline for converting informal requirements into verifiable formal semantics, advancing the practicality of explainable and trustworthy automated reasoning in software development.

Keywords: Specification Inference; Neurosymbolic Reasoning; Formal Methods; Human-AI Alignment.

Introduction

Large language models (LLMs) have made it possible to generate code directly from natural language (NL) descriptions, transforming how developers express intent. However, the code correctness remains difficult to assess. Developers often describe expected behavior through comments, examples, and tests, but these informal artifacts are ambiguous and lack the formal semantics needed for rigorous reasoning. This gap limits the use of automated reasoning systems [1–5], which rely on explicit formal specifications that define exactly what a program must do to verify correctness. Bridging this gap requires an approach that can infer, validate, and refine specifications automatically while keeping the process transparent and usable for developers.

Existing research explores the automatic inference of formal specifications from NL descriptions. Early pattern- and search-based techniques [6–10] rely on handcrafted templates and heuristics, offering limited scalability and generalization. Recent advances in LLMs show greater promise [11–15]. However, these approaches remain largely exploratory and depend solely on generative output. The resulting specifications often contain hallucinated symbols and exhibit logical defects such as vacuity, inconsistency, and ambiguity.

This project introduces SpecForge, a neurosymbolic framework that validates and refines LLM-generated specifications through a structured, multi-stage pipeline integrating lightweight program analysis, empirical testing, and symbolic reasoning. SpecForge first generates candidate specifications from NL comments, function signatures, and optional input-output examples using a LLM. It then validates and repairs these candidates across four stages: syntax and symbol repair, vacuity screening, inconsistency diagnosis, and ambiguity detection. By combining generative modeling with formal reasoning, SpecForge produces specifications that are verifiable, transparent, and closely aligned with developer intent. The system enhances human understanding by surfacing counterexamples and generating clarification questions for ambiguous cases. In doing so, SpecForge makes specification inference a systematic, explainable, and developer-guided process, advancing the practical adoption of formal methods in AI-driven software development.

Methods

Figure 1 illustrates the proposed SpecForge pipeline. It begins by using a LLM (e.g., Amazon Bedrock [16]) to generate candidate specifications from a method's signature, NL comments, and available test cases. These candidates are then validated and refined through four stages: syntax and symbol validation, vacuity checking, inconsistency diagnosis, and ambiguity clarification. After each refinement, specifications are revalidated through the relevant prior checking stages in order, starting with syntax and symbol validation, followed by vacuity and inconsistency checks to ensure correctness before advancing. The result is a set of verified specifications with solver evidence and an ambiguity report containing distinguishing inputs and clarification questions. SpecForge encodes each specification as a single `requires` or `ensures` statement in a Boogie-compatible [2] contract language. Specifications are type-checked and translated into SMT-LIB for reasoning with the Z3 solver [3], ensuring interoperability with existing verification frameworks and enabling iterative LLM-guided refinement.

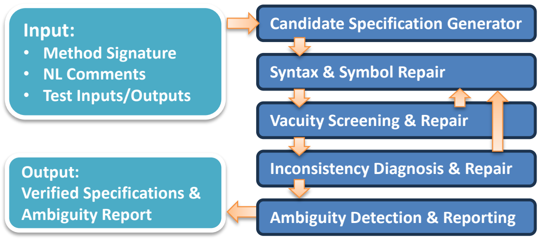
Input:
Method Signature
NL Comments
Test Inputs/Outputs
Candidate Specification Generator
Syntax & Symbol Repair
Vacuity Screening & Repair
Inconsistency Diagnosis & Repair
Ambiguity Detection & Reporting
Output:
Verified Specifications & Ambiguity Report

Figure 1: SpecForge Overview

Syntax & Symbol Repair. SpecForge statically parses each specification, type-checks it, and resolves symbols against the program scope. Missing or out-of-scope references are repaired. For undefined identifiers (e.g., a hallucinated predicate like `isValid(user)`), SpecForge employs a two-step LLM workflow: (1) infer the NL intent of a missing symbol and (2) generate a grounded replacement by matching that intent to in-scope identifiers. Lightweight program analysis tools (e.g., Tree-sitter [17]) retrieve in-scope symbols, which are ranked by semantic similarity computed over identifier names and documentation using models such as CodeBERT [18]. The top matches are passed to the LLM to synthesize a concrete replacement (e.g., `user != null && user.active == true`). The patched specification is re-evaluated before proceeding.

