SpecRLVR: Quality-Aware Reinforcement Learning with Verifiable Rewards for Formal Specification Generation

Anonymous ACL submission

Abstract

Formal specifications are essential for rigorous software quality assurance, but writing them manually remains time-consuming and error-prone. Although large language models (LLMs) have been used to automate specification generation, existing methods train on reference specifications rather than optimizing for verifiability, which makes the generated specifications often fail verification. Moreover, even verifier-passing specifications can be too weak to accurately describe the target program behavior, failing to provide effective signals for training. To address these limitations, we propose SpecRLVR, a quality-aware framework for JML specification generation. SpecRLVR leverages OpenJML verifier and a teacher model to automatically construct preference data based on a quality hierarchy, enabling the training of a quality-aware model for policy optimization. At inference time, the optimized LLM generates multiple candidate specifications, the quality-aware model scores those that pass the verifier, and the highest-scoring candidate is selected as the final specification. Experimental results on representative benchmarks demonstrate the effectiveness of SpecRLVR. For example, on the SV-COMP benchmark, SpecRLVR achieves 35.40% in Pass@5, outperforming the strongest baseline by 9.27%.

1 Introduction

Formal specifications use languages with precise syntax and semantics to describe the intended behavior of programs, and they provide the basis for rigorous software quality assurance (Meyer, 1992; Leavens et al., 2006; Barnett et al., 2005). They are widely used in downstream tasks such as test generation (Nguyen and Dang, 2023; Molina et al., 2022), model checking (Cabodi et al., 2008), and deductive program verification (Flanagan et al., 2002; Cok, 2011). However, requirements are often written in natural language by domain experts, and translating them into machine-checkable specifications such as JML (Leavens et al., 2006) remains labor-intensive and error-prone. Therefore, automating this translation is a long-standing obstacle to the broader adoption of formal methods in industrial software engineering.

Java Program

```
// returns the maximum element of a
int max(int[] a) {
  int max_ = a[0];
  for (int i = 1; i < a.length; i++) {
    if (a[i] > max_) max_ = a[i];
  }
  return max_;
}
```

Candidate JML Specification A: Verifier-passing but incomplete

```
//@ requires a != null && a.length > 0;
//@ ensures \result >= a[0];
```

Candidate JML Specification B: Verifier-passing and adequate

```
//@ requires a != null && a.length > 0;
//@ ensures (\forall int i; 0 <= i
//@   && i < a.length; \result >= a[i]);
//@ ensures (\exists int i; 0 <= i
//@   && i < a.length; \result == a[i]);
```

Figure 1: Two example specifications of different quality for the same program. Both candidates pass verifier checks, but Candidate A only states a partial property, whereas Candidate B captures the full behavior of the program.

Early work on specification generation mainly focuses on program invariants. Dynamic analysis tools such as Daikon (Ernst et al., 2007) and Houdini (Flanagan and Leino, 2001), together with later learning-based methods (Sharma et al., 2012, 2013; Garg et al., 2014, 2016), infer invariants from manually designed predicate or feature spaces and support only limited invariant forms. Recent work uses large language models (LLMs) (Chen et al., 2021; Rozière et al., 2023; Hui et al., 2024) to alleviate this limitation. AutoSpec (Wen et al., 2024) combines LLM generation with static analysis to synthesize specifications, while SpecGen (Ma et al., 2025) performs iterative repair at inference time.

FormalBench (Le-Cong et al., 2025) further evaluates LLM-based JML specification synthesis with methods such as supervised fine-tuning (SFT) and chain-of-thought (CoT) (Wei et al., 2022).

However, existing methods are typically trained to imitate reference specifications and do not incorporate verifiability into their training objectives, which often causes the generated specifications to fail verification. Moreover, even when the binary (Pass/Fail) verifier feedback is used as the optimization objective, verifier-passing specifications may still be too weak to accurately describe the target program behavior, providing insufficient signals for training. As illustrated in Figure 1, both Candidate A and Candidate B pass OpenJML (Cok, 2011) verification for a method that returns the maximum element of an array. Candidate A only requires `\result >= a[0]` and is satisfied by many functions other than `max`, while Candidate B combines a `\forall` clause and a `\exists` clause to fully characterize the intended behavior. This shows that a fine-grained signal is needed to separate adequate specifications from weak ones.

To address these problems, we propose SpecRLVR, a JML specification generation framework that uses a quality-aware model learned from preference learning to guide both policy optimization and inference-time candidate selection. Specifically, SpecRLVR first categorizes generated specifications into four levels: consistent and complete, consistent but redundant, inconsistent or incomplete, and invalid. SpecRLVR then uses a teacher model and OpenJML to automatically construct preference data, without human annotation. On this preference dataset, we train a Bradley–Terry reward model (Bradley and Terry, 1952) and use it as the reward function in GRPO (Shao et al., 2024) to optimize the specification generator. Finally, at inference time, SpecRLVR samples multiple candidate specifications for each program, discards those that fail OpenJML verification, reranks the remaining candidates with the quality score, and selects the top-ranked candidate as the final specification.

We evaluate SpecRLVR on SV-COMP and FormalBench (Le-Cong et al., 2025), comparing it with representative LLMs and specification generation methods. Experimental results show that our method generates higher-quality specifications than all baselines. For example, on SV-COMP, SpecRLVR solves 149 out of 265 programs and achieves a Pass@5 of 35.40%, exceeding the strongest baseline by 17 programs and 9.27%.

In summary, our contributions are as follows:

- We define a quality hierarchy of specification quality and use OpenJML to automatically construct preference data for training a reward model that guides LLM-based specification generation.
- We propose an inference-time scaling method that filters candidates with OpenJML, ranks the verified specifications by quality, and selects the highest-scoring one as the final specification, improving the performance of the framework.
- Experiments on SV-COMP and FormalBench show that SpecRLVR outperforms representative baselines in both formal verifiability and semantic completeness.

2 Related Work

2.1 Formal Specification Generation

Early work on formal specification generation mainly focuses on program invariants. Dynamic analysis tools such as Daikon (Ernst et al., 2007) and Houdini (Flanagan and Leino, 2001) infer invariants from runtime traces but are limited by predefined predicate spaces, while learning-based methods (Sharma et al., 2012, 2013; Garg et al., 2014, 2016; Liu et al., 2012; Sagdeo et al., 2011) rely on task-specific architectures restricted to certain invariant categories. Recent LLM-based methods broaden this scope. Pei et al. (2023) and Chakraborty et al. (2023) fine-tune or rerank LLM outputs for loop invariants, and AutoSpec (Wen et al., 2024) and SpecGen (Ma et al., 2025) combine LLMs with AST analysis and iterative refinement to produce verifiable specifications for whole programs. FormalBench (Le-Cong et al., 2025) further benchmarks LLMs on specification synthesis under complex control flow and semantics-preserving transformations. However, these works do not incorporate verification feedback into an optimization signal. In this work, we integrate a quality hierarchy of specification preferences into reward model training and leverage RL to improve specification quality.

