SPECMIND: Cognitively Inspired, Interactive Multi-Turn Framework for Postcondition Inference

Anonymous ACL submission

Abstract

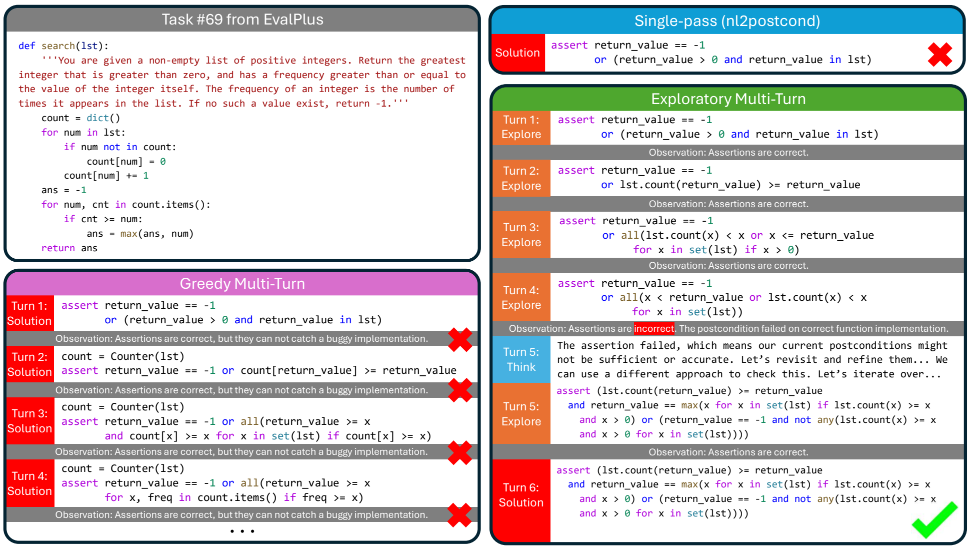
Specifications are vital for ensuring program correctness, yet writing them manually remains challenging and time-intensive. Recent large language model (LLM)-based methods have shown successes in generating specifications such as postconditions, but existing single-pass prompting often yields inaccurate results. In this paper, we present SPECMIND, a novel framework for postcondition generation that treats LLMs as interactive and exploratory reasoners rather than one-shot generators. SPECMIND employs feedback-driven multi-turn prompting approaches, enabling the model to iteratively refine candidate postconditions by incorporating implicit and explicit correctness feedback, while autonomously deciding when to stop. This process fosters deeper code comprehension and improves alignment with true program behavior via exploratory attempts. Our empirical evaluation shows that SPECMIND significantly outperforms state-of-the-art approaches in both accuracy and completeness of generated postconditions.

1 Introduction

Program specifications (pre-/postconditions) are central to checking that program behavior matches intent, but writing them manually is tedious. Prior work on automated inference broadly falls into four categories. First, program-analysis methods infer invariants dynamically from executions (Beschastnikh et al., 2011; Ernst et al., 1999)—and thus are coverage-limited—or statically from code, often at the cost of conservative, false-positive–prone results (Ramanathan et al., 2007; Wei et al., 2011). Second, data-mining approaches extract common API-usage patterns (e.g., call pairs/sequences and automata) from large codebases (Wasylkowski et al., 2007; Thummalapenta and Xie, 2009; Pradel and Gross, 2009; Wasylkowski and Zeller, 2009; Zhong et al., 2009a), but typically do not infer semantic specifications such as pre-/postconditions.

Third, EvoSpex (Molina et al., 2021) uses execution-guided evolutionary search to evolve postcondition candidates, yet its handcrafted operators only weakly exploit program semantics, leading to incomplete or brittle postconditions. Fourth, `nl2postcond` (Endres et al., 2024) leverages LLMs to translate code and informal documentation into postconditions. The limitation of simple prompting in `nl2postcond` lies in their reliance on *single-pass generation*: `nl2postcond` expects the LLM to generate a correct and complete postcondition in a single prompt. Such prompting results in *incorrect postconditions* that are syntactically plausible but semantically inaccurate, failing correctness tests or lacking discriminative power for bug detection. (Related work details are given in Appendix A.1).

In our work, SPECMIND, we aim to enhance the LLM's capability in generating postconditions from a given source code. A more robust alternative is *retry-based prompting* (**Greedy Multi-turn** or Greedy for short), where the LLM is iteratively asked to regenerate postconditions until one passes a soundness check and exceeds a certain threshold of completeness. While this approach introduces feedback into the loop, it remains *reactive and externally controlled* – the LLM is not actively involved in the reasoning process but simply retries in response to failure signals. Thus, we also propose **Exploratory Multi-Turn** (Exploratory for short), offering a more cognitively aligned framework in which the LLM is treated not just as a *code-to-code generator*, but as an **interactive reasoner**. We prompt the LLM to autonomously perform the exploration, internalize the reasoning process to derive the postcondition, and decide by itself when to stop the process. *We prompt the model to perform one or multiple turns for exploration*, i.e., submitting postconditions to obtain the intermediate feedback on soundness and completeness, before deciding to submit the final postcondition. *This method engages the model in an iterative self-refinement*



```
def search(lst):
    '''You are given a non-empty list of positive integers. Return the greatest
integer that is greater than zero, and has a frequency greater than or equal to
the value of the integer itself. The frequency of an integer is the number of
times it appears in the list. If no such a value exist, return -1.'''
    count = dict()
    for num in lst:
        if num not in count:
            count[num] = 0
        count[num] += 1
    ans = -1
    for num, cnt in count.items():
        if cnt >= num:
            ans = max(ans, num)
    return ans
```

Turn 1: Solution

```
assert return_value == -1
        or (return_value > 0 and return_value in lst)
```

Observation: Assertions are correct, but they can not catch a buggy implementation. ✗

Turn 2: Solution

```
count = Counter(lst)
assert return_value == -1 or count[return_value] >= return_value
```

Observation: Assertions are correct, but they can not catch a buggy implementation. ✗

Turn 3: Solution

```
count = Counter(lst)
assert return_value == -1 or all(return_value >= x
        and count[x] >= x for x in set(lst) if count[x] >= x)
```

Observation: Assertions are correct, but they can not catch a buggy implementation. ✗

Turn 4: Solution

```
count = Counter(lst)
assert return_value == -1 or all(return_value >= x
        for x, freq in count.items() if freq >= x)
```

Observation: Assertions are correct, but they can not catch a buggy implementation. ✗

. . .

Solution

```
assert return_value == -1
        or (return_value > 0 and return_value in lst)
```

✗

Turn 1: Explore

```
assert return_value == -1
        or (return_value > 0 and return_value in lst)
```

Observation: Assertions are correct.

Turn 2: Explore

```
assert return_value == -1
        or lst.count(return_value) >= return_value
```

Observation: Assertions are correct.

Turn 3: Explore

```
assert return_value == -1
        or all(lst.count(x) < x or x <= return_value
            for x in set(lst) if x > 0)
```

Observation: Assertions are correct.

Turn 4: Explore

```
assert return_value == -1
        or all(x < return_value or lst.count(x) < x
            for x in set(lst))
```

Observation: Assertions are incorrect. The postcondition failed on correct function implementation.

Turn 5: Think

```
The assertion failed, which means our current postconditions might
not be sufficient or accurate. Let’s revisit and refine them... We
can use a different approach to check this. Let’s iterate over...
```

Turn 5: Explore

```
assert (lst.count(return_value) >= return_value
  and return_value == max(x for x in set(lst) if lst.count(x) >= x
    and x > 0) or (return_value == -1 and not any(lst.count(x) >= x
    and x > 0 for x in set(lst))))
```

Observation: Assertions are correct.

Turn 6: Solution

```
assert (lst.count(return_value) >= return_value
  and return_value == max(x for x in set(lst) if lst.count(x) >= x
    and x > 0) or (return_value == -1 and not any(lst.count(x) >= x
    and x > 0 for x in set(lst))))
```

✓

Figure 1: Example of task #69 from EvalPlus with postconditions from Single-pass (nl2postcond (Endres et al., 2024)), Greedy Multi-turn, and Exploratory Multi-turn. Blue blocks show model reasoning (omitted for all but turn 5 of Exploratory Multi-turn for space reason), red, orange, and gray blocks show submission and exploration attempts, and feedback. ✓: correct and complete postcondition, ✗: otherwise.

process where it internalizes the feedback, reflects on prior attempts with the observations on the results of the intermediate turns, and gradually converges toward a more accurate and comprehensive postcondition. This feedback to LLMs can be implicit (e.g., soundness and/or completeness signals) or explicit (e.g., un-caught mutants). Greedy is a special case of Exploratory Multi-turn where no intermediate turn of exploration is allowed and the model submits the final result after each attempt. The key benefits of Exploratory Multi-turn include:

Deeper Code Comprehension: By interacting with feedback and refining its outputs over multiple turns, the model is encouraged to form a more semantically grounded understanding of the code logic, rather than relying on superficial patterns.

