Can LLMs Reason About Program Semantics? A Comprehensive Evaluation of LLMs on Formal Specification Inference

Anonymous ACL submission

Abstract

Large Language Models (LLMs) are increasingly being used to automate programming tasks. Yet, LLMs' capabilities in reasoning about program semantics are still inadequately studied, leaving significant potential for further exploration. This paper introduces FormalBench, a comprehensive benchmark designed to evaluate LLMs' reasoning abilities on program semantics, particularly via the task of synthesizing formal program specifications to assist verifying program correctness. This task requires both comprehensive reasoning over all possible program executions (i.e., *completeness*) and the generation of precise, syntactically correct expressions that adhere to formal syntax and semantics (i.e., *consistency*). Using this benchmark, we evaluated the ability of LLMs in synthesizing consistent and complete specifications. Our findings show that LLMs perform well with simple control flows but struggle with more complex structures, especially loops, even with advanced prompting. Additionally, LLMs exhibit limited robustness against semantic-preserving transformations. We also highlight common failure patterns and design self-repair prompts, improving success rates by 25%. FormalBench is packaged as a Pip library and will be released upon publication. An early access version can be found at https://anonymous.4open.science/r/FormalBench-6C2F/README.md.

1 Introduction

Recent advances in Large Language Models (LLMs) have demonstrated significant potential for code understanding and generation (Hou et al., 2024; Chen et al., 2021). However, as adoption grows, critical concerns emerge about their reliability in programming tasks, particularly their capacity to reason about program semantics (Liu et al., 2024c; Yang et al., 2024b; Liu et al., 2024d). A fundamental question remains: *Can LLMs reason about program semantics?* Pioneering studies (Pei et al., 2023; Chen et al., 2025) have tackled this challenge by evaluating LLMs on *partial* semantic properties, such as predicting execution traces or inferring likely program invariants. Although these efforts provide valuable insights, they examine narrow aspects of program behavior rather than a comprehensive semantic understanding. For example, execution-based evaluations (Chen et al., 2025; Jain et al., 2024) are limited to specific execution paths and inputs, offering an incomplete view of LLM's semantic reasoning.

```
//@ requires num >= 0 && t >= 0;
//@ requires num + 2*t <= Integer.MAX_VALUE;
//@ requires num + 2*t >= Integer.MIN_VALUE;
//@ ensures \result == num + 2*t;
public int theMaximumAchievableX(int num, int t) {
  int res = num;
  //@ maintaining res == num + 2*(i-1);
  //@ maintaining i >= 1 && i <= t+1;
  //@ decreasing t-i+1;
  for(int i = 1; i <= t; i++) {
    res = res + 2;
  }
  return res;
}
```

Figure 1: Illustration of a Java program annotated with JML specifications (highlighted in green).

Recent studies (Wen et al., 2024; Ma et al., 2024) have evaluated LLMs in the synthesis of program specifications expressed in formal languages such as JML (Leavens et al., 2006) and ACSL (Baudin et al., 2008). The synthesized specifications can then be used to assist in automated software verification (D'silva et al., 2008) and bug finding (Le et al., 2022). This task challenges LLMs to (1) reason exhaustively over all possible program executions and (2) generate logically precise expressions that comply with the formal syntax and semantics of the specification language.

While initial results are promising, current evaluation methodologies for LLM reasoning through formal specification inference face three key limitations. First, the evaluation datasets are *small and lack diversity*. For example, SpecGenBench (Ma

et al., 2024) and the Frama-C problems (Kirchner et al., 2015) contain only 120 and 57 programs, respectively. Second, evaluation metrics focus *narrowly on consistency*, i.e., alignment between specifications and programs, while neglecting completeness, i.e., coverage of all semantic behaviors. Completeness is particularly important for assessing LLM's ability to reason comprehensively about *complete* program behaviors. Finally, current studies primarily aim to develop new LLM-based techniques for formal specification inference rather than *evaluating the LLM reasoning capabilities* themselves. Consequently, evaluations are often ad hoc, relying on specific prompting techniques or LLMs, leading to a lack of comprehensive insights across a wide range of models and prompts.

To address the above challenges, we introduce FormalBench, a comprehensive benchmark for evaluating the reasoning capabilities of LLMs through formal specification inference. FormalBench improves the existing dataset with two notable features: (1) a large-scale dataset of 700 manually validated Java programs and 6,219 augmented programs, covering various control flow structures, and (2) a Python library with a robust suite of evaluation metrics to measure both consistency (via deductive verification) and completeness (through mutation analysis). We then leverage FormalBench to conduct a comprehensive study evaluating eight state-of-the-art LLMs across four critical dimensions: (1) their effectiveness in synthesizing complete and consistent specifications, (2) robustness against semantic-preserving code transformations, (3) impact of advanced prompting techniques, and (4) the root causes of failures and their self-repair ability.

Our findings reveal several key insights. LLMs demonstrate limited effectiveness, achieving only about 10% verification success with over 50% failures, particularly struggling with complex control-flow structures such as nested loops. Advanced prompting techniques, such as few-shot and least-to-most prompting, improve success rates to 16.6% and reduce failures, yet overall performance remains suboptimal. Robustness issues also arise, with LLMs exhibiting flip rates between 27.2% and 39.2% under semantic-preserving transformations, negatively impacting their performance. Common failures of LLMs include syntax errors, flawed inductive reasoning, incorrect postconditions, faulty loop invariants, and misjudged arithmetic bounds. However, error-specific prompts enhance LLM self-repair capabilities, improving verifiable specifications by approximately 25% and reducing failures by around 40%, although these improvements converged after a few iterations.

In summary, our main contributions include:

- We introduce FormalBench, a comprehensive dataset specifically designed for evaluating formal reasoning of LLMs about program semantics.
- We propose a robust set of evaluation metrics to assess the effectiveness and robustness of LLMs on synthesizing consistent and complete formal specifications.
- We conduct an extensive empirical study of popular LLMs using FormalBench, highlighting their limitations. We also identify common failure patterns and design customized prompts to assist LLMs in self-repairing these failures.
- We advance research in formal specification inference by releasing FormalBench as an installable Python library under Apache 2.0 License, lowering barriers for academic research and establishing foundational benchmarks and metrics for future work.

2 Problem Statement

Given an input program, the formal specification inference task is to annotate the program with a set of formal specifications, i.e., Boolean expressions written in a formal specification language. A good formal specification should be adequate, consistent, unambiguous, complete, satisfied, and minimal (Lamsweerde, 2000). In this work, we particularly focus on two key properties of specifications, including completeness and consistency, which present the correctness of generated specifications. Inspired by (Lamsweerde, 2000), we define these properties in our problem as follows:

Definition 1. (Consistency) A formal specification is considered **consistent** to a given input program if all specified properties are well-formed and true with respect to that program.

Definition 2. (Completeness) A formal specification is considered **complete** if all function properties that hold with respect to a given input program are specified in the specification.

To determine the consistency of LLM-generated specifications, i.e., specifications that hold for an input program, we use deductive verification tools that transform the annotated program into logical

proof obligations and verify them with theorem provers. These tools ensure software correctness by systematically analyzing all possible execution paths, making our consistency checking reliable.

Measuring the completeness of formal specifications is inherently challenging because of the complex behaviors exhibited by software programs. Inspired by the success of mutation testing (Andrews et al., 2005) in evaluating the completeness of test suites, we propose to use mutation analysis as a proxy to assess the completeness of formal specifications. Mutation analysis generates non-equivalent mutant variants of input programs by introducing artificial faults (Andrews et al., 2005). Ideally, a complete specification should be able to detect all such faults. Therefore, we measure the proportion of mutants that violate the specification as a proxy of its completeness.

3 Dataset Construction and Evaluation

3.1 FormalBench Construction

FormalBench is constructed in three phases to ensure its reliability and diversity: (1) curating reference Java programs paired with natural language descriptions, (2) manually verifying program correctness with respect to natural language descriptions to establish FormalBench-Base, which comprises 700 programs, and (3) augmenting FormalBench-Base using semantic-preserving transformations to create FormalBench-Diverse, which comprises 6,219 programs.