2.2 RLVR for Large Language Models

In recent years, Reinforcement Learning with Verifiable Rewards (RLVR) has become an effective post-training paradigm for improving the reasoning capabilities of LLMs. In mathematical reasoning, RLVR studies (Shao et al., 2024; DeepSeek-AI et al., 2025; Hu et al., 2025; Yu et al., 2025; Meng

et al., 2026) show that verifiable or execution-style feedback can improve model reasoning. For example, DeepSeekMath (Shao et al., 2024) introduces GRPO with rule-based rewards, while DeepSeek-R1 (DeepSeek-AI et al., 2025) demonstrates that large-scale reinforcement learning can elicit strong reasoning behaviors. RLVR has also been explored in code-related tasks, where program execution provides a natural verifiable signal (Le et al., 2022; Liu et al., 2023; Li et al., 2025; Jiang et al., 2025; Venkatkrishna et al., 2026). CodeRL (Le et al., 2022) uses unit-test feedback to guide code generation, while CodeRL+ (Jiang et al., 2025) introduces execution-semantics alignment to alleviate the sparsity of binary pass/fail rewards. These works show the value of verifiable feedback, but formal specification generation requires a finer reward signal because verifier-passing specifications can still be weak or redundant. SpecRLVR addresses this need by introducing a specification quality hierarchy into reward model training and using the learned reward to guide RL optimization.

3 Methodology

In this section, we introduce SpecRLVR, an RL-based framework for improving the ability of LLMs to generate JML specifications. As shown in Figure 2, SpecRLVR first defines a four-level quality hierarchy that distinguishes ❶ consistent and complete, ❷ consistent but redundant, ❸ inconsistent or incomplete, and ❹ invalid specifications. Based on this hierarchy, SpecRLVR constructs a preference dataset from sampling-based candidates and reference-based variants with OpenJML-guided labeling and validation. The resulting preference data are used to train a reward model that guides generator optimization. During inference, multiple candidate specifications are first filtered by OpenJML and then reranked by the reward model, with the highest-scoring specification selected as the final output.

3.1 Task Definition

Given a Java program P, the task is to generate a JML specification $S = G(P)$ that describes the intended behavior of P and can be checked by OpenJML, where G denotes the specification generator. The output specification may contain *preconditions*, *postconditions*, and *loop invariants*. Following prior work (Le-Cong et al., 2025), a desirable specification should be both consistent and complete. Consistency means that all annotations can be verified by OpenJML for the given program, while completeness means that the annotations jointly characterize the program's intended input-output behavior.

3.2 Specification Quality Hierarchy

OpenJML can validate whether a specification is consistent with the program, but this signal alone cannot measure specification quality. Even if a specification passes verification, it may still fail to fully characterize the program behavior because its constraints are too weak, or it may contain unnecessary annotations and become redundant. Therefore, we divide specification quality into four ordered levels. To illustrate the differences among these levels, we use a method that returns `x+1` as an example and show typical outputs for the same program under different specification quality.

❶ **Consistent and Complete.** The specification is syntactically valid, passes OpenJML verification, and captures the intended behavior without adding unnecessary annotations. It uses the precondition to rule out overflow and the postcondition to specify the exact relation between the input and the return value, so it provides the strongest supervision signal in our hierarchy.

```
//@ requires x < Integer.MAX_VALUE;
//@ ensures \result == x + 1;
```

❷ **Consistent but Redundant.** The specification is still syntactically valid, verifiable, and behaviorally adequate, but it contains clauses already entailed by a more compact specification. Such outputs should not be treated as wrong, because they describe the same behavior. They are nevertheless less desirable because redundancy increases specification length and can add unnecessary proof obligations.

```
//@ requires x < Integer.MAX_VALUE;
//@ ensures \result == x + 1;
//@ ensures \result > x;
//@ ensures \result != x;
```

❸ **Inconsistent or Incomplete.** The specification is syntactically valid but semantically inadequate. This level contains two common failure modes that require different evidence but should both be dispreferred. An *inconsistent specification* states a condition that the implementation cannot satisfy, so OpenJML rejects the program under that specification. An *incomplete specification* may pass verification because it places almost no constraint on the implementation. Both cases are worse than redundancy because they no longer characterize the intended program behavior.

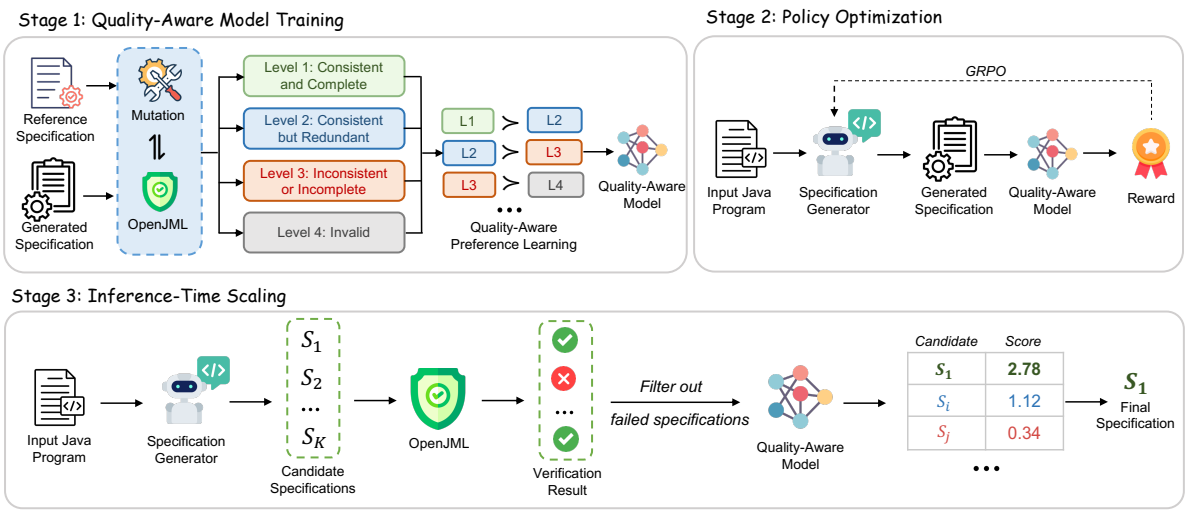

Figure 2: Overall framework of our SpecRLVR.

```
//@ ensures \result == x;   // inconsistent
//@ ensures true;   // incomplete
```

❹ **Invalid.** The output cannot be treated as a well-formed JML specification before semantic verification, because it fails at the syntax or parsing stage or mixes natural language with malformed JML. These samples provide the weakest training signal, since OpenJML cannot reliably interpret them as specifications.

```
//@ requires x < ;
//@ ensures \result == x + ;
//@ the result should be x plus one.
```

3.3 Quality-Aware Preference Data Construction

To construct a preference dataset for quality-aware specification generation, we combine sampling-based labels from OpenJML diagnostics with reference-based mutation that capture redundancy and incompleteness.

Sampling-based Candidates. To obtain natural model-generated errors, we collect invalid and inconsistent specifications through sampling rather than mutation. We first sample candidate specifications from the generator and run OpenJML on each candidate. Candidates that cannot be parsed or type checked as JML are labeled as *invalid*. Candidates that are syntactically valid but fail OpenJML verification are labeled as *inconsistent*. These two categories are taken from real model outputs, because OpenJML diagnostics can directly identify the corresponding failure modes.

Reference-based Mutation. For *incomplete* and *redundant* specifications, verifier feedback alone is insufficient to provide reliable labels. Therefore, we prompt the teacher model to mutate each reference specification into these two types. For *redundant* specifications, we add clauses that are already entailed by the reference specification, such as weaker conditions implied by existing preconditions, result constraints implied by existing postconditions, or loop bounds entailed by stronger invariants. For *incomplete* specifications, we weaken the reference specification by removing conjuncts from preconditions or postconditions, relaxing numeric bounds, replacing exact equalities with weaker inequalities, or removing key preservation conditions from loop invariants. Details about transformation rules are provided in Appendix B.