Autonomous Reasoning: In the autonomous exploration, the model determines when to stop, which introduces a degree of self-awareness and decision-making on the sufficiency of its answer.

We evaluated our model on EvalPlus (Endres et al., 2024) and FixEval (Anjum Haque et al., 2023). Exploratory Multi-turn achieves 99.4% correctness and 89.6% completeness, relatively improving over the baseline `nl2postcond` by 26.1% in correctness and 2.48X in completeness. The average increase of completeness after one submission attempt of Exploratory Multi-turn is 1.67X higher than the Greedy's. *The postconditions from our tool can detect 1.42X-2.14X more bugs than* `nl2postcond` *in FixEval with less costs (See Appendix A.2).*

2 Illustration

The function `search` in Fig. 1 *takes a non-empty list of positive integers as input and returns the greatest integer in the list whose frequency is greater than or equal to its value; if no such integer exists, it returns* -1. In the ground truth, the complete specification for this function must contain all of the following sub-conditions:

(1) the return value is either -1 or a positive integer from the input list;

(2) if the return value is positive, its frequency must be greater than or equal to its value;

(3) if the return value is positive, it must be the greatest integer satisfying the frequency condition;

(4) if the return value is -1, no integer in the list satisfies the frequency condition.

We applied the state-of-the-art, LLM-based, postcondition generation tool, `nl2postcond` (Endres et al., 2024) on the code of the `search` function. `nl2postcond` leverages a *single-pass* prompting strategy to request Llama 4 Scout to generate the postcondition for the given code.

A candidate postcondition can be evaluated using two metrics: *correctness and completeness* (Endres et al., 2024). **Correctness** refers to if the postcondition passes all test cases in the given test set. A postcondition is correct if it holds true for all test cases that the original function passes (no contradiction to the behavior in test cases). However, correctness alone is not sufficient as a trivial postcondition like `"True"` is always correct but useless in bug detection. **Completeness**, referring to discriminative power, measures how informative the postcondition is. It reflects the postcondition's ability to detect incorrect versions of the function (e.g., mutated code). This is done by checking if the postcondition can "catch" mutants (i.e., detecting bugs), i.e., the modified versions that behave differently. Formally, completeness is the percentage of mutants that can be distinguished (i.e., rejected) by the postcondition when used with the test suite. We made the following observations:

1. The state-of-the-art LLM-based, **Single-pass** approach, `nl2postcond` (Endres et al., 2024), generated the postconditions ensuring that the return value is either -1 or a positive integer from the input list, thereby satisfying the condition (1). However, none of the conditions (2)–(4) was generated.

2. **Feedback-driven Multi-Turn Approaches.** Prior work suggests that the feedback helps LLMs ground their generations more effectively (Gehring et al., 2025). Building on this, we integrated feedback into the process by explicitly indicating which generated constraints were incorrect or non-discriminative. Thus, we propose the following two feedback-driven Multi-Turn approaches:

Greedy Multi-Turn. The first approach employed a simple feedback mechanism in which Llama 4 Scout *repeatedly attempted to produce a correct postcondition* until all test cases passed and a desired completeness is reached, using feedback from previous failed attempts. This feedback includes either binary pass/fail signals or syntax errors. The iterative process enabled the model to progressively cover the condition (1) in the first attempt, the condition (2) in the second, and the condition (3) in the third, eventually combining them into a single assertion by the fourth attempt. Within the 12-attempt limit, the model focused predominantly on these three conditions, but failed to generate the condition (4). The generated assertions consistently began with `assert return_value == -1 or ...`, a pattern that could be trivially satisfied by faulty code returning -1 in all cases.

Exploratory Multi-turn. In real-world learning, students rarely solve complex problems correctly in a single attempt. Effective learners instead ask clarifying questions, explore partial solutions, and test their understanding before committing to a final answer. This differs from blindly retrying after a "try again" signal, and from single-pass solving without refinement. Inspired by this, we propose Exploratory Multi-turn for postcondition generation, where the LLM acts not as a code-to-code generator but as an **active reasoner**. It autonomously explores intermediate reasoning paths, probes partial answers, internalizes feedback, and decides when it has enough evidence to produce the result.

In the example, similar to Greedy Multi-turn, the first three attempts correctly generated conditions (1)–(3). In the 4th attempt, a minor change triggered an `AssertionError`. Unlike Greedy, which often stagnates on the same incomplete assertion, Exploratory Multi-Turn broke from its prior pattern and shifted direction in the fifth attempt. The model explicitly noted: *"The assertion failed, which means our current postconditions might not be sufficient or accurate. Let's revisit and refine them... We can use a different approach... Let's iterate over..."*. This shift produced a postcondition that covered condition (4) and discarded the overly permissive pattern `assert return_value == -1 or ...`. It finally generated the correct and complete postcondition, suggesting that Exploratory Multi-Turn helps escape local optima and improves completeness.

3 Problem Formulation

Given a function $f{:}\mathcal{I} \to \mathcal{O}$, where $\mathcal{I}$ and $\mathcal{O}$ are the input and output domains, the goal is to synthesize a postcondition $\phi : \mathcal{I} \times \mathcal{O} \to \{\texttt{true}, \texttt{false}\}$ that characterizes the correct behavior of f.

The postcondition ϕ is a logical predicate that must hold for any input-output pair (i, o) where $o = f(i)$. Formally: $\forall i \in \mathcal{I}, \quad \phi(i, f(i)) = \texttt{true}$. In practice, ϕ is often expressed as an assertion or a condition over program variables after execution. The goal is to infer such a ϕ automatically, using the following inputs: a function's source code, a finite test suite $\mathcal{T} = \{(i_1, o_1), \ldots, (i_n, o_n)\}$, and a set of mutants $\mathcal{M}$ of the original f. Two evaluation criteria for a synthesized postcondition ϕ include

Correctness (Corr.): The postcondition must hold on all known correct executions captured in the test suite: $\forall (i_j, o_j) \in \mathcal{T}, \quad \phi(i_j, o_j) = \texttt{true}$. The postcondition must reject incorrect variants.

Completeness (Comp.): Given a set of program mutants $\mathcal{M} = \{f'_1, f'_2, \ldots, f'_m\}$, the completeness of a postcondition measures the proportion of mutants for which the postcondition fails:

$$Comp(\phi) = \frac{|\{f'_k \in \mathcal{M} \mid \exists i \in \mathcal{I},\ \phi(i, f'_k(i)) = \texttt{false}\}|}{|\mathcal{M}|}$$

Correctness ensures consistency with the given code, while completeness reflects the specification's robustness in identifying deviations/defects.

```
Require: Function code $f$, test suite $T$, completeness threshold $\tau$
Ensure: Final postcondition $\phi_{\text{final}}$
1:  $\mathcal{H} \leftarrow [\,]$                                   ▷ History buffer
2:  $\phi_{\text{best}} \leftarrow \emptyset$, $s_{\text{best}} \leftarrow 0.0$
3:  while True do
4:      $prompt \leftarrow$ constructPrompt$(f, \mathcal{H})$
5:      $(\phi, type) \leftarrow$ LLM.generate$(prompt)$        ▷ $type \in$ {explore, submit}
6:      Append $(\phi, type)$ to $\mathcal{H}$                   ▷ Update history buffer
7:      if $type =$ submit then                                 ▷ $score \in [0, 1]$
8:          $(feedback, score) \leftarrow$ evaluateFeedback$(\phi, T)$
9:          annotateLast$(\mathcal{H}, feedback, score)$             ▷ Feedback
10:         if $score > s_{\text{best}}$ then
11:             $\phi_{\text{best}} \leftarrow \phi$, $s_{\text{best}} \leftarrow score$   ▷ Update best solution only on submission attempts
12:     if LLM.decidesToStop$(\mathcal{H}, s_{\text{best}})$ then
13:         return $\phi_{\text{best}}$        ▷ When the termination condition is met; return best-so-far
```

Figure 2: Feedback-Driven Exploratory Multi-Turn Algorithm with Completeness Threshold

Prompt Template

Objective: Verify correctness of a Python function using its natural language description and implementation. Your goal is to write symbolic postconditions — Python `assert` statements that validate specific behavioral properties of the function's return value.

Turn Structure: Each turn must begin with a `<think>` block containing reasoning about purpose, constraints, and edge cases. After this reasoning, choose exactly one of the following actions:

- `<assert>` – Propose a single candidate assertion for testing.
  → Triggers an `<observation>` with feedback.
- `<solution>` – Provide the final refined postcondition.