Specifically, we begin with an initial pool of 966 Java programs generated by the MBXP model (Athiwaratkun et al., 2022) for the MBJP benchmark. This dataset is released under the Apache License 2.0, which permits modification and redistribution. To filter incorrect candidates, we execute these programs against the MBJP test suite, retaining 824 programs that pass all the provided test cases. However, as MBJP's test suite is generally weak, we conduct manual validation to ensure alignment between program behavior and natural-language intents. This involves a multi-step review process: we carefully inspect each program, augment the test suite with adversarial inputs, and validate correctness against the specified intents.

The result is FormalBench-Base, a rigorously validated dataset of 700 programs with provably correct implementations. To ensure diversity, FormalBench-Base covers a wide range of control

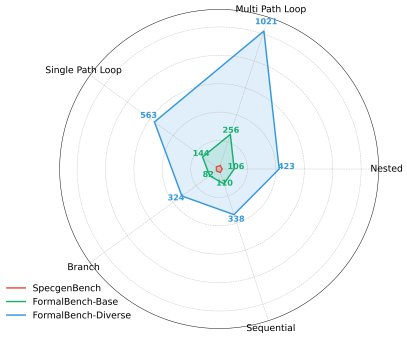


Figure 2: Distribution of our datasets and SpecGenBench over different control flow types.

flow types, including sequential, branching, single-path loops, multi-path loops, and nested loop structures, as illustrated in Figure 2.

Finally, we apply 18 semantic preservation transformations from the literature (Le-Cong et al., 2024; Rabin et al., 2021; Zhang et al., 2023) to FormalBench-Base, generating FormalBench-Diverse, a dataset of 6,219 program variants designed to evaluate the robustness of LLMs against syntactic variations. These transformations, detailed in Appendix A, span multiple levels of code structure, including naming (e.g., variable renaming), expression (e.g., switching equal expressions) and statement (e.g., transforming switch statements to if-statements).

3.2 Evaluation Metrics

Consistency Metrics. As mentioned in Section 2, we utilize deductive verification tools to determine the consistency of LLM-generated specifications. The verification process produces three outcomes: (1) *Verification Success*, where the implementation satisfies the specification; (2) *Verification Failure*, where the implementation violates the specification; and (3) *Unknown*, where the tool cannot definitively determine the result (e.g. due to timeouts or undecidability). While prior works (Wen et al., 2024; Ma et al., 2024) often merge *unknown* cases with failures, we observed that specifications that lead to timeouts are qualitatively distinct from those that cause failures. Thus, we treat *Unknown* as a separate category. Based on these categories, we define two consistency metrics as follows: (1) *Success Rate (SR)*, the proportion of specifications that pass verification; and (2) *Failure Rate (FR)*, the proportion of specifications that fail verification.

Completeness Metrics. To evaluate the com-

pleteness of a specification, we first apply mutation testing (Andrews et al., 2005) to generate a set of mutants, i.e., non-equivalent variants of the input programs created by deliberately injecting artificial faults. We then compute the fraction of these mutants that fail to satisfy the specification. This fraction is defined as the Completeness Rate (CR) of the specification.

Robustness Metrics. To evaluate the robustness of LLMs in synthesizing specifications under semantic-preserving transformations, we measure the *Flip Rate (FlR)*, which captures cases where LLMs generate verifiable specifications for the original program p but fail for its transformed versions. Moreover, we also measure the impact of unrobust behaviors on the performance of LLMs by measuring the success rate and failure rates on *FormalBench-Diverse* (Section 3.1). Since transformations may not apply universally, we normalize metrics over applicable transformations.

Details. We use OpenJML version 21.0 as our deductive verification tool and Major 3.0.1 as our mutation analysis tool. Full implementation details and formal formulations of these evaluation metrics are provided in Appendix B.

4 Experiments

In this section, we present our empirical results on LLMs using FormalBench, guided by the following research questions:

- **RQ**$_1$: *How effective are LLMs in synthesizing formal specifications?*
- **RQ**$_2$: *Can advanced prompting techniques improve the effectiveness of LLM?*
- **RQ**$_3$: *How robust are LLMs in synthesizing formal specifications?*
- **RQ**$_4$: *What are the common mistakes made by LLMs, and can they self-repair these errors?*

Following prior works (Ma et al., 2024; Flanagan and Leino, 2001), we focus on Java and its specification language, JML (Leavens et al., 2006), using OpenJML (Cok, 2011) as the verifier. LLMs are evaluated using their official chat templates and the same query prompts, as detailed in Appendix H. To ensure fairness, we use sampling with a temperature setting of 0.7 across all LLMs. For open-source LLMs, the maximum number of tokens generated is limited to 2048 due to GPU constraints. A full description of the experimental setup is provided in Appendix D.

Cost Analysis. Our experiments for closed-source LLMs cost approximately 250 USD, while those for open-source LLMs required around 100 GPU hours.

4.1 RQ$_1$: Effectiveness of LLMs

To answer the RQ$_1$, we evaluate the effectiveness of LLMs in synthesizing complete and consistent specifications by measuring their success rates, failure rates, and completeness rates. We assess eight popular open-source and proprietary LLMs on FormalBench-Base, as detailed in Appendix C. Detailed experimental results are presented in Table 1.

LLMs with zero-shot prompts. From the results in Table 1, we observe that LLMs with zero-shot prompts perform poorly in synthesizing formal specifications, achieving a success rate of around 10% and failure rates ranging from 56.4% to 99.6%. Most open source LLMs, except CodeQwen-2.5, exhibit particularly poor performance, with success rates below 3%. Upon closer analysis, we found that this poor performance of open-source LLMs is due to a lack of familiarity with the JML syntax, resulting in a significant number of invalid responses, up to 77%. For example, these models often generate natural language descriptions instead of formal JML specifications, highlighting their inability to produce follow formal grammar of JML without explicit guidance.

LLMs with few-shot prompts. To address these limitations, we enhanced the ability of LLMs to generate formal specifications by incorporating additional instructions on JML syntax and providing two demonstration examples in the few-shot prompts, as shown in the Appendix H. From the results in Table 1, we can see that this approach significantly improves LLM performance, with increases of up to 7.1 percentage points in success rates and reductions of up to 16.9 percentage points in failure rates. Furthermore, we observe a slight decline in completeness, although the overall completeness of the generated specifications remains high. This suggests a reasonable trade-off between correctness and consistency. However, despite these improvements, the success rates remain relatively low at less than 16%, highlighting the inherent challenge of synthesizing formal specifications for LLMs.

Open-source vs. Proprietary LLMs. Additionally, we observe that proprietary LLMs such as DeepSeek-V3 and GPT-4o are significantly more effective than open-source LLMs. However, the best-performing open-source LLM, CodeQwen-

	Models	Success Rate (%)	Failure Rate (%)	Completeness (%)
Open-Source LLMs	CodeQwen-1.5-7B	1.1	97.4	79.1
	+ Few-shot prompt	3.9	85.6	74.1
	CodeQwen-2.5-32B	7.6	77.1	83.3
	+ Few-shot prompt	11.4	66.8	82.1
	+ COT	9.2	69.4	86.8
	+ LTM	12.0	68.2	89.1
	DeepSeek-V2-236B	2.7	88.8	77.0
	+ Few-shot prompt	6.9	78.4	78.4
	+ COT	7.9	76.4	77.1
	+ LTM	8.9	70.4	81.6
	CodeLLaM-34B	0.1	99.6	100.0
	+ Few-shot prompt	5.3	82.7	60.3
Proprietary LLMs	DeepSeek-V3-671B	8.4	65.2	89.6
	+ Few-shot prompt	15.5	56.2	85.2
	+ COT	16.2	55.9	85.3
	+ LTM	16.6	56.8	89.6
	GPT-3.5	6.9	75.8	62.3
	+ Few-shot prompt	12.6	59.8	59.0
	o3-mini	10.0	66.2	83.5
	+ Few-shot prompt	11.7	59.7	88.7
	GPT-4o	11.2	56.4	80.4
	+ Few-shot prompt	13.4	56.4	77.6
	+ COT	13.4	61.5	81.2
	+ LTM	15.0	57.7	86.4
	Claude-3.5-Sonnet	10.5	64.5	91.2
	+ Few-shot prompt	14.7	53.1	82.6
	+ COT	14.2	59.1	83.6
	+ LTM	15.4	51.1	86.4

Table 1: Performance comparison of Open-Source and Commercial LLMs under zero-shot, in-context learning with few-shot prompt, chain-of-thought (COT), and least-to-most (LTM) prompting settings.