Variants Validation. Then, we validate redundant and incomplete variants before using them for reward learning. For a redundant variant $S_r = S^\star \cup A_r$, we construct it by adding extra annotations A_r on top of the reference specification $S^\star$. The resulting specification must still pass OpenJML verification. This check ensures that the added annotations are compatible with the original program and do not break consistency.

For an incomplete variant S_i, we first construct S_i with annotation-specific weakening operations, such as relaxing a precondition, deleting a postcondition conjunct, or removing a loop preservation fact. We then generate mutated programs by applying edits to the original Java program, such as changing arithmetic operators, boundary conditions, or return expressions. We keep S_i only when the original program still verifies under S_i, and at least one mutated program also verifies under S_i

but fails to verify under $S^\star$. This result indicates that S_i allows a faulty program variant that the reference specification rejects, so S_i is treated as incomplete rather than as an equivalent reformulation. Candidates whose verification results do not match this criterion are discarded.

3.4 Quality-Aware Model Training and GRPO Optimization

SpecRLVR uses the preference pairs from Section 3.3 to train a reward model and then uses this reward to optimize the specification generator. The reward model takes a program–specification pair (P, S) as input and outputs a scalar score $R_\phi(P, S)$ that reflects the quality hierarchy defined above.

Quality-Aware Model Training. Given the preference pairs constructed in Section 3.3, we write each pair as $S^+ \succ S^-$, where S^+ denotes the preferred specification and S^- denotes the dispreferred one for the same program. The quality-aware model is trained with the Bradley–Terry objective

$$\mathcal{L}_{\mathrm{RM}} = -\mathbb{E}_{(P,S^+,S^-)} \log \sigma\big(R_\phi(P, S^+) - R_\phi(P, S^-)\big)\,.$$

This objective increases the score gap between the preferred and dispreferred specifications. The learned reward can assign lower scores to low-quality specifications and higher scores to adequate and effective specifications.

GRPO Optimization. Starting from a supervised specification generator π_θ, we further optimize it with group relative policy optimization. For each program P, the generator samples a group of K candidate specifications $\{S_i\}_{i=1}^K$. Let $R_i = R_\phi(P, S_i)$ denote the reward of the i-th candidate, and let μ_R and σ_R denote the mean and standard deviation of rewards within the group. The relative advantage is $A_i = (R_i - \mu_R)/\sigma_R$.

Let $r_i(\theta) = \pi_\theta(S_i \mid P)/\pi_{\mathrm{old}}(S_i \mid P)$ denote the probability ratio, and let $\bar{r}_i(\theta) = \mathrm{clip}(r_i(\theta), 1-\epsilon, 1+\epsilon)$. The policy is then updated with a clipped GRPO objective while staying close to the reference policy

$$\begin{aligned}\mathcal{L}_{\mathrm{GRPO}} = &-\mathbb{E}_P \frac{1}{K}\sum_{i=1}^{K} \min(r_i(\theta)A_i, \bar{r}_i(\theta)A_i) \\ &+ \beta D_{\mathrm{KL}}(\pi_\theta \| \pi_{\mathrm{ref}}).\end{aligned}$$

3.5 Inference-Time Scaling

To reduce the instability of single-sample generation, we use inference-time scaling to select a stronger output with the learned quality-aware model. For each input program P, the generator samples a candidate set $\mathcal{C}_N(P) = \{S_i\}_{i=1}^N$ instead of returning a single specification. OpenJML is then used as a hard filter before scoring, which removes syntactically invalid or verification-failing candidates and reduces unnecessary reward-model calls. The quality-aware model scores only the remaining candidates, and the highest-scoring specification is returned

$$\hat{S} = \arg \max_{S \in \mathcal{C}_N(P) \cap \mathcal{V}(P)} R_\phi(P, S),$$

where $\mathcal{V}(P)$ denotes the set of candidates that pass OpenJML checking.

4 Experiment

4.1 Experimental Settings

Benchmark. To evaluate the effectiveness of SpecRLVR, we select 265 class definitions from the SV-COMP benchmark in the Java category and apply preprocessing to ensure that the programs are compilable and terminate normally by replacing competition-specific library calls (e.g., `Verifier.nondetInt()`) with semantically equivalent Java standard library implementations. FormalBench (Le-Cong et al., 2025) is a dataset consisting of 6,219 Java programs paired with corresponding JML specifications, all of which are manually validated to ensure consistency between program behavior and specifications. To ensure diversity, the dataset covers a wide range of control-flow structures, including sequential execution, branching, single-path loops, multi-path loops, and nested loops, and further applies 18 semantic-preserving transformations to evaluate the robustness of large language models to syntactic variations.

Evaluation Metrics. Following prior work (Ma et al., 2025; Le-Cong et al., 2025), we report the Number of Passes and Pass@k based metrics for verifiability and completeness. The Number of Passes, denoted as Num., records the absolute count of programs for which the method produces a successful final specification. Verifiability Pass@k counts a program as passed if the final specification selected from k sampled candidates passes OpenJML verification. Completeness Pass@k uses a stricter criterion and counts a program as passed only if the final specification passes OpenJML verification and mutation-based completeness evalua-

tion. Details of the mutation-based completeness evaluation are provided in Appendix E.

Baselines. Few-shot LLM uses in-context examples for single-turn specification generation. LLM (SFT) fine-tunes the base model on code-specification pairs. LLM + CoT prompts the model to generate intermediate reasoning steps. AutoSpec (Wen et al., 2024) decomposes programs into hierarchical component graphs for modular specification generation. SpecGen (Ma et al., 2025) combines verifier-guided generation with mutation-based repair. We evaluate both open-source models, including LLaMA-3-8B (Grattafiori et al., 2024), Qwen2.5-7B (Bai et al., 2023), Qwen2.5-Coder-8B, and DeepSeek-Coder-6.7B (Guo et al., 2024), and closed-source models, including GPT-5.2, Gemini-3-Flash, and Claude-Sonnet-4.5. Further details on these base models are provided in Appendix C.

Implementation Details. For validation, we use OpenJML (Cok, 2011) to check specifications in the experiment, which is the latest tool for checking the consistency between Java code and JML specifications. Given the intrinsic limitations of OpenJML, we set a 5 minutes timeout to prevent issues like program unresponsiveness. The experiments are conducted on 8 NVIDIA A100 GPUs, each with 40GB of memory. The framework is implemented using PyTorch (Paszke, 2019) and Huggingface Transformers (Wolf et al., 2020), with LLaMA-Factory (Zheng et al., 2024) as the foundation for model customization. For efficient training, we set the learning rate to 5e-5 and the batch size to 16. The model is run for 10 epochs under each setting to ensure the adequacy of the training results. In the inference phase, we set the temperature to 0.8, top-p to 0.95, and the maximum token limit to 1024. More details are shown in Appendix A.

4.2 Experimental Results

Table 1 reports the main results on SV-COMP and FormalBench. We further report the performance of specification generation methods under different base models in Appendix D.