You will now be given the function `{function_name}`:

```
{function_signature}
    {function_docstring}
    {function_implementation}
```

Let's begin

Turn 1: $(\phi_1, \textbf{feedback}_1)$

`<think>` … `</think>` `<assert>` ϕ_1 `</assert>` `<observation>` feedback$_1$ `</observation>`

. . .

Turn $t-1$**:** $(\phi_{t-1}, \textbf{feedback}_{t-1})$

`<think>` … `</think>` `<assert>` ϕ_{t-1} `</assert>` `<observation>` feedback$_{t-1}$ `</observation>`

Figure 3: Template prompt for Greedy and Exploratory.

The highlighted `<assert>` *action is specific to* ***Exploratory Multiturn***.

Objective of Postcondition Inference/Generation. Given a function f, a test suite $\mathcal{T}$, and a set of mutants $\mathcal{M}$, the objective of SPECMIND is to synthesize a postcondition ϕ that:

1. Achieves full correctness over $\mathcal{T}$;
2. Maximizes completeness over $\mathcal{M}$;

4 Exploratory Multi-turn Algorithm

Algorithm supporting information *Completeness Score* $s \in [0, 1]$: Each feedback evaluation returns a quantitative measure of completeness (e.g., how much of the expected behavior is covered).

Threshold $\tau \in [0, 1]$: A predefined target completeness level. The loop continues until the model achieves this threshold or chooses to stop early.

Best-so-far Tracking: The algorithm keeps track of the best postcondition submitted by the model so far (based on the highest completeness score) and returns it if the loop ends prematurely.

LLM.decidesToStop. At each attempt, the model may choose to submit a postcondition once it is confident after a process of exploration and refinement or reaches a maximum number of attempts. The submitted postcondition is then evaluated and, if it improves upon the *Best-so-far Tracking* and satisfies the completeness threshold, the algorithm triggers *LLM.decidesToStop* to terminate and return the best postcondition generated so far.

Details. Fig. 2 outlines Exploratory Multi-turn for a given function. We treat inference as an iterative process guided by structured feedback: SPECMIND lets the LLM refine candidates until the completeness criterion is met or the submission budget is exhausted. We maintain a history buffer $\mathcal{H}$ that stores all attempted postconditions (exploratory and submitted) with metadata (line 1), and a best-so-far record (ϕ_{best}, s_{best}) tracking the strongest submitted candidate (line 2).

Each iteration synthesizes a prompt from the function code f and prior feedback in $\mathcal{H}$ (line 4), encouraging the model to reflect on earlier mistakes. It then produces either an exploratory refinement or a submission. Submitted candidates are evaluated by a feedback engine that returns qualitative outcomes (e.g., pass/fail) and a normalized completeness score $s \in [0, 1]$ (line 6). We log the submission, feedback, and score in $\mathcal{H}$ (line 9), and update (ϕ_{best}, s_{best}) if $s > s_{best}$ (lines 10–11). Termination is controlled by `LLM.decidesToStop` (line 12), which stops when a candidate reaches threshold τ, returns diminish, or the maximum attempts μ is reached; the algorithm returns ϕ_{best} (line 13).

Overall, we use the LLM as an interactive reasoner that incorporates feedback and self-regulates the search, reducing hallucination and overgeneralization. Greedy Multi-turn (not shown) repeatedly prompts the LLM to resubmit until all tests pass or until τ or the attempt budget is reached.

5 Prompt Design and Feedback

Prompt Design. We adopt a modular and evolving prompt template that reflects the model's current understanding, errors in prior attempts, and the functional structure of the code under analysis. The base prompt includes the full code of the target function and, optionally, a high-level natural language description (if available). Moreover, each subsequent prompt incorporates an interaction history consisting of prior postconditions, associated feedback, and explicit instructions for refinement. Formally, let the function be denoted f, and let the interaction history up to turn t be represented as:

$$\mathcal{H}_t = [(\phi_1, \text{feedback}_1), (\phi_2, \text{feedback}_2), \ldots, (\phi_{t-1}, \text{feedback}_{t-1})]$$

The prompt at turn t, denoted Prompt_t, is synthesized as: $\text{Prompt}_t = \texttt{PromptTemplate}(f, \mathcal{H}_t)$ where `PromptTemplate` is a function that formats code, prior generations, and critiques into an LLM-readable input. A prompt template is shown in Fig. 3.

Our evolving structure allows the LLM to internalize past mistakes and progressively refine its output. Unlike static prompt, our dynamic history-aware formulation fosters deeper reasoning and removes the need to reset the context between turns.

Feedback Mechanism. Our feedback mechanism serves two goals: 1) guiding the LLM's refinement with actionable signals, and 2) providing a quantitative estimate of each candidate's semantic completeness. Each LLM-generated postcondition ϕ_t is evaluated against the code's observed behavior, typically using a test suite and a set of mutants.

The feedback $\texttt{feedback}_t$ includes: (a) a **correctness indicator**–whether ϕ_t holds on all tests; and (b) a **completeness score**–the fraction of mutants "caught" by ϕ_t (i.e., faulty variants that violate the postcondition). If the completeness threshold is not reached within the attempt budget, we also return the remaining uncaught mutants as additional feedback. We append the prompt–feedback pair to the history $\mathcal{H}_t$ to steer subsequent generations toward remaining behavioral gaps. If available, we can use symbolic execution or verification backends; otherwise, correctness is checked empirically on tests and completeness is measured via mutant catching. Grounding prompts in correctness and completeness yields an informative refinement loop.

6 Empirical Evaluation

We aim to answer the following questions:

RQ1. [Effectiveness–Efficiency]. How effective and efficient is our tool in postcondition inference?

RQ2. [Reasoning]. How does SPECMIND with Exploratory Multi-turn perform reasoning?

RQ3. [Stratifying Results]. How does SPECMIND with feedback perform on hard cases?

RQ4. [Cost Efficiency]. How efficient is SPECMIND in token costs and actual costs?

Benchmarks. We selected EvalPlus as our dataset as it was used in `nl2postcond` (Endres et al., 2024). It has 164 Python problems, each with a function stub, textual description, reference implementation, and validation tests (Liu et al., 2023). EvalPlus updates the HumanEval benchmark (Chen et al., 2021), containing the same problems but with more extensive test suites ($\approx$ 775 test cases per problem). We also used FixEval (Anjum Haque et al., 2023) for bug detection experiment (see Appendix A.2).

Baselines. We chose as baseline the state-of-the-art `nl2postcond`, with access to the reference implementation. `nl2postcond` was shown to outperform prior approaches such as TOGA (Dinella et al., 2022) and Daikon (Ernst et al., 1999). We used Llama 4 Scout as the underlying LLM for both `nl2postcond` and our tool in all experiments.

Evaluation Metrics. We use two key metrics: 1) *Correctness* refers to whether the postcondition passes all test cases in the given test suite; 2) Completeness is the percentage of mutants distinguishable (i.e., rejected) by the postcondition given the test suite. We used the same mutant set from `nl2postcond` for a fair comparison.

To calculate the efficiency over submission attempts, we define the efficiency score in a setting as $\mathbf{E} = \frac{1}{N} \sum_{i=1}^{N} \frac{\text{completeness score}_i}{\text{number of submission turns}_i}$, where N is the number of functions in the dataset. The efficiency score $\mathbf{E}$ measures the average increase of completeness after one submission attempt.

6.1 Effectiveness and Efficiency (RQ1)

The values in bold in Table 1 indicate *the best results* across all configurations, with Exploratory achieving peak performance at τ=90, μ=12 (**99.4% correctness, 89.6% completeness**) while requiring an average of 7.2 attempts and 1.7 submissions per task. `nl2postcond` achieves 73.3% correctness and 36.0% completeness using a single attempt.

Across most settings, for both Greedy and Exploratory Multi-turn, despite allowing a higher maximum number of attempts, *the average number of attempts range between 2.3-7.2, with the correctness increasing from 84.3% to 99.4% and completeness increasing from 69.2% to 89.6%*. The higher maximum number of attempts allowed, the higher correctness and completeness for Greedy and Exploratory Multi-turn. For Exploratory Multi-turn, among 3.8-7.2 attempts, it explores 3.5 times

Table 1: Postcondition Generation Effectiveness (RQ1). τ: completeness threshold, μ: max turns; R. Sampl.: run nl2postcond μ independent times, Subs: avg submissions, Corr: correctness, Comp.: Completeness.