2.5, shows considerable promise with a success rate of 12.0%, only 3 percentage points lower than GPT-4o. This performance is particularly impressive given CodeQwen-2.5's compact size of 32B parameters. These findings suggest that open-source LLMs still have significant potential for further advancements in this domain due to their flexibility and cost-effectiveness compared to proprietary LLMs.

Distribution over different Control-flow types. Finally, we analyze verification success and failure distributions across different control flow types (see detailed visualizations in Appendix F). From this analysis, we observe that LLMs are primarily capable of generating verifiable specifications for programs with simple control-flow structures, such as sequential or branched programs. However, they often struggle to synthesize formal specifications for programs that contain loops, where the complexity of control flow increases significantly, with a success rate of less than 10% and failure rates of more than 50%. These findings highlight the limitations of LLMs in reasoning about complex control-flow structures, particularly loops, which require more advanced logical and inductive reasoning capabilities.

4.2 RQ_2: Impact of Advanced Prompting Techniques

To answer the RQ_2, we evaluate the top LLMs from RQ_1 (CodeQwen-2.5, DeepSeek-V2, DeepSeek-V3, GPT-4, and Claude 3.5 Sonnet) using two prompting techniques: chain-of-thought (CoT)(Kojima et al., 2022) and least-to-most (LTM)(Zhou et al., 2022). Detailed prompt designs are in Appendix H.

Least-to-Most Prompts. As shown in Table 1, LTM consistently improves the effectiveness of these LLMs, improving both consistency and completeness metrics. For example, the success rate of DeepSeek-V3 increases from 15.5% with few-shot prompts to 16. 6% with LTM (a 7% improvement), while the completeness rate increases from 85. 2% to 89. 6% (a 5% improvement). These improvements are observed not only in proprietary LLMs,

but also in open-source LLMs. Specifically, the success rates of CodeQwen-2.5 and DeepSeek-V2 improve significantly, from 11.4% and 6.9% to 12.0% and 8.9%, respectively. Overall, these findings suggest that LTM prompting, when combined with few-shot demonstrations, should be used to optimize the effectiveness of LLMs in synthesizing program specifications.

Chain-of-Thought Prompts. In contrast, the impact of CoT on LLMs is mixed, with both positive and negative outcomes. For example, CoT improves the success rate of DeepSeek-V3 from 15.5% to 16.2%. However, it has no effect on the success rate of GPT-4o and even decreases the performance of the Claude 3.5 Sonnet. CoT even significantly increases the failure rates of GPT-4o and Claude-3.5-Sonnet from 56.4% and 53.1% to 61.5% and 59.1%. We suspect that this is because CoT relies on the model's ability to self-reason, while LTM provides human-instructed reasoning steps and explicit demonstrations, which guide the models toward better reasoning.

Effectiveness on Complex Control-Flow Programs. While LTM prompting can further improve the performance of LLMs, these improvements are primarily observed in reasoning programs involving branching and sequential control flow. In contrast, the impact of LTM prompting on improving reasoning for programs with complex control flow, such as those containing loops, remains unclear. As a result, the performance of LLMs in loop-containing programs remains low, with success rates of less than 10%. This further underscores the limitations of LLMs in reasoning about programs with loops, which require more advanced inductive reasoning capabilities.

4.3 RQ_3: Robustness of LLMs

To answer the RQ_3, we assess LLM robustness by evaluating their performance on semantically equivalent but syntactically diverse programs using FormalBench-Diverse-N, a subset of 1,794 *natural* program transformations (Le-Cong et al., 2024) from FormalBench-Diverse (see Appendix E). Due to resource constraints, we focus on the top three LLMs: GPT-4, Claude 3.5 Sonnet, and DeepSeek-V3.

Our experimental results, presented in Table 2, reveal significant robustness challenges for all evaluated LLMs. Specifically, we observe flip rates, the proportion of semantically equivalent programs for which the model does not generate verifiable specifications, ranging from 27. 2% to 39. 2%. Among the models, Claude-3.5-Sonnet is the most severely impacted, with a flip rate of 39.2%, indicating that it does not generate verifiable specifications for nearly 40% of the transformations when the original programs had verifiable specifications.

More critically, this lack of robustness leads to a notable decrease in success rates and an increase in failure rates. For instance, the success rate of DeepSeek-V3 drops from 9.3% to 7.8%, a 16% reduction, while GPT-4o and Claude-3.5-Sonnet experience reductions of 9.5% and 17%, respectively. Similarly, failure rates increase by up to 6.6%, further underscoring the sensitivity of LLMs to the syntactic variation created by semantic-preserving transformations.

These findings highlight the limited robustness of LLMs against semantic-preserving transformations, which expose a critical dependence on syntactic patterns rather than underlying semantic properties. This indicates that current LLMs still lack the deep semantic reasoning capabilities necessary to generalize across functionally equivalent but syntactically varied programs.

4.4 RQ_4: Common Failures and Self-Repair Ability of LLMs

To answer the RQ_4, we begin by conducting a semi-automated analysis, as outlined in Appendix G, to categorize the failures of LLMs. For each type of failure, we sample a subset of instances and investigate their root causes. Building on these insights, we design customized prompts that include failure descriptions, additional guidance, and illustrative examples to enable LLMs to self-repair these errors. Additional details on repair prompts are provided in the Appendix I.

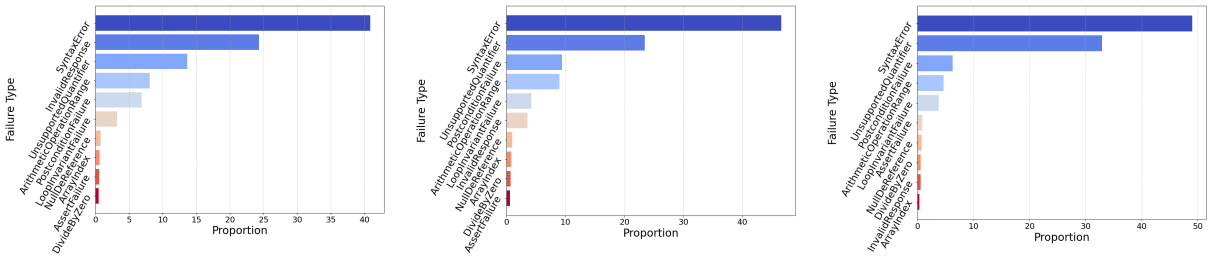
4.4.1 Common Failures

In total, we identified 32 failures of LLMs based on their error messages. Figure 3 illustrates the 10 most common failure categories encountered across llm when using zero-shot, few-shot, and least-to-most (LTM) prompts.

Syntax Errors. Among failure types, "SyntaxError" is the most frequent, and LLMs often generate specifications that violate the JML or Java syntax. This issue persists in most models. A common example is the error message "Unexpected or misspelled JML token," which occurs when an LLM produces incorrect JML grammar. This highlights the challenge of expressing im-

Model Name	FormalBench-Base		FormalBench-Diverse-N		Flip Rate (%)
	SR (%)	FR (%)	SR (%)	FR (%)	
DeepSeek-V3	9.3	62.1	7.8	65.0	27.2
Claude-3.5-Sonnet	10.0	64.1	8.3	64.3	39.20
GPT-4o	11.9	60.1	10.9	64.1	29.2

Table 2: Robustness evaluation of LLMs on FormalBench-Base and FormalBench-Diverse-N benchmarks using different metrics Success Rate (SR), Failure Rate (FR), and Flip Rate.