Among open-source base models, SFT generally outperforms few-shot prompting, but the overall Verifiable pass rate and Completeness pass rate remain low, reflecting the limitations of improving models' specification generation ability through SFT and few-shot prompting alone. Closed-source models perform better than most open-source baselines, but still fall behind methods specific to specification generation. Among specification generation methods, AutoSpec and SpecGen substantially outperform general LLM baselines on both datasets. In contrast, SpecRLVR achieves the best overall results. Taking SV-COMP as an example, compared with the strongest baseline, SpecGen, SpecRLVR increases the number of solved programs from 132 to 149, improves Completeness Pass@1 from 20.43% to 29.53%, and improves Completeness Pass@5 from 26.13% to 35.40%.

In conclusion, our proposed SpecRLVR method achieves the highest solved counts, Verifiable pass rate, and Completeness pass rate among all compared methods on both benchmarks.

4.3 Analysis

4.3.1 Ablation Study

Table 2 reports the ablation of SpecRLVR over its four components and four preference-data levels, together with Vanilla RLVR as a reference baseline, which trains the specification generator with GRPO using the binary (Pass/Fail) OpenJML signal as the only reward. More ablation studies are provided in Appendix F. The component ablation includes four variants. w/o RL optimization and w/o inference-time scaling remove the RL stage and the inference-time stage, respectively. w/o Quality-aware model in RL replaces the quality-aware model with the binary OpenJML signal as the reward during policy optimization while keeping the rest of the pipeline. w/o Quality-aware model at inference replaces the quality-aware model with a prompt-based LLM judge that scores verifier-passing candidates. For the preference-data ablation, we remove one of the four levels at a time and upsample the remaining three so that the total number of preference pairs remains unchanged.

The experimental results show that removing any component reduces Completeness Pass@1. Vanilla RLVR reaches only 17.16% on SV-COMP and 15.46% on FormalBench, confirming that the binary verifier signal alone is too coarse for specification generation. The largest drop among the component ablations comes from removing the quality-aware RL stage, where the score decreases to 24.62% on SV-COMP and 22.85% on FormalBench. This indicates that the overall design remains effective even when any single component is removed. For the preference-data ablation, removing the consistent and complete level causes the largest performance drop, reducing Complete-

Method	SV-COMP ($N = 265$)					FormalBench ($N = 6219$)				
	# Num. ↑	Verifiable ↑		Completeness ↑		# Num. ↑	Verifiable ↑		Completeness ↑	
		Pass@1	Pass@5	Pass@1	Pass@5		Pass@1	Pass@5	Pass@1	Pass@5
Open-source models										
LLaMA 3-8B + Few-shot	48	18.43	21.16	7.23	8.47	275	17.32	23.85	6.13	9.04
LLaMA 3-8B + SFT	56	27.83	30.62	14.36	16.71	352	23.86	26.45	11.27	12.83
Qwen 2.5-7B + Few-shot	49	19.16	26.74	7.54	11.32	298	18.94	21.07	7.24	8.36
Qwen 2.5-7B + SFT	62	26.87	32.18	13.42	17.59	339	23.43	28.92	10.65	14.51
Qwen 2.5-Coder-8B + Few-shot	38	16.28	17.93	4.93	5.46	152	14.27	22.81	3.42	6.27
Qwen 2.5-Coder-8B + SFT	65	28.14	34.65	13.95	18.94	392	22.18	24.38	8.86	10.07
DeepSeek-Coder-6.7B + Few-shot	32	17.52	23.86	6.41	9.71	177	15.65	17.42	4.83	5.39
DeepSeek-Coder-6.7B + SFT	56	27.41	29.13	13.27	14.85	325	22.84	28.61	10.14	13.78
Closed-source models										
GPT-5.2 + Few-shot	67	24.86	31.05	12.43	16.21	412	27.16	30.42	13.65	15.18
GPT-5.2 + CoT	69	27.45	34.86	14.52	18.93	418	27.85	31.45	14.23	16.74
Gemini-3-Flash + Few-shot	72	26.24	33.18	13.74	17.85	435	28.63	32.07	14.96	17.46
Gemini-3-Flash + CoT	75	28.92	36.24	16.13	20.43	446	29.42	33.16	15.84	19.12
Claude-Sonnet-4.5 + Few-shot	74	26.93	33.92	14.26	18.31	442	29.14	32.74	15.32	17.92
Claude-Sonnet-4.5 + CoT	77	29.57	36.95	16.65	20.97	453	30.18	33.85	16.38	19.64
Specification generation methods										
AutoSpec	115	32.18	36.45	18.27	21.07	712	30.96	38.62	17.36	22.45
SpecGen	132	34.32	41.78	20.43	26.13	702	32.47	35.91	18.17	20.36
SpecRLVR	**149**	**42.83**	**50.36**	**29.53**	**35.40**	**735**	**37.24**	**44.18**	**25.43**	**28.91**
Gain (Δ)	↑17	↑8.51	↑8.58	↑9.10	↑9.27	↑23	↑4.77	↑5.56	↑7.26	↑6.46

Table 1: Main experimental results on SV-COMP and FormalBench. Best results are **bolded** and second-best results are underlined. The Gain (Δ) row reports the absolute improvement of SpecRLVR over the second-best method in each column.

Variant	SV-COMP	FormalBench
SpecRLVR	**29.53**	**25.43**
Vanilla RLVR	17.16	15.46
Component ablation		
w/o RL optimization	24.62	22.85
w/o inference-time scaling	26.85	24.06
w/o Quality-aware model in RL	25.73	23.42
w/o Quality-aware model at inference	27.91	24.71
Preference-data ablation		
w/o consistent and complete	14.62	13.64
w/o consistent but redundant	27.51	22.24
w/o inconsistent or incomplete	23.85	20.36
w/o invalid	25.12	21.54

Table 2: Ablation study of SpecRLVR on SV-COMP and FormalBench, reported as Completeness Pass@1 (%). The base specification generator is Qwen2.5-7B.

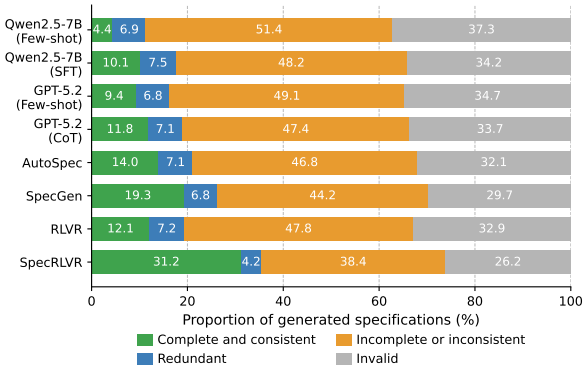

Figure 3: Distribution of generated specifications on SV-COMP. Each bar shows the proportion of complete and consistent, redundant, incomplete or inconsistent, and invalid specifications generated by each method.

ness Pass@1 to 14.62% on SV-COMP and 13.64% on FormalBench, which shows that this level contributes the most among the four levels.

4.3.2 Specification Quality Distribution

Figure 3 shows the quality distribution of generated specifications on SV-COMP. SpecRLVR produces the largest proportion of complete and consistent specifications at 31.20%, well above SpecGen at 19.33% and AutoSpec at 13.97%. Across all three error categories, SpecRLVR also stays below every other method, with redundant at 4.20%, incomplete or inconsistent at 38.40%, and invalid at 26.20%. These results show that SpecRLVR effectively reduces all these types of erroneous specifications and generates more valid specifications.