Method	**Config.**	**Attempts min-max**	**Avg Subs min-max**	**Corr.**	**Comp.**
nl2postcond (Baseline)	Single-pass	1.0	1.0	73.3%	36.0%
R.Sampl. w. nl2postcond	$\tau = 50$ $\mu = 4$	3.0 (1 - 4)	3.0 (1 - 4)	79.2%	41.2%
Greedy		2.3 (1 - 4)	2.3 (1 - 4)	93.7%	69.2%
Exploratory		3.8 (3 - 4)	1.0 (1 - 2)	84.3%	70.6%
R.Sampl. w. nl2postcond	$\tau = 50$ $\mu = 8$	5.1 (1 - 8)	5.1 (1 - 8)	84.9%	47.9%
Greedy		2.9 (1 - 8)	2.9 (1 - 8)	96.9%	70.7%
Exploratory		5.7 (3 - 8)	1.2 (1 - 3)	93.1%	81.5%
R.Sampl. w. nl2postcond	$\tau = 50$ $\mu = 12$	6.9 (1 - 12)	6.9 (1 - 12)	84.9%	50.9%
Greedy		3.3 (1 - 12)	3.3 (1 - 12)	100.0%	75.0%
Exploratory		6.5 (3 - 12)	1.3 (1 - 5)	98.1%	86.4%
R.Sampl. w. nl2postcond	$\tau = 70$ $\mu = 4$	3.5 (1 - 4)	3.5 (1 - 4)	79.2%	41.3%
Greedy		2.6 (1 - 4)	2.6 (1 - 4)	95.6%	68.6%
Exploratory		3.8 (3 - 4)	1.0 (1 - 2)	84.3%	70.4%
R.Sampl. w. nl2postcond	$\tau = 70$ $\mu = 8$	6.4 (1 - 8)	6.4 (1 - 8)	84.9%	48.4%
Greedy		3.8 (1 - 8)	3.8 (1 - 8)	96.9%	78.6%
Exploratory		5.7 (2 - 8)	1.2 (1 - 4)	93.1%	83.7%
R.Sampl. w. nl2postcond	$\tau = 70$ $\mu = 12$	9.0 (1 - 12)	9.0 (1 - 12)	84.9%	51.9%
Greedy		5.3 (1 - 12)	5.3 (1 - 12)	100.0%	78.4%
Exploratory		6.5 (3 - 12)	1.4 (1 - 6)	97.5%	85.9%
R.Sampl. w. nl2postcond	$\tau = 90$ $\mu = 4$	3.7 (1 - 4)	3.7 (1 - 4)	79.2%	41.5%
Greedy		3.2 (1 - 4)	3.2 (1 - 4)	96.9%	76.4%
Exploratory		3.8 (3 - 4)	1.1 (1 - 2)	84.3%	71.6%
R.Sampl. w. nl2postcond	$\tau = 90$ $\mu = 8$	7.1 (1 - 8)	7.1 (1 - 8)	84.9%	48.9%
Greedy		5.1 (1 - 8)	5.1 (1 - 8)	99.4%	81.1%
Exploratory		6.2 (3 - 8)	1.4 (1 - 4)	94.3%	83.2%
R.Sampl. w. nl2postcond	$\tau = 90$ $\mu = 12$	10.3 (1 - 12)	10.3 (1 - 12)	84.9%	52.6%
Greedy		6.2 (1 - 12)	6.2 (1 - 12)	98.7%	85.8%
Exploratory		7.2 (3 - 12)	1.7 (1 - 6)	**99.4%**	**89.6%**

on average and submits the resulting postcondition from 1.0 to 1.7 times. As seen, on average, SpecMind uses *a small number of attempts* and does not need to use up the attempt budget. However, the highest numbers of attempts reach the maximum budget. That means that *it needs to use all the budget of attempts in hard cases* (see RQ3).

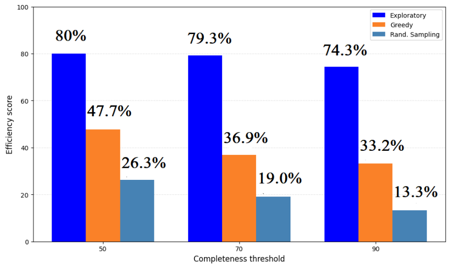


Figure 4: Efficiency for configurations with μ=12.

6.1.1 Correctness

We first study the effect of the attempt budget μ. With μ=4, both Exploratory and Greedy Multi-turn outperform `nl2postcond`. Under the most relaxed setting (τ=50, μ=4), Greedy reaches 93.7% correctness in 2.3 attempts on average, a 20-point gain over `nl2postcond` (73.3%). Exploratory also improves over `nl2postcond`, but achieves 84.3% due to limited exploratory turns. Exploratory benefits more from larger μ: the average number of functions with correct postconditions rises from 139.0 ($\mu = 4$) to 153.7 (μ=8) and 161.3 (μ=12). Greedy improves less with μ. On average, increasing μ by 4 yields a 6.8% correctness gain for Exploratory versus 2.1% for Greedy, as Exploratory can probe more candidates before submission (Section 6.2).

For comparison, we also evaluate **Random Sampling** with `nl2postcond`, i.e., querying the LLM independently μ times with early stopping when τ is met. Random Sampling achieves 79.2% correctness at (τ=50, μ=4) and peaks at 84.9% under the strictest setting (τ=90, μ=12). Under the same setting, Greedy and Exploratory reach 98.7% and 99.4% correctness, using 6.2 and 7.2 attempts on average, compared to 10.3 for Random Sampling.

6.1.2 Completeness

Exploratory Multi-turn outperforms Greedy in all settings except τ=90, μ=4. The smallest completeness gap is at τ=50, μ=4 (70.6% vs. 69.2%), and the largest at τ=50, μ=12 (86.4% vs. 75.0%). Our tool's lowest completeness is 68.6% at τ=70, μ=4, far above single-pass `nl2postcond` (36.0%), highlighting the benefit of multi-turn refinement for more discriminative specifications. Random Sampling improves with more attempts (e.g., at τ=90, 41.5% with μ=4 to 52.6% with μ=12) but remains well below SpecMind's worst case (68.6%). We also analyze the dynamics of completeness scores across submission attempts. Representative trends and aggregate statistics are in Appendix A.3.

6.1.3 Efficiency

Fig. 4 shows that Exploratory Multi-turn is more submission-efficient than Greedy, peaking at 80.0% vs. 47.7%. That is, the average completeness gain per submission is **1.67**× higher for Exploratory. Raising the completeness threshold increases final completeness but reduces efficiency: from τ=50 to τ=90, efficiency drops by 5.7% for Exploratory and 14.5% for Greedy, suggesting that more discriminative postconditions require more exploratories. Compared to Random Sampling, Exploratory is more efficient (80.0% vs. 26.3%) and degrades less as τ increases. As τ rises from 50-70-90, Exploratory achieves 3.04×, 4.17×, and 5.59× the efficiency of Random Sampling. See a case study comparing two approaches in Appendix A.4.

Table 2: Exploratory Multi-turn's reasoning behavior categories with descriptions, frequencies, examples (RQ2).

Reasoning Category	Description	Freq	Example
Return Value Type & Range	Checking the type and range of the return value.	77	"The assertion I proposed checks if … To further strengthen the assertion, I can check if the return value is a list of strings…"
Check Base Cases	Checking the base cases of a function.	32	"Let's check if the function returns the correct values for the base cases. For $n = 0$, the return value should be 0…"
Check Edge Cases	Adding postconditions checking edge cases: empty inputs or boundary values.	63	"Let's consider the case where the input number is an integer. The decimal part should be exactly 0…"
Postcondition Combination	Combining multiple postconditions into one.	59	"Both properties seem to be correct… Let's combine these insights into a single assertion…"
Refine Assertions	Refining assertions to make them more concise, meaningful, or comprehensive.	419	"The combined assertion seems correct and comprehensive. However, let's make it more concise and readable…"
Verify Function Behavior	Adding assertions to verify correctness of calculations.	270	"The first assertion checks… Next, we want to check that the return value, when added to apples and blues, equals the total number of fruits…"
Submit Final Solution	Submitting assertions as the final solution without further refining.	129	"The assertion is now concise and readable. It correctly checks…, let's consider submitting it..."

6.2 SpecMind's Reasoning at Each Attempt

6.2.1 Experimental Methodology

This experiment analyzes SpecMind's reasoning behavior across attempts, focusing on Exploratory Multi-turn because each attempt may depend on prior exploratory or submission turns. (Greedy Multi-turn contains only submission turns, which are a subset of Exploratory.) We use τ=70 since τ=50 often stops too early and τ=90 frequently exhausts the budget without meeting the threshold. We set μ=12 to observe behavior over more attempts. We analyze all natural-language reasoning texts produced per attempt (enclosed by `<think>...</think>`), totaling 1,026 instances; each case has 6.5 reasoning instances on average.