(a) LLMs with Zero-shot prompts (b) LLMs with Few-shot prompts (c) LLMs with LTM prompts

Figure 3: Top-10 failure category of LLMs with various prompts

plicit program intent in formal languages, a key distinction from natural language specifications such as code comments. More critically, LLMs with zero-shot prompts yield about 25% invalid responses (e.g., producing Javadoc comments instead of JML). Fortunately, this rate drops to 5% with few-shot prompts and further to 1% with LTM prompts.

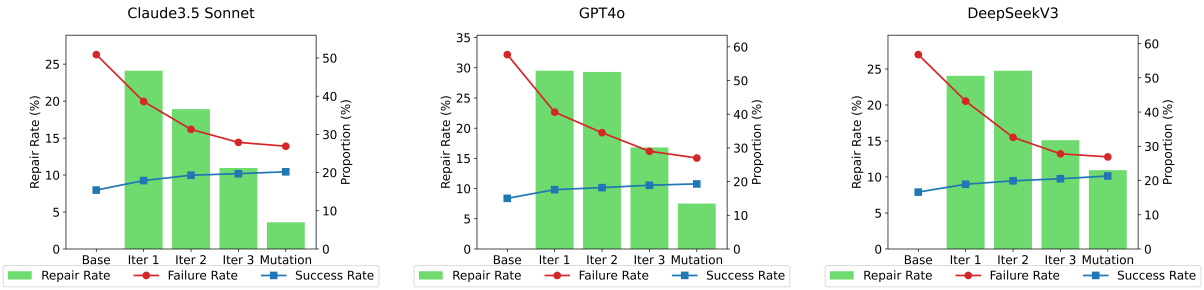
Reasoning Errors. LLMs often encounter reasoning errors, particularly with quantifiers, postconditions, loop invariants, and arithmetic bounds. The most frequent are "UnsupportedQuantifier" errors. In these cases, LLMs rely on inductive quantifiers such as \sum or \product, which are not supported by deductive verification for reasoning about program behaviors. In practice, formal experts must supplement these quantifiers with auxiliary mathematical functions and lemmas to enable inductive reasoning. For improved formal specification synthesis, LLMs must adopt similar human-like strategies rather than relying solely on unsupported quantifiers.

Following "UnsupportedQuantifier", the next most common failure categories, "PostconditionFailure", "LoopInvariantFailure", and "ArithmeticOperationRange", account for nearly 30% of total failures. These errors occur when the verification tool cannot prove postconditions, loop invariants, or arithmetic bounds (e.g., to prevent overflow). Our analysis identifies three main root causes: (1) incorrect specifications, (2) weak or incorrect preconditions that render specifications unprovable, and (3) incomplete reasoning about program behavior, leaving verifiers with insufficient information. These findings underscore the need for LLMs to enhance their reasoning capabilities for more effective formal specification synthesis.

4.4.2 Self-Repair

To evaluate LLM's self-repair ability, we (1) develop a simple failure classifier using pattern matching and (2) design customized prompts with failure descriptions, guidance, and examples (see Appendix H). Additionally, we assess SpecGen's mutation-based repair for verification failures. Due to resource constraints, we evaluate only the top three LLMs: Claude-3.5-Sonnet, GPT-4o, and DeepSeek-V3, and present the results in Figure 4.

The results show that LLMs effectively repair errors using our custom prompts, improving success rates by 25%, from 16% to 20%, and reducing failure rates from over 50% to under 30%. Mutation-based repair further increases success by 0.5 percentage points and reduces failure by 1 to 2 percentage points. Additionally, LLMs can also self-repair across various error categories, such as fixing 53.7% of "SyntaxErrors", 79% of "LoopInvariantFailures", and 65% of "PostconditionFailures" in the first iterations. This pattern persists across subsequent iterations, highlighting the flexibility of self-repair. In contrast, mutation-based repair is limited to specific errors, such as not addressing "ArithmeticOperationRange" errors. Notably, both methods preserve the completeness of



(a) Claude-3.5-Sonnet (b) GPT-4o (c) DeepSeekV3

Figure 4: Effectiveness and self-repair rates of LLMs across iterations: "Iter i" represents self-repair with feedback, while "mutation" represents results from mutation-based repair in the final iteration.

generated specifications, improving the number of verifiable specifications without sacrificing quality. However, we identify two limitations of self-repair approaches. First, self-repair rates decrease with each iteration, leading to saturation in success and failure rates. Second, mutation-based repair is computationally expensive and requires frequent calls to verification tools, so it should be used sparingly, ideally as a final step, to minimize costs.

5 Related Works

Reasoning Evaluation of LLMs. Numerous datasets have been curated to evaluate the reasoning capabilities of LLMs across diverse domains, including mathematical (Cobbe et al., 2021; Hendrycks et al.), logical (Liu et al., 2021; Yang et al., 2022), and causal reasoning (Jin et al., 2024, 2023). Recent work explores code reasoning, evaluating LLMs' ability on program semantic inference. (Hu et al., 2018; Jain et al., 2024; Chen et al., 2025). Early studies focus on code summarization (Husain et al., 2019; Hu et al., 2018), capturing high-level understanding rather than deep semantic reasoning. Recent studies examine code reasoning in detail through output prediction (Jain et al., 2024), execution trace simulation (Chen et al., 2025), and invariant inference (Pei et al., 2023), yet they still address only partial program semantics. In contrast, FormalBench targets formal specification inference, demanding exhaustive reasoning that produces precise, verifiable specifications for every possible execution.

Formal Specification Inference. Traditional dynamic analysis methods, such as Daikon (Ernst et al., 2007), Houdini (Flanagan and Leino, 2001), and DIG (Nguyen et al., 2014), infer likely invariants from observed behaviors using predefined templates. However, these tools often yield trivial invariants (e.g., nums != null) and struggle with complex functional properties (Ma et al., 2024). Recent work leverages LLMs to address these limitations. Early approaches (Pei et al., 2023; Chakraborty et al., 2023) fine-tuned LLMs for invariant inference but focused on specific cases, such as loop invariants or unverified likely invariants. Nilizadeh et al.(Nilizadeh et al., 2021) manually crafted complete program specifications to assess automated repair effectiveness. Building on this, newer methods such as SpecGen(Ma et al., 2024) and AutoSpec (Wen et al., 2024) automatically generate full formal specifications via iterative refinement and static analysis. However, as discussed in Section 1, their evaluations remain limited, highlighting the need for FormalBench and more comprehensive assessments of LLM effectiveness.

6 Conclusion

In this work, we introduce FormalBench, a comprehensive and large-scale benchmark for specification inference. FormalBench integrates robust evaluation metrics to assess the consistency, completeness, and robustness of LLMs on this task. Using FormalBench, we conduct an extensive evaluation of eight popular LLMs, revealing their limited effectiveness and lack of robustness in synthesizing formal specifications, even with advanced prompting techniques. We further analyze their common failure patterns and propose a set of customized prompts, leveraging LLMs' self-repair capabilities to enhance their performance. Overall, FormalBench aims to enable a thorough evaluation and deeper understanding of LLMs in formal program reasoning. The dataset and evaluation infrastructure will be publicly released upon publication.

7 Limitations.

The limitations of this work are as follows:

First, mutation analysis, which sits at the core of our completeness metric, relies on predefined rules to systematically break program behavior, assuming that the generated mutants will exhibit semantic differences from the original program, thereby triggering detectable errors. However, the presence of equivalent mutants, i.e., semantically identical variants of the original program, presents a challenge, as they evade detection, leading to false positives and undermining the accuracy of completeness metrics. To mitigate this issue, we incorporate Equivalent Mutant Suppression (EMS) (Kushigian et al., 2024), a state-of-the-art technique for filtering out equivalent mutants. While EMS reduces their prevalence, some undetected equivalents may still affect the validity of our results. Future research should focus on the development of metrics that more effectively measure the completeness of generated specifications, thereby reducing reliance on mutation analysis as an isolated proxy.