4.3.3 Quality-Aware Model Accuracy

We evaluate the quality-aware model on a separate evaluation set of candidate specifications labeled with the same four preference levels used during training. We feed each candidate to the reward model and compare the resulting reward score distributions across the four levels. Figure 4 shows a clear score ordering across the four preference levels. Complete and consistent specifications re-

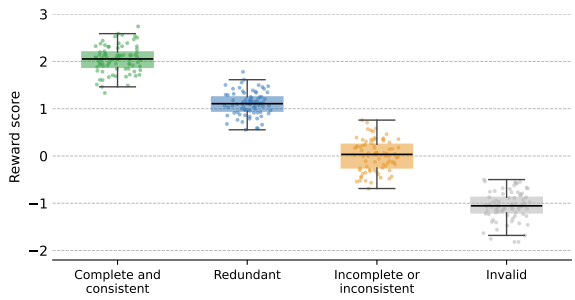

Figure 4: Reward score distributions across four preference levels. Each point corresponds to a held-out candidate specification, and the box summarizes the score distribution within each quality level.

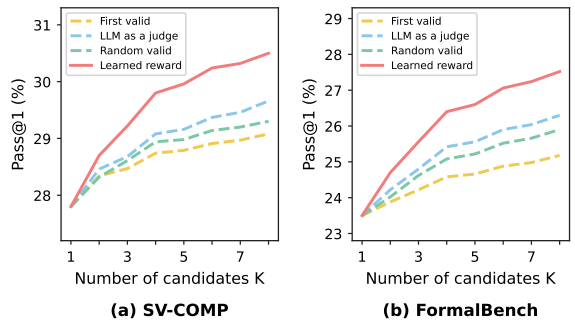

Figure 5: Impact of inference-time scaling on SV-COMP and FormalBench under different candidate selection strategies.

ceive the highest scores, followed by redundant specifications and then incomplete or inconsistent specifications, while invalid specifications receive the lowest scores. The separation between these specifications indicates that the reward model can capture differences among verifier-passing specifications.

4.3.4 Impact of Inference-Time Scaling

We evaluate inference-time scaling by varying the number of sampled candidates from $K = 1$ to $K = 8$. Candidates are first filtered by OpenJML, and the remaining verifier-passing candidates are selected by one of four strategies. First-valid takes the first verifier-passing candidate, Random-valid samples one uniformly, LLM-as-a-judge scores them with a prompted LLM and picks the top one, and Learned-reward picks the candidate with the highest score from quality-aware model.

Figure 5 shows that Pass@1 grows with K on both benchmarks, with most gains before $K = 4$ and diminishing returns afterwards. Learned-reward reranking is the highest across both datasets, indicating that the quality-aware model selects

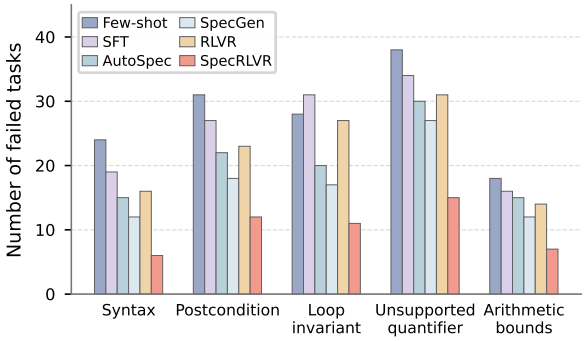

Figure 6: Comparison of failure statistics across major SV-COMP error types between SpecRLVR and the baseline methods.

stronger specifications among verifier-passing candidates and that SpecRLVR benefits from this scoring mechanism.

4.3.5 Failure Type Analysis

Figure 6 compares failed tasks by primary error type. Syntax errors refer to specifications that violate JML syntax. Postcondition failures occur when the generated postconditions cannot prove the intended behavior. Loop invariant failures occur when missing or weak invariants prevent OpenJML from proving loop correctness. Unsupported quantifier failures arise when the model uses quantified expressions that the verifier cannot use for the required reasoning. Arithmetic bound failures indicate missing range constraints needed to prove arithmetic safety.

The analysis shows that SpecRLVR has the fewest failed tasks across all five error types. This result shows that our method reduces the primary errors compared with the baseline methods, rather than shifting failures from one error type to another.

5 Conclusion

In this paper, we present SpecRLVR, an RL framework for JML specification generation. SpecRLVR constructs quality-aware preference data with the teacher model and OpenJML, trains a quality-aware model from the resulting hierarchy, and uses the model for optimization and candidate selection. Experiments on SV-COMP and FormalBench show that SpecRLVR improves over baselines.

In future work, we will extend SpecRLVR to more specification languages and larger program repositories. We will also improve the quality hierarchy with finer semantic checks and study more efficient inference-time selection methods.

Limitations

SpecRLVR relies on inference-time candidate selection to obtain its final output. For each program, the model samples multiple candidate specifications, filters them with OpenJML, and reranks the remaining candidates with the quality-aware model. This design improves the stability of the selected specification, but it also means that the final result depends on additional test-time computation.

In addition, in the data construction and experimental design of this work, we prioritized coverage of the three most general and dominant specification elements, namely preconditions, postconditions, and loop invariants. For some clause types that depend on specific semantics or tool-specific features, we did not conduct dedicated extensions or separate evaluations at this stage.

References

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, and 1 others. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609*.

Mike Barnett, K Rustan M Leino, and Wolfram Schulte. 2005. The Spec# programming system: An overview. In *International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 49–69. Springer.

Ralph Allan Bradley and Milton E Terry. 1952. Rank analysis of incomplete block designs: I. the method of paired comparisons. *Biometrika*, 39(3/4):324–345.

Gianpiero Cabodi, Sergio Nocco, and Stefano Quer. 2008. Strengthening model checking techniques with inductive invariants. *IEEE transactions on computer-aided design of integrated circuits and systems*, 28(1):154–158.

Saikat Chakraborty, Shuvendu Lahiri, Sarah Fakhoury, Akash Lal, Madanlal Musuvathi, Aseem Rastogi, Aditya Senthilnathan, Rahul Sharma, and Nikhil Swamy. 2023. Ranking llm-generated loop invariants for program verification. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 9164–9175.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

David R Cok. 2011. Openjml: Jml for java 7 by extending openjdk. In *NASA formal methods symposium*, pages 472–479. Springer.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, and 1 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

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.

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.

Pranav Garg, Christof Löding, Parthasarathy Madhusudan, and Daniel Neider. 2014. Ice: A robust framework for learning invariants. In *International Conference on Computer Aided Verification*, pages 69–87. Springer.

Pranav Garg, Daniel Neider, Parthasarathy Madhusudan, and Dan Roth. 2016. Learning invariants using decision trees and implication counterexamples. *ACM Sigplan Notices*, 51(1):499–512.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, and 1 others. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yifan Wu, YK Li, and 1 others. 2024. Deepseek-coder: when the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Jingcheng Hu, Yinmin Zhang, Qi Han, Daxin Jiang, Xiangyu Zhang, and Heung-Yeung Shum. 2025. Open-reasoner-zero: An open source approach to scaling up reinforcement learning on the base model. *arXiv preprint arXiv:2503.24290*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, and 1 others. 2024. Qwen2.5-Coder technical report. *arXiv preprint arXiv:2409.12186*.