To categorize reasoning behaviors, we first use an LLM to cluster reasoning texts in five batches (200–226 each), producing distinct, non-overlapping categories. We then manually consolidate and refine these into seven observable categories. Finally, we use an LLM to classify all reasoning texts into the seven categories for consistent labeling, and manually validate the results by inspecting a random 10% sample per category.

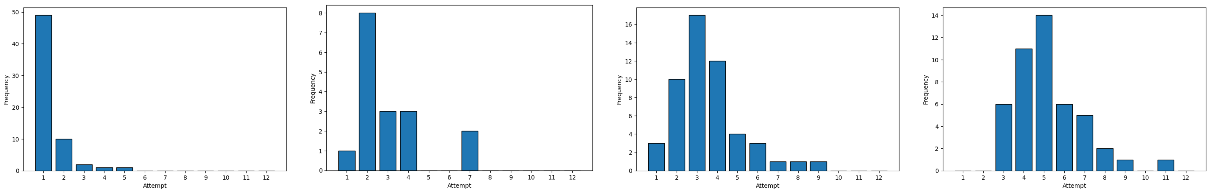
6.2.2 Reasoning of Exploratory Multi-turn

Table 2 presents the categories of SpecMind's reasoning along with their descriptions, frequencies, and representative examples. The most frequent behavior is *Refine Assertions*, appearing in 419 reasonings, where the model improves existing assertions to make them more concise and comprehensive. Typical phrases include: *"To refine our postconditions..."*, *"A more accurate property might be..."*, or *"The current postcondition does not accurately reflect..."*. The second most frequent category is *Verify Function Behavior*, with 270 reasonings. In this category, the model proposes assertions to verify functional properties.

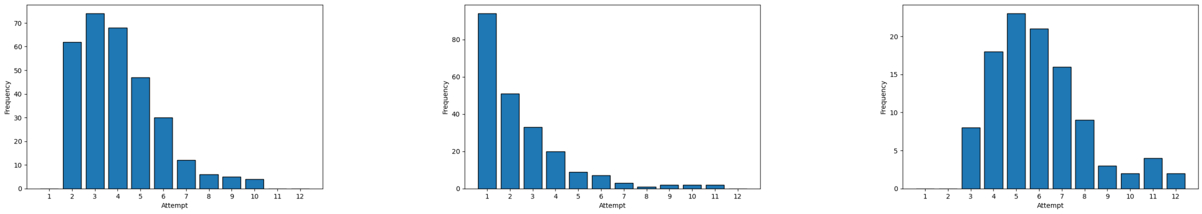
Reasoning Patterns. Fig. 5 shows the occurrences of *7 categories across attempts* for all samples, excluding unsolvable cases. *Return Value Type and Range* and *Verify Function Behavior* concentrate in the first attempt (49 and 94 instances). *Check Base Cases* and *Check Edge Cases* also appear early, peaking at the 2nd (8) and 3rd (17) attempts, indicating that the model initially explores return values and base/edge cases. *Submit Final Solution* occurs from attempts 3–12 and peaks around attempts 5–6, consistent with Exploratory requiring about 5–6 attempts per case. *Refine Assertions* is frequent between attempts 2–6, peaking at attempts 3–4 (74 and 68), suggesting the model refines before submitting. *Postcondition Combination* emerges at attempt 3 and peaks at attempt 5 (14), indicating it combines exploratory results before submitting. See Appendix A.5 for an example.

6.3 Stratifying Results on Hard Cases (RQ3)

We analyze the hardest cases: the model exhausts the attempt budget without reaching the completeness threshold, under the strict setting τ=90, μ=12. This yields 23 hard cases out of 159: one with no correct postcondition and 22 with completeness < 90 (average 56.2%). In the standard setting, we provide only binary feedback: whether a postcondition passes all tests (correctness) and whether it catches all mutants (completeness). For these hard cases, we **add richer feedback** by returning a randomly-selected uncaught mutant and asking the



(a) Ret. Val. Type/Range (b) Check Base Cases (c) Check Edge Cases (d) Postcond. Comb.



(e) Refine Assertions (f) Verify Function Behavior (g) Submit A Solution

Figure 5: Frequency distribution analysis from seven reasoning categories across attempts (RQ2)

model for refinement. We then extend the budget by 4 attempts (μ=16) to measure the impact.

With enhanced feedback, correct postconditions increase to **22/23** and average completeness rises to **85.1%**; the model reaches the threshold in 14/22 correct cases. Applying binary feedback first (τ=90, μ=12) and using enhanced feedback only for the remaining hard cases with 4 extra attempts improves performance to **99.7%** correctness and **92.0%** completeness (vs. 99.4% and 89.6% with binary feedback only). If we use enhanced feedback from the start with τ=90, μ=12, the model produces correct postconditions for 21/23 hard cases; among 22 correct cases, 9 reach the completeness target, with overall average completeness 90.6%.

6.4 LLM Token Usage & Cost Analysis (RQ4)

We analyze the token usage between Random Sampling (running `nl2postcond` μ independent times) and SPECMIND. Under the strictest configuration (completeness threshold τ=90, maximum number of turns μ=12), Random Sampling uses an average of 302 tokens per instance (min=19 and max=1,666 tokens), which is much lower than the usages of Multi-turn approaches. Greedy uses an average of 5,752 tokens per instance (ranging from 266 to 25,909), while Exploratory uses an average of 8,299 tokens per instance (ranging 1,313–29,956).

The higher token usage per instance for our tool stems from two factors: (1) inclusion of Chain-of-Thought tokens (`nl2postcond` does not have CoT) and (2) inclusion of the history buffer (Equation 1). However, the actual costs remain modest: 0.18 cent and 0.25 cent per instance for Greedy and Exploratory, respectively (τ=90, μ=12).

This high token usage is a **tradeoff for more bug detection**. Fig. 6 displays the trends for Random Sampling, Greedy, and Exploratory when the completeness threshold increases (μ=50,70,90). When μ increases, each approach uses more tokens due to more attempts. However, across thresholds, both Greedy and Exploratory catch more mutants/bugs than Random Sampling. Greedy caught 24.7 mutants with 3,065.9 tokens per instance ($\approx$ 0.0029 cent/bug) and 27.8 mutants with 5,752 tokens per instance ($\approx$ 0.0047 cent/bug). Exploratory achieved the highest bug detection performance, reaching 29.2 mutants with 6110.5 tokens per instance ($\approx$ 0.0046 cent/bug) and peaking at 30.9 with 8,299 tokens per instances ($\approx$ 0.0058 cent/bug). Random Sampling caught 17.6 mutants with 298 tokens. Using more tokens (3,182 tokens) with maximum allowed attempts μ=170, its number of detected bugs reaches to 23.3 (**1.4 bugs less** compared to 24.7 bugs by Greedy with 3k tokens), with 0.0082 cent/bug (**2.83X cost more**). Random Sampling has higher cost as it generates more output tokens (which cost more), whereas our tool requires more input tokens but produces fewer outputs. In brief, **with lower attempts, Exploratory incurs lower costs with more detected bugs**.

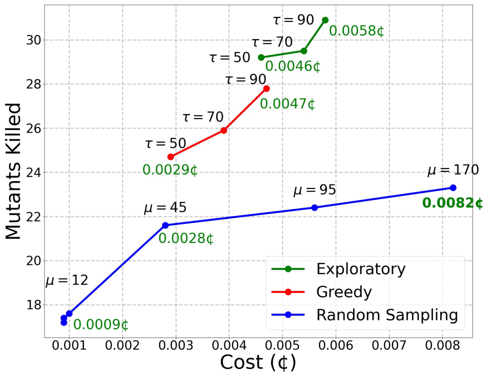


Figure 6: Token Usages & Numbers of Caught Mutants

Limitations & Threats to Validity

Our evaluation is conducted on EvalPlus (Endres et al., 2024), which may not reflect the full diversity of real-world codebases. While EvalPlus is a large public benchmark with postcondition annotations and enables a fair comparison against `nl2postcond` on its original setting, it primarily consists of Python tasks of moderate complexity. Thus, generalization to larger systems, other programming languages, or other forms of specifications is not guaranteed. In addition, we evaluate mainly GPT-family models; performance may differ for other model families or future versions.

Our results may also be influenced by the underlying pre-training knowledge of LLMs. Although we cannot fully rule out exposure to similar code during pretraining, our evaluation centers on generating new postconditions and validating them via execution, which is unlikely to be explained solely by memorization.

We operationalize postcondition quality using two proxies: correctness (passing the test suite) and completeness (rejecting mutants). Both measures inherit limitations from their backends: correctness is bounded by test coverage, and completeness depends on the quality and representativeness of the mutant set. If the tests or mutants fail to capture realistic behaviors or defects, our metrics may overestimate true specification quality or bug-detection usefulness.