Second, our experiments produced a significant number of "unknown" results from the program verification tools. Manual inspection suggests that these cases are generally associated with higher-quality specifications compared to those with verification failures; yet, they introduce ambiguity due to inherent limitations in deductive verification tools. Future work should prioritize techniques for interpreting or eliminating these ambiguous results, possibly through enhanced symbolic execution or dynamic verification methods.

Finally, our experiments did not include OpenAI-o1 and DeepSeekR1, the latest LLMs at the time of writing. For OpenAI-o1, the associated costs were prohibitively high, so we could not incorporate it into our experiments due to resource constraints. As an alternative, we conducted experiments on o3-mini, the latest reasoning model from OpenAI, with reasonable cost. For DeepSeekR1, access was unavailable at the time of writing due to a security breach affecting the service [1].

[1] https://www.scmp.com/tech/tech-war/article/3296975/deepseek-outage-adds-growing-pains-amid-political-scrutiny-some-see-opportunity

References

James H Andrews, Lionel C Briand, and Yvan Labiche. 2005. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th international conference on Software engineering*, pages 402–411.

Anthropic. 2024. Claude-3.5-sonnet. https://www.anthropic.com/news/claude-3-5-sonnet.

Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, et al. 2022. Multi-lingual evaluation of code generation models. *arXiv preprint arXiv:2210.14868*.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609*.

Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. cvc4. In *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*, pages 171–177. Springer.

Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. 2008. Acsl: Ansi c specification language. *CEA-LIST, Saclay, France, Tech. Rep. v1*, 2.

Saikat Chakraborty, Shuvendu K Lahiri, Sarah Fakhoury, Madanlal Musuvathi, Akash Lal, Aseem Rastogi, Aditya Senthilnathan, Rahul Sharma, and Nikhil Swamy. 2023. Ranking llm-generated loop invariants for program verification. *arXiv preprint arXiv:2310.09342*.

Junkai Chen, Zhiyuan Pan, Xing Hu, Zhenhao Li, Ge Li, and Xin Xia. 2025. Reasoning runtime behavior of a program with llm: How far are we? In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

David R Cok. 2011. Openjml: Jml for java 7 by extending openjdk. In *NASA Formal Methods: Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings 3*, pages 472–479. Springer.

Vijay D'silva, Daniel Kroening, and Georg Weissenbacher. 2008. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178.

Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. 2007. The daikon system for dynamic detection of likely invariants. *Science of computer programming*, 69(1-3):35–45.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

Cormac Flanagan and K Rustan M Leino. 2001. Houdini, an annotation assistant for esc/java. In *International Symposium of Formal Methods Europe*, pages 500–517. Springer.

Cormac Flanagan, K Rustan M Leino, Mark Lillibridge, Greg Nelson, James B Saxe, and Raymie Stata. 2002. Extended static checking for java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*.

Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the naturalness of software. *Communications of the ACM*, 59(5):122–131.

Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33(8):1–79.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th conference on program comprehension*, pages 200–210.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*.

Zhijing Jin, Yuen Chen, Felix Leeb, Luigi Gresele, Ojasv Kamal, Zhiheng Lyu, Kevin Blin, Fernando Gonzalez Adauto, Max Kleiman-Weiner, Mrinmaya Sachan, et al. 2024. Cladder: A benchmark to assess causal reasoning capabilities of language models. *Advances in Neural Information Processing Systems*, 36.

Zhijing Jin, Jiarui Liu, Zhiheng Lyu, Spencer Poff, Mrinmaya Sachan, Rada Mihalcea, Mona Diab, and Bernhard Schölkopf. 2023. Can large language models infer causation from correlation? *arXiv preprint arXiv:2306.05836*.

René Just. 2014. The major mutation framework: Efficient and scalable mutation analysis for java. In *Proceedings of the 2014 international symposium on software testing and analysis*, pages 433–436.

Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2015. Framac: A software analysis perspective. *Formal aspects of computing*, 27(3):573–609.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213.

Benjamin Kushigian, Samuel J Kaufman, Ryan Featherman, Hannah Potter, Ardi Madadi, and René Just. 2024. Equivalent mutants in the wild: Identifying and efficiently suppressing equivalent mutants for java programs. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 654–665.

Axel van Lamsweerde. 2000. Formal specification: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 147–159.

Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W O'Hearn. 2022. Finding real bugs in big programs with incorrectness logic. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA1):1–27.

Thanh Le-Cong, Dat Nguyen, Bach Le, and Toby Murray. 2024. Towards reliable evaluation of neural program repair with natural robustness testing. *Preprint*, arXiv:2402.11892.

Gary T Leavens, Albert L Baker, and Clyde Ruby. 2006. Preliminary design of jml: A behavioral interface specification language for java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38.

Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, et al. 2024a. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434*.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024b. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.

Jian Liu, Leyang Cui, Hanmeng Liu, Dandan Huang, Yile Wang, and Yue Zhang. 2021. Logiqa: a challenge dataset for machine reading comprehension with logical reasoning. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*, pages 3622–3628.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024c. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36.

Yue Liu, Thanh Le-Cong, Ratnadira Widyasari, Chakkrit Tantithamthavorn, Li Li, Xuan-Bach D Le, and David Lo. 2024d. Refining chatgpt-generated code: Characterizing and mitigating code quality issues. *ACM Transactions on Software Engineering and Methodology*, 33(5):1–26.

Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. 2024. Specgen: Automated generation of formal program specifications via large language models. *arXiv preprint arXiv:2401.08807*.

Thanhvu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2014. Dig: A dynamic invariant generator for polynomial and array invariants. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(4):1–30.

Amirfarhad Nilizadeh, Gary T Leavens, Xuan-Bach D Le, Corina S Păsăreanu, and David R Cok. 2021. Exploring true test overfitting in dynamic automated program repair using formal methods. In *2021 14th IEEE conference on software testing, verification and validation (ICST)*, pages 229–240. IEEE.

OpenAI. 2023. Gpt-3.5 turbo. https://openai.com/chatgpt.

OpenAI. 2024. Gpt-4o. https://openai.com/index/hello-gpt-4o/.

Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2023. Can large language models reason about program invariants? In *International Conference on Machine Learning*, pages 27496–27520. PMLR.

Md Rafiqul Islam Rabin, Nghi DQ Bui, Ke Wang, Yijun Yu, Lingxiao Jiang, and Mohammad Amin Alipour. 2021. On the generalizability of neural program models with respect to semantic-preserving program transformations. *Information and Software Technology*, 135:106552.

Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the" naturalness" of buggy code. In *Proceedings of the 38th International Conference on Software Engineering*, pages 428–439.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Cheng Wen, Jialun Cao, Jie Su, Zhiwu Xu, Shengchao Qin, Mengda He, Haokun Li, Shing-Chi Cheung, and Cong Tian. 2024. Enchanting program specification synthesis by large language models using static analysis and program verification. In *International Conference on Computer Aided Verification*, pages 302–328. Springer.

An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jialong Tang, Jialin Wang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Ma, Jin Xu, Jingren Zhou, Jinze Bai, Jinzheng He, Junyang Lin, Kai Dang, Keming Lu, Keqin Chen, Kexin Yang, Mei Li, Mingfeng Xue, Na Ni, Pei Zhang, Peng Wang, Ru Peng, Rui Men, Ruize Gao, Runji Lin, Shijie Wang, Shuai Bai, Sinan Tan, Tianhang Zhu, Tianhao Li, Tianyu Liu, Wenbin Ge, Xiaodong Deng, Xiaohuan Zhou, Xingzhang Ren, Xinyu Zhang, Xipin Wei, Xuancheng Ren, Yang Fan, Yang Yao, Yichang Zhang, Yu Wan, Yunfei Chu, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zhihao Fan. 2024a. Qwen2 technical report. *arXiv preprint arXiv:2407.10671*.