Xue Jiang, Yihong Dong, Mengyang Liu, Hongyi Deng, Tian Wang, Yongding Tao, Rongyu Cao, Binhua Li, Zhi Jin, Wenpin Jiao, Fei Huang, Yongbin Li, and Ge Li. 2025. CodeRL+: Improving code generation via reinforcement with execution semantics alignment. *arXiv preprint arXiv:2510.18471*.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. In *Advances in Neural Information Processing Systems*.

Thanh Le-Cong, Bach Le, and Toby Murray. 2025. Can LLMs reason about program semantics? a comprehensive evaluation of LLMs on formal specification inference. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 21991–22014. Association for Computational Linguistics.

Gary T Leavens, Albert L Baker, and Clyde Ruby. 2006. Preliminary design of JML: A behavioral interface specification language for Java. In *ACM SIGSOFT Software Engineering Notes*, volume 31, pages 1–38.

Qingyao Li, Xinyi Dai, Xiangyang Li, Weinan Zhang, Yasheng Wang, Ruiming Tang, and Yong Yu. 2025. Codeprm: Execution feedback-enhanced process reward model for code generation. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 8169–8182.

Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81.

Jiate Liu, Yiqin Zhu, Kaiwen Xiao, Qiang Fu, Xiao Han, Wei Yang, and Deheng Ye. 2023. RLTF: Reinforcement learning from unit test feedback. *Transactions on Machine Learning Research*.

Lingyi Liu, David Sheridan, William Tuohy, and Shobha Vasudevan. 2012. A technique for test coverage closure using goldmine. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(5):790–803.

Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. 2025. Specgen: Automated generation of formal program specifications via large language models. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 16–28. IEEE.

Haoming Meng, Kexin Huang, Shaohang Wei, Chiyu Ma, Shuo Yang, Xue Wang, Guoyin Wang, Bolin Ding, and Jingren Zhou. 2026. Sparse but critical: A token-level analysis of distributional shifts in rlvr fine-tuning of llms. In *The Fourteenth International Conference on Learning Representations*.

Bertrand Meyer. 1992. Applying design by contract. In *Computer*, volume 25, pages 40–51.

Facundo Molina, Marcelo d'Amorim, and Nazareno Aguirre. 2022. Fuzzing class specifications. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1008–1020.

Thi-Hanh Nguyen and Duc-Hanh Dang. 2023. Tc4mt: A specification-driven testing framework for model transformations. *International Journal of Software Engineering and Knowledge Engineering*, 33(06):953–991.

A Paszke. 2019. Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703*.

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.

Baptiste Rozière, Jonas Gehre, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, and 1 others. 2023. Code Llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Parth Sagdeo, Viraj Athavale, Sumant Kowshik, and Shobha Vasudevan. 2011. Precis: Inferring invariants using program path guided clustering. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 532–535. IEEE.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, and 1 others. 2024. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V Nori. 2013. A data driven approach for algebraic loop invariants. In *European Symposium on Programming*, pages 574–592. Springer.

Rahul Sharma, Aditya V Nori, and Alex Aiken. 2012. Interpolants as classifiers. In *International Conference on Computer Aided Verification*, pages 71–87. Springer.

Vatsal Venkatkrishna, Indraneil Paul, and Iryna Gurevych. 2026. Aletheia: What makes rlvr for code verifiers tick? *arXiv preprint arXiv:2601.12186*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

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.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, and 1 others. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*, pages 38–45.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, and 1 others. 2025. Dapo: An open-source llm reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*.

Yaowei Zheng, Richong Zhang, Junhao Zhang, YeYanhan YeYanhan, and Zheyan Luo. 2024. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pages 400–410.

A Implementation Details

SpecRLVR and Vanilla RLVR are both initialized from Qwen 2.5-7B-Instruct and share the same backbone, so that the gap between them reflects only the reward signal and the sampling strategy rather than model capacity; for SpecRLVR, the teacher model used to construct the preference data is GPT-5.2. For SpecGen and AutoSpec, we replace their underlying LLM call with Claude-Sonnet-4.5, the strongest closed-source model in our main table, so that the comparison is not confounded by a weaker backbone on the baseline side; results on the remaining base models are reported in Appendix D, and the full list of base models is summarized in Appendix C.

Because the available data in the original benchmarks is limited, we apply a data synthesis pipeline on top of SV-COMP and FormalBench to ensure sufficient training and a fair comparison. Starting from the programs and reference specifications in these two benchmarks, we prompt the teacher model to generate variant Java programs and corresponding JML specifications by paraphrasing identifiers, restructuring control flow, and recombining method bodies, so that the synthesized samples remain in the same distribution while differing in surface form from the seeds.

To prevent leakage from the evaluation set into training, we then apply a rule-based filter that computes ROUGE overlap between each synthesized program and every program in the two benchmarks, as well as between each synthesized specification and every reference specification, and we discard any synthesized (code, specification) pair whose ROUGE-L score with any benchmark item exceeds 0.5 on either the code side or the specification side. Each retained candidate is further required to pass OpenJML consistency checking and the mutation-based completeness checking described in Appendix E before being added to training. This yields 5,000 code-specification pairs used for SFT and 5,000 preference pairs used for training the quality-aware model, none of which exceed the ROUGE-L (Lin, 2004) threshold with any program or specification in the two evaluation benchmarks.

On top of this common training setup, GRPO samples 8 rollouts per prompt, uses a KL coefficient of 0.04, and normalizes the advantage within each group; Vanilla RLVR and SpecRLVR share these hyperparameters and differ only in the reward function, where Vanilla RLVR uses a binary verifier reward and SpecRLVR uses the quality-aware reward introduced in Section 3. For few-shot prompting, we provide three pairs of code-specification demonstrations for the LLMs. For all methods reported in the main table, the inference-time sample size is fixed to $K = 4$, and the effect of varying K is reported in Section 4.3.4.

B Mutation Details

This appendix details how we construct redundant and incomplete variants from a reference specification $S^\star$. We decompose $S^\star$ into three annotation categories, namely `requires` clauses, `ensures` clauses, and loop invariants, and apply category-specific transformations. Each variant differs from $S^\star$ by exactly one transformation so that its quality label can be attributed to a single edit.

Redundant Transformations. A redundant variant must remain consistent and complete with respect to the implementation. It may add clauses, but these clauses should not contribute new semantic constraints.

For `requires` clauses, we use three rules. *Tautology injection* adds a clause that is always true under the current precondition. One example is restating a Java type guarantee such as `requires arr != null` when this is already guaranteed by the calling context. *Implication weakening* adds a clause that is implied by an existing one. One example is adding `requires x >= 0` when `requires x > 0` already exists. *Disjunctive expansion* replaces a clause P with $P \vee P'$, where P' is any predicate over the same variables.

For `ensures` clauses, we use three analogous rules. *Implied consequence* adds a postcondition that is derivable from the current postconditions. One example is adding `ensures \result > \old(x)` when `ensures \result == \old(x) + 1` is already present. *Equivalent re-expression* duplicates a postcondition in an algebraically equivalent form, such as pairing `\result == x + 1` with `\result - 1 == x`. *Frame restatement* appends `assignable \nothing` to a method whose body already has no side effects.

For loop invariants, we use two rules. *Bound widening* duplicates a tight bound with a weaker bound, such as adding `loop_invariant i >= 0` when `loop_invariant 0 <= i && i <= n` is already present. *Tautological invariant* injects an invariant that holds trivially across iterations, such as `loop_invariant \old(n) == n` in a loop that

does not modify `n`.