Finally, our conclusions depend on experimental choices such as the attempt budget μ and completeness threshold τ. While we evaluate across multiple configurations to reduce sensitivity to any single setting, other datasets, feedback designs, or hyperparameter choices could change the observed trade-offs between effectiveness and efficiency.

References

Juan Altmayer Pizzorno and Emery D. Berger. 2025. Coverup: Effective high coverage test generation for python. *Proc. ACM Softw. Eng.*, 2(FSE).

Glenn Ammons, Rastislav Bodík, and James R. Larus. 2002. Mining specifications. In *Proceedings of the 29th ACM SIGPLAN SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 4–16. ACM.

Md Mahim Anjum Haque, Wasi Uddin Ahmad, Ismini Lourentzou, and Chris Brown. 2023. FixEval: Execution-based Evaluation of Program Fixes for Programming Problems . In *2023 IEEE/ACM International Workshop on Automated Program Repair (APR)*, pages 11–18, Los Alamitos, CA, USA. IEEE Computer Society.

Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D. Ernst. 2011. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proceedings of the 19th Symposium on Foundations of Software Engineering*, ESEC/FSE '11, pages 267–277. ACM.

Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. 2018. Translating code comments to procedure specifications. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, page 242–253, New York, NY, USA. Association for Computing Machinery.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Patrick M. Cousot, Radhia Cousot, Francesco Logozzo, and Michael Barnett. 2012. An abstract interpretation framework for refactoring with application to extract methods with contracts. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, page 213–232, New York, NY, USA. Association for Computing Machinery.

Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu K. Lahiri. 2022. Toga: a neural method for test oracle generation. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 2130–2141, New York, NY, USA. Association for Computing Machinery.

Madeline Endres, Sarah Fakhoury, Saikat Chakraborty, and Shuvendu K. Lahiri. 2024. Can large language models transform natural language intent into formal method postconditions? *Proc. ACM Softw. Eng.*, 1(FSE).

Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP'01, pages 57–72. ACM.

Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. 1999. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE'99, pages 213–224. ACM.

Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. 2016. Learning invariants using decision trees and implication counterexamples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, page 499–512, New York, NY, USA. Association for Computing Machinery.

Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Taco Cohen, and Gabriel Synnaeve. 2025. RLEF: Grounding code LLMs in execution feedback with reinforcement learning. In *Forty-second International Conference on Machine Learning*.

Darren Key, Wen-Ding Li, and Kevin Ellis. 2022. I speak, you verify: Toward trustworthy neural program synthesis. *CoRR*, abs/2210.00848.

Ted Kremenek, Paul Twohey, Godmar Back, Andrew Ng, and Dawson Engler. 2006. From uncertainty to belief: inferring the specification within. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 161–176. USENIX Association.

Shuvendu K. Lahiri, Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, Madanlal Musuvathi, Piali Choudhury, Curtis von Veh, Jeevana Priya Inala, Chenglong Wang, and Jianfeng Gao. 2023. Interactive code generation via test-driven user-intent formalization. *Preprint*, arXiv:2208.05950.

Larissa Laich, Pavol Bielik, and Martin Vechev. 2020. Guiding program synthesis by learning to generate examples. In *International Conference on Learning Representations*.

Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, page 919–931. IEEE Press.

Zhenmin Li and Yuanyuan Zhou. 2005. Pr-miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 13th Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 306–315. ACM.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Benjamin Livshits and Thomas Zimmermann. 2005. Dynamine: finding common error patterns by mining software revision histories. *SIGSOFT Softw. Eng. Notes*, 30(5):296–305.

Antonio Mastropaolo, Nathan Cooper, David Nader Palacio, Simone Scalabrino, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2023. Using transfer learning for code-related tasks. *IEEE Transactions on Software Engineering*, 49(4):1580–1598.

Facundo Molina, Pablo Ponzio, Nazareno Aguirre, and Marcelo Frias. 2021. Evospex: An evolutionary algorithm for learning postconditions. In *Proceedings of the 43rd International Conference on Software Engineering*, ICSE '21, page 1223–1235. IEEE Press.

Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. 2009. Graph-based mining of multiple object usage patterns. In *Proceedings of the Symposium on Foundations of Software Engineering*, ESEC/FSE '09, pages 383–392. ACM.

Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. 2012. Inferring method specifications from natural language api descriptions. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, page 815–825. IEEE Press.

Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2023. Can large language models reason about program invariants? In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 27496–27520. PMLR.

Michael Pradel and Thomas R. Gross. 2009. Automatic generation of object usage specifications from large method traces. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 371–382. IEEE Computer Society.

Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. 2007. Static specification inference using predicate mining. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 123–134. ACM.

Sharon Shoham, Eran Yahav, Stephen Fink, and Marco Pistoia. 2007. Static specification mining using automata-based abstractions. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, page 174–184, New York, NY, USA. Association for Computing Machinery.

Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. 2007. /*icomment: bugs or bad comments?*/. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, page 145–158, New York, NY, USA. Association for Computing Machinery.

Lin Tan, Yuanyuan Zhou, and Yoann Padioleau. 2011. acomment: mining annotations from comments and code to detect interrupt related concurrency bugs. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, page 11–20, New York, NY, USA. Association for Computing Machinery.

Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T. Leavens. 2012. @tcomment: Testing javadoc comments to detect comment-code inconsistencies. In

Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, ICST '12, page 260–269, USA. IEEE Computer Society.

Suresh Thummalapenta and Tao Xie. 2009. Alattin: Mining alternative patterns for detecting neglected conditions. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 283–294. IEEE Computer Society.

Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2021. Unit test case generation with transformers and focal context. *Preprint*, arXiv:2009.05617.

Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, and Neel Sundaresan. 2022. Generating accurate assert statements for unit test cases using pretrained transformers. In *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*, AST '22, page 54–64, New York, NY, USA. Association for Computing Machinery.

Vasudev Vikram, Caroline Lemieux, Joshua Sunshine, and Rohan Padhye. 2024. Can large language models write good property-based tests? *Preprint*, arXiv:2307.04346.

Andrzej Wasylkowski and Andreas Zeller. 2009. Mining temporal specifications from object usage. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 295–306. IEEE Computer Society.

Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. 2007. Detecting object usage anomalies. In *Proceedings of the Symposium on Foundations of Software Engineering*, ESEC-FSE '07, pages 35–44. ACM.

Yi Wei, Carlo A. Furia, Nikolay Kazmin, and Bertrand Meyer. 2011. Inferring better contracts. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 191–200. ACM.

Chadd C. Williams and Jeffrey K. Hollingsworth. 2005. Automatic mining of source code repositories to improve bug finding techniques. *IEEE Trans. Softw. Eng.*, 31(6):466–480.

Jianan Yao, Gabriel Ryan, Justin Wong, Suman Jana, and Ronghui Gu. 2020. Learning nonlinear loop invariants with gated continuous logic networks. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 106–120, New York, NY, USA. Association for Computing Machinery.

Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. 2009a. Mapo: Mining and recommending api usage patterns. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, pages 318–343. Springer-Verlag.

Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. 2009b. Inferring resource specifications from natural language api documentation. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, page 307–318, USA. IEEE Computer Society.

Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald Gall. 2017. Analyzing apis documentation and code to detect directive defects. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 27–37.

A Appendix

A.1 Related Work

Researchers have proposed automated approaches that fall into three categories: *program analysis-based*, *data mining-based*, and *large language model (LLM)-based.*

Program analysis-based methods use dynamic or static techniques. Dynamic approaches (Ammons et al., 2002; Beschastnikh et al., 2011; Ernst et al., 1999) infer properties by monitoring executions but depend heavily on test coverage. Static analyses (Shoham et al., 2007; Engler et al., 2001; Kremenek et al., 2006; Ramanathan et al., 2007; Wei et al., 2011) and abstract interpretation (Cousot et al., 2012) avoid this dependency but often produce conservative or imprecise specifications due to high false-positive rates. Several approaches attempted to generate specification by analyzing API documentation or code comments (Pandita et al., 2012; Tan et al., 2007, 2011, 2012; Blasi et al., 2018; Zhong et al., 2009b; Zhou et al., 2017).

Data mining-based approaches extract common API usage patterns from large codebases (Li and Zhou, 2005; Livshits and Zimmermann, 2005; Zhong et al., 2009a). They identify call pairs, sequences, and finite-state models (Wasylkowski et al., 2007; Williams and Hollingsworth, 2005; Thummalapenta and Xie, 2009; Nguyen et al., 2009; Pradel and Gross, 2009; Wasylkowski and Zeller, 2009). While useful for API usages, they rarely infer semantic specifications such as pre/-postconditions (Ramanathan et al., 2007).