Zhou Yang, Zhensu Sun, Terry Zhuo Yue, Premkumar Devanbu, and David Lo. 2024b. Robustness, security, privacy, explainability, efficiency, and usability of large language models for code. *arXiv preprint arXiv:2403.07506*.

Zonglin Yang, Li Dong, Xinya Du, Hao Cheng, Erik Cambria, Xiaodong Liu, Jianfeng Gao, and Furu Wei. 2022. Language models as inductive reasoners. *arXiv preprint arXiv:2212.10923*.

Weiwei Zhang, Shengjian Guo, Hongyu Zhang, Yulei Sui, Yinxing Xue, and Yun Xu. 2023. Challenging machine learning-based clone detectors via semantic-preserving code transformations. *IEEE Transactions on Software Engineering*, 49(5):3052–3070.

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, et al. 2022. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625*.

A Semantic-preserving Transformations

In this study, we curate a set of 18 semantic-preserving transformations from recent studies (LeCong et al., 2024; Zhang et al., 2023; Rabin et al., 2021) including:

- **VariableRenaming-1** replaces a variable name by its first characters;
- **VariableRenaming-1** replaces a variable name by substitutions derived from CodeBERT (Feng et al., 2020);
- **SwitchRelation** transforms relational expressions by swapping the operands. For example, the expression `a < b` is transformed into `b > a`.
- **Unary2Add** modifies unary operations or increments by converting them into normal assignment statements. For instance, `i++;` is transformed into `i = i + 1;`.
- **Add2Equal** converts add/subtract assignments into equal assignments. For example, `a += 9;` is transformed into `a = a + 9;`, and `b -= 10;` is transformed into `b = b - 10;`.
- **MergeVarDecl** merges multiple variable declarations into a single statement. For instance, `int a;` and `int b;` are merged into `int a, b;`.
- **InfixDividing** divides an in/pre/post-fix expression into two separate expressions, storing intermediate results in a temporary variable. For example, `x = a + b * c` is transformed into `temp = b * c;` followed by `x = a + temp`.
- **SwitchEqualExp** switches the two expressions on both sides of an infix expression where the operator is `=`. For instance, `a == b` is transformed into `b == a`.
- **SwitchStringEqual** switches the order of string equality checks. For example, `a.equals(b)` is transformed into `b.equals(a)`.
- **For2While** transforms a for-loop into a while-loop, restructuring the loop for different control flow requirements.
- **While2For** transforms a while-loop into a for-loop;
- **ElseIf2If** transforms an If...Else if... structure into a nested If...Else structure;
- **Switch2If** transforms a Switch-Case structure into an If-Else structure, converting switch-based logic into a series of conditional checks.
- **SwapStatement** swaps two statements that have no control or data dependency;
- **ReverseIf** switches the code blocks in the if statement and the corresponding else statement, inverting the condition and its associated logic.
- **If2CondExp** changes a single if statement into a conditional expression statement, simplifying the code into a more concise form. For example, `if (condition) { StatementA } else { StatementB }` becomes `condition ? StatementA : StatementB`.
- **CondExp2If** changes a conditional expression statement into a single if statement. For example, `condition ? StatementA : StatementB` becomes `if (condition) { StatementA } else { StatementB }`.
- **DividingComposedIf** divides an if statement with a compound condition $(\wedge, \vee, -)$ into two nested if-statements, breaking down complex conditions into simpler, more manageable parts.

B Evaluation Metrics

In this appendix, we present the formal definition and implementation details of our evaluation metrics, presented in Section 3.2

B.1 Consistency Metrics

Given a benchmark dataset $\mathcal{D}$, an LLM $\mathcal{L}$, and a verification tool $\mathcal{V}$, success rate (SR) and failure rate (FR) are formally defined as follows:

$$SR(\mathcal{L}) = \frac{\left|\{r \in \mathcal{D} \mid g = \mathcal{L}(r) \wedge \mathcal{V}(g, r) = \text{ok}\}\right|}{|\mathcal{D}|},$$

$$FR(\mathcal{L}) = \frac{\left|\{r \in \mathcal{D} \mid g = \mathcal{L}(r) \wedge \mathcal{V}(g, r) = \text{fail}\}\right|}{|\mathcal{D}|},$$

where $g = \mathcal{L}(p)$ is the specification generated by $\mathcal{L}$ for the reference program r, and $\mathcal{V}(g, p)$ denotes the verification result of g on p using $\mathcal{V}$. To

ensure the consistency between the generated specification and the reference program, we employ OpenJML (Cok, 2011), a widely used program verification tool. Specifically, we utilize its latest version (21.0) in the `esc` mode (Extended Static Checker (Flanagan et al., 2002)) with the CVC4 SMT solver (Barrett et al., 2011). Additionally, we enable arithmetic mode and assume that pointers are nullable by default.

B.2 Completeness Metrics

For a generated specification g and a reference program r, the completeness rate (CR) is formally defined as follows:

$$CR(g, r) = \frac{\left|\{p \in \mathcal{P}(r) \mid \mathcal{V}(g, p) \neq \text{ok}\}\right|}{|\mathcal{P}(r)|},$$

where $\mathcal{P}(r)$ is the set of mutants for r, and $\mathcal{V}(g, p)$ is the verification result of g on mutant p. Higher CR indicates greater completeness, as g detects more faults. To generate s$\mathcal{P}(r)$, we utilize Major (Just, 2014), a widely recognized mutation testing framework, using its latest version (3.0.1). To mitigate the generation of equivalent mutants, we further employ EMS (Kushigian et al., 2024), a state-of-the-art equivalent mutant suppression technique.

B.3 Robustness Metrics

To evaluate the robustness of LLMs, we leverage a set of 18 semantic-preserving transformations, presented in Section A. Given p, its set of transformed programs $\mathcal{T}$, LLM $\mathcal{L}$, and verification tool $\mathcal{V}$, with $g = \mathcal{L}(p)$ verified as correct ($\mathcal{V}(g, p) = \text{ok}$), Flip Rate (FlR) is defined as follows:

$$FlR(p, \mathcal{T}) = \frac{\left|\{t \in \mathcal{T} \mid g' = \mathcal{L}(t) \wedge \mathcal{V}(g', t) \neq \text{ok}\}\right|}{|\mathcal{T}|}.$$

Moreover, we also measure the consistency and completeness metrics of LLMs on our transformed dataset. Since transformations may not apply universally, we normalize metrics over applicable transformations. For a reference program r, its set of transformed programs $\mathcal{T}$, and metric $\mathcal{M}$, the normalized metric $\mathcal{M}'$ is defined as follows:

$$\mathcal{M}'(\mathcal{T}) = \frac{\sum_{t \in \mathcal{T}} \mathcal{M}(t)}{|\mathcal{T}|},$$

where $\mathcal{M}$ can be SR, FR, or CR.

C Evaluated Large Language Models

We evaluate the following models with our FormalBench benchmark:

- **Open-source LLMs:**
  - CodeQwen-1.5: CodeQwen-1.5-7B (Bai et al., 2023)
  - CodeQwen-2.5: Qwen2.5-Coder-32B-Instruct (Yang et al., 2024a; Hui et al., 2024)
  - CodeLLama: CodeLlama-34b-Instruct-hf (Roziere et al., 2023)
  - DeepSeek-V2 (Liu et al., 2024a)
- **Proprietary LLMs:**
  - DeepSeek-V3-671B (Liu et al., 2024b)
  - GPT3.5: GPT-3.5-turbo (OpenAI, 2023)
  - GPT-4o (OpenAI, 2024)
  - Claude: Claude-3.5-Sonnet (Anthropic, 2024)

D Experimental Settings

To query LLMs, we implemented our framework using LangChain, an open-source framework designed to streamline the development of applications leveraging llm. For running open-source LLMs, we use an NVIDIA A100 GPU with 80GB of VRAM and an Intel® Xeon® Gold 6326 CPU operating at 2.90 GHz. For running the verification tool, we leverage an Intel® Xeon® Platinum 8358 CPU operating at 2.90 GHz with 28 CPU cores and 1953GB of RAM.