Every redundant candidate is checked by OpenJML on the original implementation. We also check that each added clause is entailed by $S^\star$ in assumption mode, where $S^\star$ is assumed and the new clause is proved as an assertion. Any candidate that fails either check is discarded.

Incomplete Transformations. An incomplete variant must still pass OpenJML on the original implementation. At the same time, it must lose enough constraint strength so that plausible buggy variants are no longer ruled out.

For `requires` clauses, we use three rules. *Conjunct removal* drops one conjunct from a multi-conjunct precondition. *Range relaxation* widens a numeric bound, such as replacing `requires 0 < x && x < 100` with `requires x >= 0`. *Vacuous precondition* replaces the whole precondition block with `requires true`.

For `ensures` clauses, we use four rules. *Conjunct removal* drops one postcondition conjunct. *Equality relaxation* replaces an equality with a weaker inequality, such as replacing `\result == x + 1` with `\result >= x`. *Quantifier weakening* replaces a universally quantified clause with an existentially quantified clause over the same predicate. *Vacuous postcondition* removes the `ensures` block or replaces it with `ensures true`.

For loop invariants, we use three rules. *Invariant removal* deletes one invariant required in the original loop. *Bound omission* removes the upper bound on the loop counter while keeping the lower bound. *Tautologization* replaces an invariant with `loop_invariant true`.

Every incomplete candidate must pass OpenJML on the original implementation. We then require it to fail on at least one mutant from the completeness mutation suite in Appendix E, while $S^\star$ rejects that same mutant. Candidates that cannot be distinguished from $S^\star$ by any mutant are discarded.

Variant Sampling and Balancing. For each reference $S^\star$ and each transformation rule, we generate up to three variants by varying which clause is targeted. We then sample uniformly across rules so that no single rule dominates the preference set, and we deduplicate variants by canonicalizing clause order and whitespace. The resulting pool is partitioned into redundant and incomplete subsets, which together with the sampled invalid and inconsistent specifications populate the four levels of the quality hierarchy used to train the reward model.

C Base Model Details

This section summarizes the base models that appear in the main experiments. SpecRLVR and Vanilla RLVR are initialized from Qwen 2.5-7B-Instruct. The remaining open-source models are evaluated under the Few-shot and SFT protocols, and the closed-source models are evaluated under the Few-shot and CoT.

Open-source models. **LLaMA 3-8B** is a general-purpose dense decoder-only model trained on a multilingual web corpus. It is included as a general-purpose baseline that has not been specialized for code, so its performance reflects how a strong general model behaves on JML specification generation without code-specific pretraining. **Qwen 2.5-7B-Instruct** is the instruction-tuned variant of the Qwen 2.5 dense model family. It also serves as the initialization for SpecRLVR and Vanilla RLVR, so its Few-shot and SFT rows in the main table characterize the starting point that our RL methods build on. **Qwen 2.5-Coder-8B** is the code-specialized member of the Qwen 2.5 family, further pretrained on a large source-code corpus. It represents an open code-pretrained model of comparable scale to Qwen 2.5-7B-Instruct, which lets us measure how much code pretraining alone contributes before any RL. **DeepSeek-Coder-6.7B** is a code-focused dense model trained on a large multilingual code corpus with repository-level context. It is included as an independent code-specialized open model from a different family, so the gap between it and CodeQwen 2.5-8B reflects cross-family variance under the same protocol.

Closed-source models. **GPT-5.2** is OpenAI's strongest general-purpose reasoning model available at the time of submission. We use the chat completion endpoint with the default reasoning settings. **Gemini-3-Flash** is Google's fast multimodal model with explicit code reasoning support. It is included to cover a high-throughput closed-source baseline that is widely used in code-related tasks. **Claude-Sonnet-4.5** is Anthropic's strongest mid-tier model at the time of submission and the strongest closed-source model in our main table. We also use it as the underlying LLM for SpecGen and AutoSpec so that those baselines are not handicapped by a weaker backbone.

D Impact of Different Base Models on Performance

To examine how the choice of base model affects performance, we apply Vanilla RLVR and SpecRLVR to four open-source base models, and apply AutoSpec and SpecGen to three closed-source base models. We report all metrics used in the main table, namely the Number of Passes (Num.), Verifiability Pass@1, Verifiability Pass@5, Completeness Pass@1, and Completeness Pass@5. The results are summarized in Table 3.

For the open-source models, SpecRLVR delivers consistent gains across all four backbones and across all five metrics, and the ranking of backbones is stable under both Verifiability and Completeness. The improvement holds for both general-purpose models such as LLaMA-3-8B and code-specialized models such as Qwen2.5-Coder-8B and DeepSeek-Coder-6.7B, which indicates that the quality-aware reward and inference-time selection are not tied to a specific backbone family. Qwen2.5-7B remains the strongest backbone under SpecRLVR and is therefore used as the default in the main experiments.

For the closed-source models, AutoSpec and SpecGen scale smoothly with the underlying model and both reach their best performance with Claude-Sonnet-4.5 across all five metrics, which motivates the choice of Claude-Sonnet-4.5 as the default backbone for these baselines in the main experiments. Even when paired with their strongest backbone, AutoSpec and SpecGen still trail SpecRLVR with Qwen2.5-7B on every metric, which shows that the gain of SpecRLVR comes from the training and selection strategy rather than from a stronger base model.

E Mutation-Based Completeness Evaluation Method

A specification may pass OpenJML verification while still placing almost no constraint on the program, for example `ensures true` verifies on every implementation. To measure how tightly a generated specification g characterizes a reference program r, we evaluate g on a set of mutated versions of r and ask whether g still verifies. A tight specification should fail to verify once the program behavior is altered, while a vacuous specification continues to verify on the mutants.

Following prior work (Le-Cong et al., 2025), given a reference program r and a generated specification g that has been verified by OpenJML on r, let $\mathcal{M}(r)$ denote the set of mutants generated from r. The completeness rate is defined as

$$\mathrm{CR}(g, r) \;=\; \frac{\big|\{m \in \mathcal{M}(r) \mid \mathcal{V}(g, m) \neq \mathrm{ok}\}\big|}{|\mathcal{M}(r)|},$$

where $\mathcal{V}(g, m)$ is the OpenJML verification result of g on mutant m. A higher CR means g rejects more altered versions of r and therefore characterizes the intended behavior more tightly.

We generate $\mathcal{M}(r)$ with the Major mutation framework, which produces small semantic perturbations such as replacing arithmetic and relational operators, flipping logical connectors, shifting conditional boundaries, deleting statements, mutating constants, and altering return values. Mutants that fail to compile are discarded. To avoid penalizing g on mutants that are behaviorally identical to r, we filter $\mathcal{M}(r)$ with the EMS equivalent-mutant suppression tool and keep only the surviving non-equivalent mutants for the computation of CR.

The Completeness Pass@k metric requires that g both pass OpenJML verification on r and reject every surviving mutant in $\mathcal{M}(r)$, that is $\mathrm{CR}(g, r) = 1$. OpenJML is invoked in ESC mode with the CVC4 SMT solver and the 5 minute timeout used in the main experiments.

F Additional Ablation Study

To complement the Completeness Pass@1 ablation in the main text, we report the same ablation under Completeness Pass@5 in Table 4. Following the main ablation table, we include Vanilla RLVR as a reference baseline, and the component variants and preference-data variants follow the definitions used in the main text.