In general, ML approaches for specification generation have advanced in multiple directions, including postcondition inference (Endres et al., 2024), synthesizing test oracles (Dinella et al., 2022; Mastropaolo et al., 2023; Tufano et al., 2022), improving test coverage (Lemieux et al., 2023; Altmayer Pizzorno and Berger, 2025), and generating unit tests (Lahiri et al., 2023; Tufano et al., 2021).

These techniques rely on different types of inputs. AthenaTest (Tufano et al., 2021) generates both the input and the oracle of a unit test directly from the implementation of the focal method, while TOGA generates only the oracle (Dinella et al., 2022). TiCoder (Lahiri et al., 2023) leverages LLMs to produce both inputs and outputs based on a natural language description of user intent.

While these methods target the production of concrete test cases (and sometimes oracles), our work instead focuses on generating postconditions. Similarly, EvoSpex (Molina et al., 2021) applies evolutionary learning to infer functional input–output relationships. More recent efforts have explored property-based specifications. Vikram *et al.* (Vikram et al., 2024) propose leveraging LLMs to generate property-based tests (PBTs), while Speculyzer(Key et al., 2022) enumerates likely properties and candidate inputs with LLMs, but uses them primarily as heuristics to guide code generation rather than to ensure correctness or completeness of specifications. Beyond input–output properties, machine learning has also been applied to infer intermediate specifications, such as program invariants (Garg et al., 2016; Laich et al., 2020; Yao et al., 2020; Pei et al., 2023).

A.2 Bug Detection via Generated Postconditions

Experimental Methodology. In this experiment, we aim to evaluate the usefulness of the generated postconditions. We adapted an evaluation approach used in `nl2postcond` (Endres et al., 2024) and TOGA (Dinella et al., 2022). The evaluation idea is that it aims to evaluate if a model can generate postconditions that distinguish between correct and buggy code regarding regression and triggering test cases in regression testing. We used FixEval (Anjum Haque et al., 2023), a benchmark comprising buggy and correct submission code to competitive programming problems. From the full test set of 243k Python buggy-correct pairs, we filtered out runtime errors and kept 157,355 pairs. We then randomly selected 384 bugs, ensuring 95% statistical significance. We evaluated the effectiveness of postconditions that a user could have used to catch a bug as the postconditions fail on the buggy version, and succeed on the correct one. The rationale is as follows. Using the associated test suite for a problem in the dataset to ensure correctness, as we run a model on the correct version, the resulting postcondition ϕ acts as an oracle of intended behavior of the code. When we apply that oracle to the executions of the buggy solution, any deviation from the intended relation between inputs and outputs will violate the postcondition ϕ (i.e., there exists an input i such that postcondition ϕ(i,buggy_function(i))=`false`). This is the same completeness definition: postconditions "reject" a faulty variant (mutant); here, the buggy version plays the role of a single mutant. Thus, the generated postcondition can be used in detecting future bugs in regression testing. We used the same criteria in `nl2postcond` for a *bug-discriminating* postcondition. We consider a generated postcondition to be bug-discriminating if it satisfies the following: (1) The postcondition passes all the trigger and regression tests, on the correct version of a function. (2) The postcondition fails a trigger test or a regression test on the buggy version (Endres et al., 2024).

Experimental Results. As seen in Table 3, both Greedy and Exploratory approaches are able to distinguish more buggy versions than the baseline `nl2postcond` being run μ times independently (for a fair comparison). When μ=3 submission attempts, the baseline distinguished 16.15% of the buggy versions from the correct ones, with passing 66.41% of test cases. In contrast, Greedy distinguishes relatively 53.19% more buggy versions with passing 59.38% of test cases. Importantly, Exploratory catches **2X** more buggy versions than the baseline with **34.64%** of passing test cases.

As we increased the number of submission attempts to $\mu = 5$, all approaches performs better in terms of both correctness and number of distinguishable bugs. As $\mu = 10$, Exploratory performs the best as it distinguished **39.58%** of the buggy versions, i.e., better than both the baseline and Greedy.

An Example. Let us use an example of the correct code (not shown), aiming to solve the task: *given a starting point and a mapping of each point to the next, find the position reached after K steps.*

1. When running the baseline `nl2postcond` in Random Sampling μ independent times, the LLM does not learn from its past experience, leading to repeated mistakes. For example, the postconditions produced at the attempts #2 and #5 are the same even though the one at attempt #2 is incorrect. This approach exhausted all 10 attempts without catching the bug.

2. For Greedy approach, the LLM demonstrated the ability to learn from past mistakes. For example,

Table 3: Performance on Distinguishing Bugs. μ: # of submissions; R.Sampl.: run `nl2postcond` μ independent times; Corr.: correctness; Dist. Bugs: # of distinguishable bugs.

Approach	$\mu = 3$		$\mu = 5$		$\mu = 10$	
	% Corr.	% dist. bugs	% Corr.	% dist. bugs	% Corr.	% dist. bugs
R.Sampl. with nl2postcond	66.41%	16.15%	76.82%	22.40%	86.20%	27.86%
Greedy	59.38%	24.74%	67.71%	30.73%	87.24%	38.02%
Exploratory	**79.69%**	**34.64%**	**90.10%**	**36.98%**	**95.57%**	**39.58%**

in attempt #2, it proposed to check if the output is a positive number:

```
assert isinstance(return_values[0], int) and return_values[0] > 0
```

Getting the correctness feedback, it simplified the condition to only check if the output is an integer: `assert isinstance(return_values[0], int)`. However, none of the 10 retries can catch the bug.

3. Exploratory Multi-turn started with a simple postcondition checking if the output is an integer. After getting the feedback on its incorrect condition, it **expanded the condition to a correct one** to check that the output string can be converted to an integer.

We also observed that Exploratory was able to **refine the postcondition**. For example, at attempt #3, it checks that the output index is within the bounds of the sequence:

```
N, K = map(int, lines[0].split()); sequence = list(map(int, lines[1].split()));
assert 1 <= int(return_values[0]) <= N
```

After getting the positive feedback, it continued to refine the post condition to a stricter one and eventually submitted the correct one at the attempt #5. Its reasoning texts are as follows: *"The current postcondition checks if the output index is within the bounds ... However, to make the postcondition more specific and meaningful, I will consider...Given that the sequence represents a linked list with possible cycles, the output value should be an element in the sequence."* The final condition caught the buggy version as the output is not an element of the sequence.

```
N, K = map(int, lines[0].split()); sequence = list(map(int, lines[1].split()))
assert int(return_values[0]) in sequence
```

A.3 Behavioral Patterns in Completeness Across Submissions

To analyze the change in completeness score across submission attempts, we consider the strictest setting with completeness threshold $\tau = 90$ and maximum number of turns $\mu = 12$. We plot the completeness score over submission attempts for eight selected solvable cases (i.e., cases where the final solution is correct and satisfies the completeness target) in Fig. 7. These cases are representative of typical behaviors observed in our dataset and are selected here for illustration.

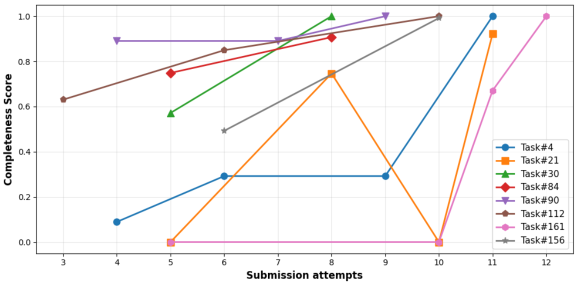
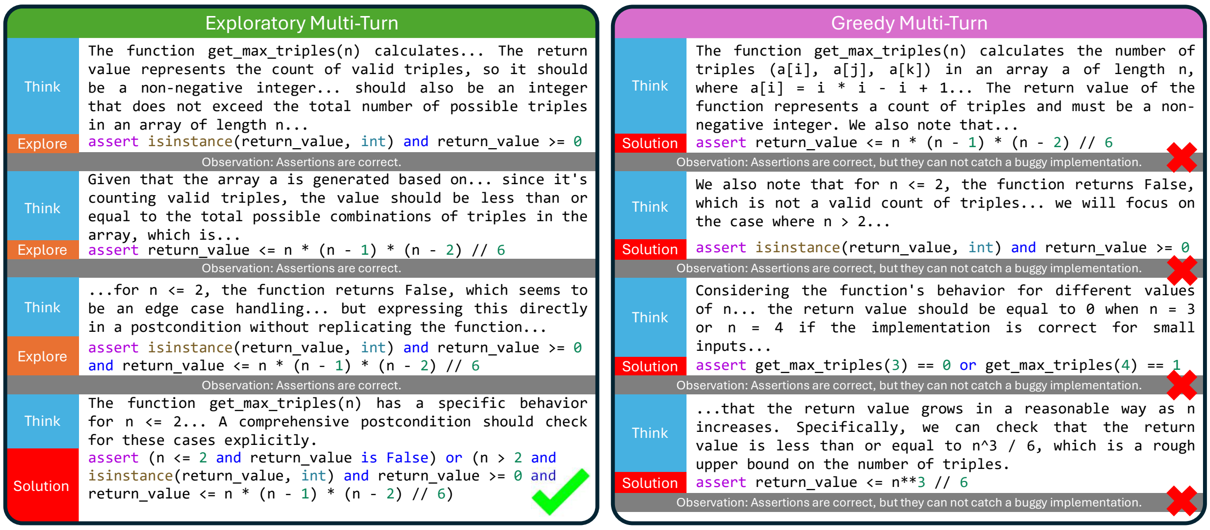