Vacuity Screening & Repair. SpecForge detects specifications that are unreachable or uninformative and strengthens them through LLM-guided refinement using concrete and symbolic evidence. Vacuity arises in two forms: *reachability*, which tests whether any input satisfies the precondition, and *informativeness*, which checks whether the postcondition adds constraints beyond the precondition.

SpecForge begins with test-based probing for empirical assessment. Using available input–output pairs, it checks whether the SMT solver can find a test input satisfying the precondition. If none exists, the specification is unreachable. For informativeness, when a test triggers the precondition, SpecForge perturbs the corresponding output and re-evaluates the postcondition. If it always holds, the specification is uninformative. Empirically validated specifications are then analyzed for symbolic vacuity. A specification is unreachable when $\mathsf{UNSAT}(Pre)$, and uninformative when $\mathsf{UNSAT}(Pre \wedge \neg Post)$. Vacuous specifications are flagged for LLM-guided strengthening, where the model interprets test failures and symbolic evidence to propose revised formulations. Each revision is revalidated through syntax & symbol repair and rechecked for vacuity.

Inconsistency Diagnosis & Repair. SpecForge ensures the remaining specifications are feasible and repairs them when contradictions arise. It performs concrete feasibility checks using test cases, followed by symbolic analysis with the SMT solver. In the concrete phase, for each test input that satisfies a precondition, the solver verifies whether at least one corresponding postcondition holds for the expected output. If none do, the input–output pair is recorded as a counterexample. Specifications that pass these empirical checks then undergo symbolic feasibility reasoning by solving $\mathsf{SAT}(Pre \wedge Post)$. When the formula is unsatisfiable, the solver extracts an unsat core identifying the minimal conflicting predicates. The unsat core and failed test pairs are passed to the LLM, which explains the contradiction. The LLM then proposes localized repairs, such as adjusting the logical relation between inputs and outputs or refining the applicable condition. Each revision is revalidated through syntax and vacuity checks before rechecking the inconsistency.

Ambiguity Detection & Reporting. SpecForge detects ambiguity by examining relationships among specifications to identify when logically valid ones express different interpretations of the same intent. The process includes three steps: *relevance filtering*, *symbolic comparison*, and *interpretive clarification*, and generates NL explanations and questions that summarize each ambiguity and prompt developer confirmation.

Relevance filtering determines which specifications likely describe the same behavior. It clusters candidates based on symbolic and linguistic similarity, using shared variables, overlapping guard predicates, and comparable syntactic structures as evidence. Token overlap, normalized predicate embeddings, and structural similarity help form these clusters, ensuring that only related specifications are compared.

Within each cluster, SpecForge uses symbolic reasoning to identify inputs on which two specifications produce different semantic outcomes. The solver searches for inputs and, if relevant, outputs that satisfy $Pre_1 \wedge Post_1 \wedge \neg Post_2$ or its converse, focusing only on overlapping behavioral regions where both specifications apply.

When a distinguishing input is found, SpecForge uses test data and LLM interpretation to explain the difference. Even if the exact input is absent from the test set, nearby examples provide context that helps determine which interpretation aligns more closely with observed behavior. Test data also indicates which ambiguities affect commonly tested behaviors, helping prioritize those most relevant to real usage. The LLM then translates the formal divergence into a clear, developer-oriented question such as "`Should the returned item be the first item before or after the specified element is removed?`".

Expected Results and Milestones

This project will deliver the SpecForge prototype and a dataset containing methods from open-source repositories with NL comments, test cases, and verified specifications. The framework applies to any codebase with function-level documentation, including open-source projects and industry-scale systems with documentation and test suites, such as standard libraries, service APIs, and business logic within AWS SDKs.

We will evaluate the system quantitatively and qualitatively, showing that test-based reasoning efficiently filters low-quality candidates while symbolic reasoning ensures logical soundness. Key metrics include symbol recovery accuracy, reductions in vacuous and inconsistent specifications, ambiguity resolution success, and human judgments of clarity and usefulness. The project milestones are as follows:

- **M1–3:** Collect repository methods, comments, and test cases to build dataset.
- **M4–6:** Release initial SpecForge prototype with syntax and symbol repair and baseline validation.
- **M7–9:** Add vacuity, inconsistency analysis, and ambiguity detection.
- **M10–12:** Perform full evaluation, release code and dataset, and begin drafting publications.