E Constructions of FormalBench-Diverse-Natural

To construct FormalBench-Diverse-Natural, we first assess the naturalness of semantic-preserving transformations by measuring the relative change in cross-entropy, following established methods in previous studies (Le-Cong et al., 2024; Ray et al., 2016; Hindle et al., 2016). We then select 50% of the transformations, sorted by naturalness score. Finally, we choose programs with at least three transformations to avoid bias in the calculation of normalized metrics.

F Distribution of Verification success and failures over different control-flow types.

In this section, we present the distribution of verification success and failures over different control-flow types for three evaluated LLMs.

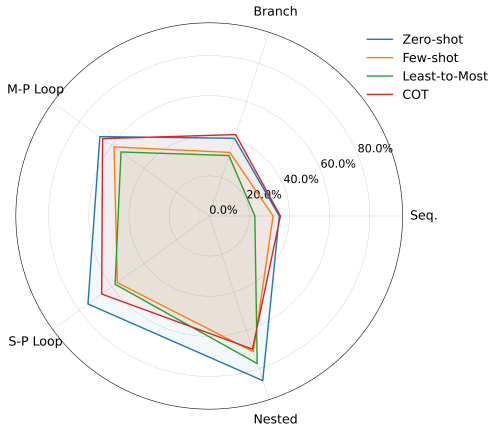


Figure 5: Verification Failures of Claude-3.5-Sonnet

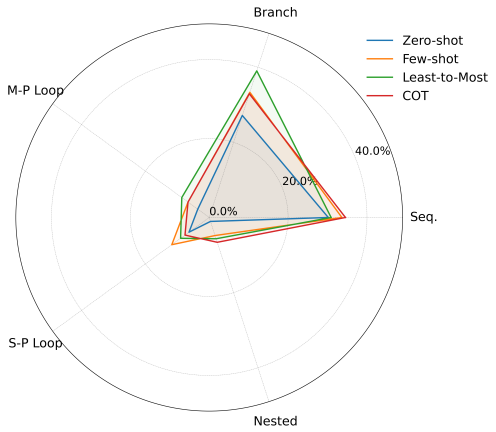


Figure 6: Verification Successes of Claude-3.5-Sonnet

G Failure Analysis

Since each specification can contain failures at multiple locations, our analysis begins by separating these errors into atomic errors. We then conduct a manual analysis of each error to identify common patterns in their error messages. For example, failures related to postconditions consistently include the following string in their messages: "The prover cannot establish an assertion (Postcondition)", as illustrated in Listing 1. Based on these patterns, we build a simple pattern matching to classify failures. This process is repeated until no remaining unknown patterns are found.

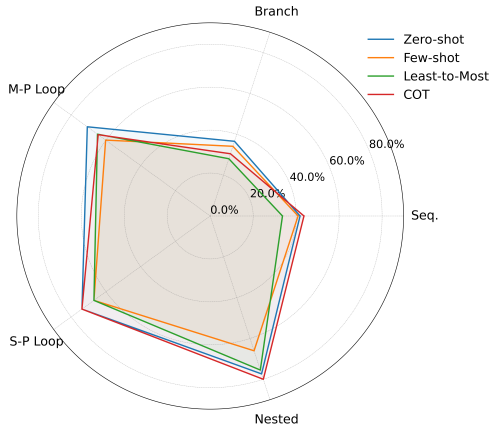


Figure 7: Verification Failures of GPT-4o

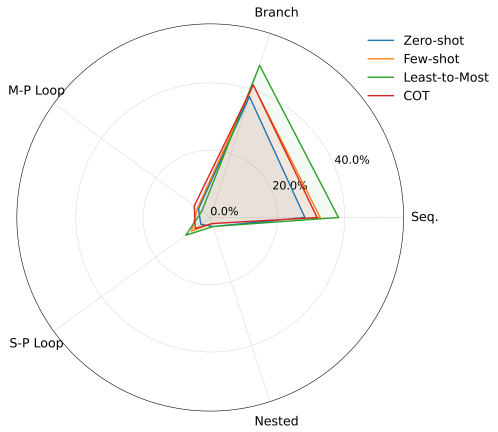


Figure 8: Verification Successes of GPT-4o

H Prompts

In this section, we present prompt templates used in this study for specification generation, including zero-shot, few-shot, chain-of-thought, and least-to-most prompts.

Zero-shot prompt

(System) You are an expert in Java Modeling Language (JML). You will be provided with Java code snippets and their task descriptions. Your task is to generate JML specifications for the given Java code. The specifications should be written as annotations within the Java code and must be compatible with the OpenJML tool for verification. Ensure the specifications include detailed preconditions, postconditions, necessary loop invariants, invariants, assertions, and any relevant assumptions.
(User) Please generate JML specifications for the provided Java code.
CODE
{code}

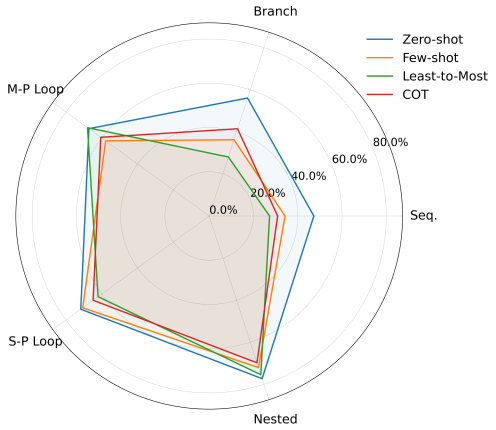


Figure 9: Verification Failures of DeepSeek-V3

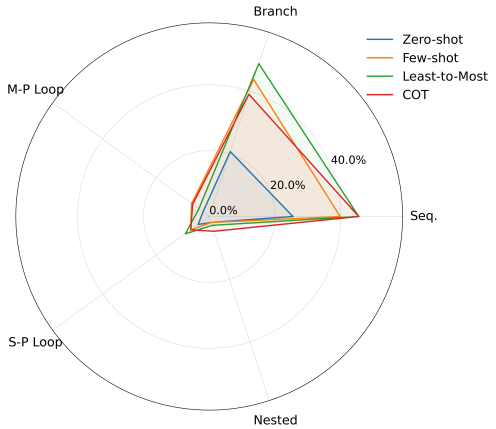


Figure 10: Verification Successes of DeepSeek-V3

Listing 1: A example of error messages for a postcondition failures

```
/tmp/PairOrSum.java:77: verify: The
    prover cannot establish an assertion
     (Postcondition: /tmp/PairOrSum.java
    :69:) in method spec_partialOrSum
        return sum;

/tmp/PairOrSum.java:69: verify:
    Associated declaration: /tmp/
    PairOrSum.java:77:
      @ ensures \result >= 0;
```

Few-shot prompt

(System) You are an expert in Java Modeling Language (JML). You will be provided with Java code snippets. Your task is to generate JML specifications for the given Java code. The specifications should be written as annotations within the Java code and must be compatible with the OpenJML tool for verification. Ensure the specifications include detailed preconditions, postconditions, necessary loop invariants, invariants, assertions, and any relevant assumptions. Please also adhere to the following syntax guidelines for JML:
JML text is written in comments that either:
a) begin with //@ and end with the end of the line, or
b) begin with /*@ and end with */. Lines within such a block comment may have the first non-whitespace characters be a series of @ symbols.
{examples}
(User) Please generate JML specifications for the provided Java code.
CODE
{code}

Chain-of-thought prompt

(System) You are an expert in Java Modeling Language (JML). You will be provided with Java code snippets. Your task is to generate JML specifications for the given Java code. The specifications should be written as annotations within the Java code and must be compatible with the OpenJML tool for verification. Ensure the specifications include detailed preconditions, postconditions, necessary loop invariants, invariants, assertions, and any relevant assumptions. Please also adhere to the following syntax guidelines for JML:
JML text is written in comments that either:
a) begin with //@ and end with the end of the line, or
b) begin with /*@ and end with */. Lines within such a block comment may have the first non-whitespace characters be a series of @ symbols.
{examples}
(User) Please generate JML specifications for the provided Java code.
CODE
{code}
Let's think step by step!