The Pass@5 results follow the same trend as Pass@1. Vanilla RLVR reaches only 19.28% on SV-COMP and 20.13% on FormalBench, which again shows that the binary verifier signal alone is too coarse to distinguish specifications of different quality even when more candidates are sampled. Removing the quality-aware RL stage causes the largest component-level drop, reducing Pass@5 from 35.40% to 30.82% on SV-COMP and from 28.91% to 26.58% on FormalBench, which shows that quality-aware optimization remains the dominant contributor at the larger evaluation budget. For the preference-data ablation, removing the consistent and complete level remains the most damaging, reducing Pass@5 to 17.46% on SV-COMP and

Method	Base model	SV-COMP ($N = 265$)					FormalBench ($N = 6219$)				
		# Num. ↑	Verifiable ↑		Completeness ↑		# Num. ↑	Verifiable ↑		Completeness ↑	
			Pass@1	Pass@5	Pass@1	Pass@5		Pass@1	Pass@5	Pass@1	Pass@5
Open-source base models											
SpecRLVR	LLaMA-3-8B	138	39.62	47.18	26.74	32.16	692	33.96	40.27	22.83	26.14
	Qwen2.5-7B	**149**	**42.83**	**50.36**	**29.53**	**35.40**	**735**	**37.24**	**44.18**	**25.43**	**28.91**
	Qwen2.5-Coder-8B	144	41.27	48.95	28.16	33.84	668	32.85	39.42	21.74	25.06
	DeepSeek-Coder-6.7B	141	40.53	48.07	27.42	32.91	678	33.18	40.05	22.36	25.73
Closed-source base models											
AutoSpec	GPT-5.2	102	29.43	33.84	16.32	18.92	652	28.74	36.18	15.47	20.13
	Gemini-3-Flash	109	31.05	35.16	17.54	20.13	685	29.83	37.45	16.28	21.27
	Claude-Sonnet-4.5	**115**	**32.18**	**36.45**	**18.27**	**21.07**	**712**	**30.96**	**38.62**	**17.36**	**22.45**
SpecGen	GPT-5.2	118	31.46	38.74	18.16	23.85	642	30.16	33.42	16.34	18.74
	Gemini-3-Flash	125	32.84	40.27	19.27	24.96	672	31.24	34.78	17.42	19.53
	Claude-Sonnet-4.5	**132**	**34.32**	**41.78**	**20.43**	**26.13**	**702**	**32.47**	**35.91**	**18.17**	**20.36**

Table 3: Impact of different base models on SV-COMP and FormalBench. SpecRLVR is trained on four open-source base models, and AutoSpec and SpecGen are evaluated with three closed-source base models. The default base model used for each method in the main experiments is shown in bold.

Variant	SV-COMP	FormalBench
SpecRLVR	**35.40**	**28.91**
Vanilla RLVR	19.28	20.13
Component ablation		
w/o RL optimization	30.82	26.58
w/o inference-time scaling	32.16	27.34
w/o Quality-aware model in RL	31.47	26.93
w/o Quality-aware model at inference	33.54	28.12
Preference-data ablation		
w/o consistent and complete	17.46	15.52
w/o consistent but redundant	32.97	25.27
w/o inconsistent or incomplete	28.63	23.14
w/o invalid	30.15	24.48

Table 4: Ablation study of SpecRLVR on SV-COMP and FormalBench, reported as Completeness Pass@5 (%). The base specification generator is Qwen2.5-7B.

15.52% on FormalBench. This indicates that the contribution of each component and each preference level is stable across both evaluation budgets.

G Prompts

This section lists the prompts used in our experiments. Figure 7 shows the Few-shot specification generation prompt used by all open-source and closed-source models under the Few-shot protocol, as well as by SpecRLVR at training (rollout) and inference time. Figure 8 shows the Chain-of-Thought (CoT) prompt used by the closed-source models. Figure 9 shows the LLM-as-a-judge prompt used in the inference-time selection study in Section 4.3.4 and in the *w/o Quality-aware model at inference* ablation. The placeholders in braces are filled in at runtime, and the in-context examples are sampled from the synthesized training pool described in Appendix A so that no example overlaps with the evaluation benchmarks.

Few-shot specification generation prompt

```
[System]
You are an expert in the Java Modeling Language (JML). You will be given Java code snippets,
and your task is to produce JML specifications for them. The specifications should appear as
annotations inside the Java code and should be accepted by the OpenJML tool for verification.
Make the specifications informative by including the relevant preconditions, postconditions, loop
invariants, class invariants, assertions, and any assumptions that the code relies on.

Follow the JML syntax conventions below. JML text lives inside Java comments that either (a) start
with //@ and run to the end of the line, or (b) start with /*@ and end with */. Within a block
comment, the first non-whitespace characters of each line may be a sequence of @ symbols.

{examples}

[User]
Please generate JML specifications for the provided Java code.

### CODE
{code}
```

Figure 7: Few-shot specification generation prompt. Used by every model in the Few-shot rows of Table 1, and shared by SpecRLVR for SFT, GRPO rollouts, and inference-time sampling. The placeholder `{examples}` is replaced by N in-context (code, specification) pairs, and `{code}` is replaced by the target Java program.

Chain-of-Thought specification generation prompt

```
[System]
You are an expert in the Java Modeling Language (JML). You will be given Java code snippets,
and your task is to produce JML specifications for them. The specifications should appear as
annotations inside the Java code and should be accepted by the OpenJML tool for verification.
Make the specifications informative by including the relevant preconditions, postconditions, loop
invariants, class invariants, assertions, and any assumptions that the code relies on.

Follow the JML syntax conventions below. JML text lives inside Java comments that either (a) start
with //@ and run to the end of the line, or (b) start with /*@ and end with */. Within a block
comment, the first non-whitespace characters of each line may be a sequence of @ symbols.

{examples}

[User]
Please generate JML specifications for the provided Java code.

### CODE
{code}

Let's think step by step!
```

Figure 8: Chain-of-Thought (CoT) specification generation prompt. Used by the closed-source models in the CoT rows of Table 1.

LLM-as-a-judge reranking prompt

You are evaluating candidate JML specifications for the same Java program. Each candidate has already been verified by OpenJML, so all of them are consistent with the implementation. Your task is to pick the single candidate that most tightly characterizes the intended behavior of the program.

Use the following criteria, listed in decreasing order of importance.

- **Completeness**. Prefer specifications whose ensures clauses pin down the return value and the post-state for every control-flow path, and whose loop invariants are strong enough to prove the postcondition.
- **Non-redundancy**. Penalize specifications that contain tautological clauses, clauses logically entailed by other clauses, or restated frame conditions on side-effect-free code.
- **Precision**. Prefer tight numeric bounds, equalities over inequalities, and universally quantified clauses over weaker existential ones, when both are consistent with the implementation.

Java code: “‘java {target_code} “‘

Candidate 1: “‘java {candidate_1} “‘

. . .

Candidate {K}: “‘java {candidate_K} “‘

Return your answer in the following JSON format and nothing else. “‘json {{"best": <integer index from 1 to {K}>, "reason": "<one-sentence justification>"}} “‘

Figure 9: LLM-as-a-judge reranking prompt. Used by the *LLM-as-a-judge* strategy in Section 4.3.4 and by the *w/o Quality-aware model at inference* variant in the ablation study.