All datasets, models, and code will be released openly. SpecForge will demonstrate how symbolic reasoning, empirical testing, and neural modeling reinforce one another, creating a rigorous, interpretable framework for automated formalization and verifiable AI-assisted software development.

Potential Applications to Amazon. SpecForge aligns with tools such as Amazon Q Developer [19], which generates code from NL prompts, and with the AWS Automated Reasoning Group, which formally verifies systems like s2n-tls [20]. By automatically generating and validating formal specifications, SpecForge can supply correctness contracts for Amazon Q Developer outputs and strengthen AWS verification pipelines, improving the security, reliability, and traceability of cloud-scale software systems.

Assumptions and Limitations. SpecForge assumes access to representative code–comment pairs and input–output tests reflecting intended behavior. It relies on LLMs for specification generation and SMT solvers for symbolic reasoning. While data misalignment or solver limits may cause incomplete results, the multi-stage validation mitigates these risks and future work will enhance robustness under noisy inputs.

Tool Release and Maintenance. SpecForge and its dataset will be released under the open-source MIT license with documentation, tutorials, and examples on GitHub. The tool will be actively maintained with regular updates by a core contributor during and after the project.

Funds Needed and Personnel

We ask for $80,000 USD in cash funding and $40,000 in AWS Promotional Credits. The funds will be used for hiring Ph.D. students, performing the tasks, and covering travel expenses.

Appendix A - References

[1] K Rustan M Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 348–370. Springer, 2010.

[2] K. Rustan M. Leino. This is Boogie 2. Technical report, Microsoft Research, 2008.

[3] L. De Moura and N. Bjørner. Z3: An efficient smt solver. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[4] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer Science & Business Media, 2013.

[5] Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. *Isabelle/HOL: a Proof Assistant for Higher-order Logic*. Springer, 2002.

[6] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /* iComment: Bugs or Bad Comments?*. In *ACM SIGOPS Operating Systems Review*, pages 145–158. ACM, 2007.

[7] Lin Tan, Yuanyuan Zhou, and Yoann Padioleau. aComment: Mining Annotations from Comments and Code to Detect Interrupt Related Concurrency Bugs. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 11–20. IEEE, 2011.

[8] Juan Zhai, Yu Shi, Minxue Pan, Guian Zhou, Yongxiang Liu, Chunrong Fang, Shiqing Ma, Lin Tan, and Xiangyu Zhang. C2S: Translating Natural Language Comments to Formal Program Specifications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 25–37, 2020.

[9] Alberto Goffi, Alessandra Gorla, Michael D Ernst, and Mauro Pezzè. Automatic Generation of Oracles for Exceptional Behaviors. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 213–224. ACM, 2016.

[10] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. Translating Code Comments to Procedure Specifications. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 242–253, Amsterdam, Netherlands, July 2018. ACM.

[11] 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.

[12] Danning Xie, Byungwoo Yoo, Nan Jiang, Mijung Kim, Lin Tan, Xiangyu Zhang, and Judy S Lee. Impact of Large Language Models on Generating Software Specifications. *arXiv preprint arXiv:2306.03324*, 2023.

[13] Matthias Cosler, Christopher Hahn, Daniel Mendoza, Frederik Schmitt, and Caroline Trippel. nl2spec: Interactively Translating Unstructured Natural Language to Temporal Logics with Large Language Models. In *International Conference on Computer Aided Verification*, pages 383–396. Springer, 2023.

[14] Jens U Kreber and Christopher Hahn. Generating Symbolic Reasoning Problems with Transformer GANs. *arXiv preprint arXiv:2110.10054*, 2021.

[15] Jialun Cao, Yaojie Lu, Meiziniu Li, Haoyang Ma, Haokun Li, Mengda He, Cheng Wen, Le Sun, Hongyu Zhang, Shengchao Qin, et al. From Informal to Formal—Incorporating and Evaluating LLMs on Natural

Language Requirements to Verifiable Formal Proofs. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics*, 2025.

[16] Amazon Bedrock. `https://aws.amazon.com/bedrock/`, 2025.

[17] Max Brunsfeld. Tree-sitter: A Parser Generator Tool and Incremental Parsing Library. `https://tree-sitter.github.io/tree-sitter/`, 2018.

[18] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. *arXiv preprint arXiv:2002.08155*, 2020.

[19] Amazon Q Developer. `https://aws.amazon.com/q/developer/`, 2025.

[20] An Implementation of the TLS/SSL Protocols. `https://github.com/aws/s2n-tls`, 2025.