Least-to-Most prompt

(System) You are an expert in Java Modeling Language (JML). You will be provided with Java code snippets. Your task is to generate JML specifications for the given Java code. The specifications should be written as annotations within the Java code and must be compatible with the OpenJML tool for verification. Ensure the specifications include detailed preconditions, postconditions, necessary loop invariants, invariants, assertions, and any relevant assumptions.
Please also adhere to the following syntax guidelines for JML:
JML text is written in comments that either:
a) begin with //@ and end with the end of the line, or
b) begin with /*@ and end with */. Lines within such a block comment may have the first non-whitespace characters be a series of @ symbols.
{examples}
(User) Please generate JML specifications for the provided Java code.
CODE
{code}
Let's break down this problem:
1. What are the weakest preconditions for the code? Be sure to include preconditions related to nullness and arithmetic bounds.
2. What are the strongest postconditions for the code?
3. What necessary specifications are required to prove the above post-conditions? This includes loop invariants, assertions, assumptions, and ranking functions.
After answering these questions, let's generate the specifications for the code and provide solution after '### SPECIFCIATION'

I Self-Repair Prompts

In this section, we illustrate several repair prompts designed to address the most common errors. For a complete list of all repair prompts, please refer to our repository.

Fixing prompt for Syntax Errors

(System) You are an experts on Java Modeling Language (JML). Your task is to fix the JML specifications annotated in the target Java code. You will be provided the error messages from the OpenJML tool and you need to fix the specifications accordingly.
(User) The following Java code is annotated with JML specifications:
{current specification}
OpenJML Verification tool failed to verify the specifications given above, with error information as follows:
ERROR MESSAGE:
{error messages}
ERROR TYPES: Syntax Error
To resolve the syntax error, you should consider the following steps:
1. Identify whether the error is due to a Java syntax issue or a JML syntax issue.
2. Review the code to identify the specific location and nature of the syntax error.
3. Correct the syntax error based on the language rules and conventions.
Please refine the specifications so that they can pass verification. Provide the specifications for the code and include the solution written between triple backticks, after '### FIXED SPECIFICATION'.

Fixing prompt for Unsupported Sum/NumOf/Product Quantifier Expressions

(System) You are an experts on Java Modeling Language (JML). Your task is to fix the JML specifications annotated in the target Java code. You will be provided the error messages from the OpenJML tool and you need to fix the specifications accordingly.
(User) The following Java code is annotated with JML specifications:
{current specification}
OpenJML Verification tool failed to verify the specifications given above, with error information as follows:
ERROR MESSAGE:
{error messages}
ERROR TYPES: Unsupported Sum/NumOf/Product Quantifier Expressions
OpenJML does not fully support JML's inductive quantifiers like \num_of, \sum, and \product in specifications. These operators require inductive reasoning (e.g., counting elements, summing values over a range, or computing products), which is difficult for SMT solvers (the engines behind OpenJML and most of deductive verification tools) to handle.
To avoid the use of \sum, \num_of, and \product quantifiers in your JML specifications, you can express your specifications using induction steps to help OpenJML's verifiers to reason about your code. You can do this by define mathematical functions and lemmas through model methods. For example, you can should not use \product quantifier in the following specifications:
{Examples with reasoning}
Please refine the specifications so that they can pass verification. Provide the specifications for the code and include the solution written between triple backticks, after '### FIXED SPECIFICATION'.

Fixing prompt for Unsupported Min/-Max Quantifier Expressions

(System) You are an experts on Java Modeling Language (JML). Your task is to fix the JML specifications annotated in the target Java code. You will be provided the error messages from the OpenJML tool and you need to fix the specifications accordingly.
(User) The following Java code is annotated with JML specifications:
{current specification}
OpenJML Verification tool failed to verify the specifications given above, with error information as follows:
ERROR MESSAGE:
{error messages}
ERROR TYPES: Unsupported Min/Max Quantifier Expressions
OpenJML does not fully support JML's inductive quantifiers like \min, \max in specifications. These operators require inductive reasonings, which is difficult for SMT solvers (the engines behind OpenJML and most of deductive verification tools) to handle.
To avoid the use of \min and \max quantifiers in your JML specifications, you can use the \forall quantifier to express your specifications. For example, you should not use \max quantifier in the following specifications:
{Examples with reasoning}
Please refine the specifications so that they can pass verification. Provide the specifications for the code and include the solution written between triple backticks, after '### FIXED SPECIFICATION'.

Fixing prompt for Loop Invariant Failures

(System) You are an experts on Java Modeling Language (JML). Your task is to fix the JML specifications annotated in the target Java code. You will be provided the error messages from the OpenJML tool and you need to fix the specifications accordingly.
(User) The following Java code is annotated with JML specifications:
{current specification}
OpenJML Verification tool failed to verify the specifications given above, with error information as follows:
ERROR MESSAGE:
{error messages}
ERROR TYPES: Loop Invariant Failures
This error occurs when the loop invariant, a condition that must hold true before the loop begins and remain true after each iteration, is not properly established or maintained. This semantic error typically arises when verifiers fail to confirm the correctness of the synthesized loop invariant. The causes of this error include: (1) an incorrect loop invariant, (2) wrong/weak preconditions that prevent the invariant from holding at the start of the loop, or (3) incomplete reasoning about the loop, leading to insufficient information for the verifier to verify the invariant.
To resolve the error, please consider the following steps:
1. Carefully review the loop invariant to ensure it correctly captures the necessary conditions that hold true before and after each iteration of the loop.
2. Carefully examine preconditions to ensure they are strong enough to establish the loop invariant at the beginning of the loop.
3. Add additional assertions or assumptions within the loop to help the verifier reason about the loop invariant.
For example, consider the following code snippet with a loop invariant failure:
{Examples with reasoning}
Please refine the specifications so that they can pass verification. Provide the specifications for the code and include the solution written between triple backticks, after '### FIXED SPECIFICATION'.

Fixing prompt for Post-Condition Failures

(System) You are an experts on Java Modeling Language (JML). Your task is to fix the JML specifications annotated in the target Java code. You will be provided the error messages from the OpenJML tool and you need to fix the specifications accordingly.
(User) The following Java code is annotated with JML specifications:
{current specification}
OpenJML Verification tool failed to verify the specifications given above, with error information as follows:
ERROR MESSAGE:
{error messages}
ERROR TYPES: Post-condition Failures
This error occurs when the postcondition, a condition that must hold true after the execution of a program or function, is not satisfied. This type of semantic error typically arises when verifiers are unable to confirm that the program's logic guarantees the postcondition under all valid inputs and scenarios. The causes of this error include: (1) an incorrect or incomplete postcondition, (2) wrong/weak preconditions that prevent the program from reaching a state where the postcondition holds, or (3) incomplete reasoning about the programs, leading to insufficient information for the verifier to verify the postcondition.
To resolve the error, please consider the following steps:
1. Review the postcondition to ensure it correctly captures the expected behavior of the program or function.
2. Check the preconditions to ensure they are strong enough to reach a state where the postcondition holds.
3. Add additional assertions or assumptions within the program or function to help the verifier reason about the postcondition.
For example, consider the following code snippet with a postcondition failure:
{Examples with reasoning}
Please refine the specifications so that they can pass verification. Provide the specifications for the code and include the solution written between triple backticks, after '### FIXED SPECIFICATION'.