Figure 7: Trend of completeness score from our seleted cases in *EvalPlus*, using Exploratory Multi-turn with configuration τ=90 and μ=12

The selected tasks require between two and four submission attempts. All eight cases show an increase in completeness from the first submission to the final submission that reaches the completeness threshold. Across all 164 tasks, we observe that 23 cases show no change in completeness between consecutive submissions (e.g., case #90 and case #161), indicating early convergence. In contrast, 18 cases temporarily drop to zero completeness due to the submission of an incorrect postcondition before recovering in later attempts (e.g., case #10, not shown). The earliest correct submission occurs at attempt 3 in 11 cases (e.g., case #112), suggesting that Exploratory Multi-turn often identifies a correct core postcondition early and incrementally improves its discriminative power.

A.4 A Case Study of Exploratory vs. Greedy Multi-turn

To illustrate the behavioral differences between Greedy and Exploratory Multi-turn, Fig. 8 presents



Exploratory Multi-Turn

Think

The function get_max_triples(n) calculates... The return value represents the count of valid triples, so it should be a non-negative integer... should also be an integer that does not exceed the total number of possible triples in an array of length n...

Explore

```
assert isinstance(return_value, int) and return_value >= 0
```

Observation: Assertions are correct.

Think

Given that the array a is generated based on... since it's counting valid triples, the value should be less than or equal to the total possible combinations of triples in the array, which is...

Explore

```
assert return_value <= n * (n - 1) * (n - 2) // 6
```

Observation: Assertions are correct.

Think

...for n <= 2, the function returns False, which seems to be an edge case handling... but expressing this directly in a postcondition without replicating the function...

Explore

```
assert isinstance(return_value, int) and return_value >= 0
and return_value <= n * (n - 1) * (n - 2) // 6
```

Observation: Assertions are correct.

Think

The function get_max_triples(n) has a specific behavior for n <= 2... A comprehensive postcondition should check for these cases explicitly.

Solution

```
assert (n <= 2 and return_value is False) or (n > 2 and
isinstance(return_value, int) and return_value >= 0 and
return_value <= n * (n - 1) * (n - 2) // 6)
```

✓

Greedy Multi-Turn

Think

The function get_max_triples(n) calculates the number of triples (a[i], a[j], a[k]) in an array a of length n, where a[i] = i * i - i + 1... The return value of the function represents a count of triples and must be a non-negative integer. We also note that...

Solution

```
assert return_value <= n * (n - 1) * (n - 2) // 6
```

Observation: Assertions are correct, but they can not catch a buggy implementation. ✗

Think

We also note that for n <= 2, the function returns False, which is not a valid count of triples... we will focus on the case where n > 2...

Solution

```
assert isinstance(return_value, int) and return_value >= 0
```

Observation: Assertions are correct, but they can not catch a buggy implementation. ✗

Think

Considering the function's behavior for different values of n... the return value should be equal to 0 when n = 3 or n = 4 if the implementation is correct for small inputs...

Solution

```
assert get_max_triples(3) == 0 or get_max_triples(4) == 1
```

Observation: Assertions are correct, but they can not catch a buggy implementation. ✗

Think

...that the return value grows in a reasonable way as n increases. Specifically, we can check that the return value is less than or equal to n^3 / 6, which is a rough upper bound on the number of triples.

Solution

```
assert return_value <= n**3 // 6
```

Observation: Assertions are correct, but they can not catch a buggy implementation. ✗

Figure 8: A comparison between Exploratory Multi-turn and Greedy Multi-turn for case #147 from *EvalPlus*: the figure illustrates SPECMIND's outputs (τ=50 and μ=4), model's reasoning are given in the blue blocks, submission attempts are given in the red blocks, and exploration attempts are enclosed in the orange blocks. The feedback is shown in the gray boxes. ✓: correct and complete postcondition, ×: otherwise.

a detailed analysis of case #147 from EvalPlus under $\tau = 50$ and $\mu = 4$. The task defines the function `get_max_triples(n)`, which computes the number of valid triples in an array a of length n, where $a[i] = i \times i - i + 1$, such that the sum of the triples is divisible by 3. If $n \leq 2$, the function returns `False`.

Exploratory Multi-turn submits its final solution at attempt #4, whereas Greedy Multi-turn submits a candidate postcondition at every attempt by design, with its best-performing one appearing in attempt #1. In the first attempt, both approaches identify two key properties: (1) the return value should be a non-negative integer, and (2) it should be bounded by the total number of possible triples. Exploratory probes the first property through an exploratory assertion, while Greedy directly submits the second, more complex constraint.

In the second attempt, Exploratory explores the second property and explicitly notes its potential relevance. Greedy, however, reverts to the first property and briefly mentions the case $n \leq 2$ in its reasoning but dismisses it as irrelevant, despite its semantic importance.

In the third attempt, Exploratory combines the two properties and explicitly incorporates the $n \leq 2$ case. In contrast, Greedy submits assertions targeting specific input values (e.g., $n = 2$, $n = 3$). Notably, its reasoning states `get_max_triples(n)=0`, while the submitted assertion enforces `get_max_triples(n)=1`; this mismatch happens to pass tests but reflects unstable reasoning.

In the final attempt, Exploratory refines its earlier reasoning by adding the missing $n \leq 2$ sub-condition and submits a complete postcondition. Greedy instead submits a new constraint, `return_value <= n**3 // 6`, which is weaker than a stricter constraint it had already produced earlier, resulting in a correct yet incomplete specification.

Overall, Exploratory Multi-turn produces a correct and complete postcondition using a single submission and three exploratory steps. It systematically explores sub-conditions and composes them into a final specification. In contrast, Greedy Multi-turn identifies a partially correct condition early but fails to refine it into a comprehensive postcondition within the same budget.

A.5 Example Trajectory of Reasoning Categories

Table 4 presents a representative reasoning trajectory produced by Exploratory Multi-turn for case #159 from the EvalPlus dataset. This task requires implementing the function `eat`, which computes the total number of carrots eaten and the number of carrots remaining, given `number` (already eaten carrots), `need` (carrots to be eaten), and `remaining` (carrots in stock).

The trajectory begins with *Verify Function Behavior*, where the model describes the function's overall behavior and proposes assertions checking basic structural properties of the return value.

Table 4: Exploratory Multi-turn's trajectory for function #159 in EvalPlus dataset. It requires implementing the function `eat`, which calculates the total number of carrots eaten and the number of carrots remaining, given `number` (carrots already eaten), `need` (carrots that need to be eaten), and `remaining` (carrots in stock).

<table>
<tr><th>Att.</th><th>Category</th><th>Reasoning (ret=return_value, rem=remaining, num=number.)</th></tr>
<tr><td>1</td><td>Verify Func. Behavior</td><td>The function ‘eat‘ calculates... The function returns... Key constraints and edge cases:... To verify the correctness of the function, we need to write symbolic postconditions that check these properties.
`» assert isinstance(ret, list) && len(ret) == 2 && isinstance(ret[0], int) && isinstance(ret[1], int)`</td></tr>
<tr><td>2</td><td>Check Edge Cases</td><td>The first assertion checks... However, we need to add more assertions to cover other properties of the function. Let's consider the case when ‘need‘ is less than or equal to ‘remaining‘... We should also consider the case when ‘need‘ is greater than ‘remaining‘...
`» assert need <= rem || (need > rem && ret[1] == 0)`</td></tr>
<tr><td>3</td><td>Refine Assertions</td><td>The second assertion checks if... should also check if the total number of carrots eaten is correct in both...
`» assert (need <= rem && ret[0] == num + need) || (need > rem && ret[0] == num + rem)`</td></tr>
<tr><td>4</td><td>Postcondition Combining</td><td>... let's try to combine them into a single assertion...
`» assert (need <= rem && ret[1] == rem - need) || (need > rem && ret[1] == 0)`
`» assert (need <= rem && ret[0] == num + need) || (need > rem && ret[0] == num + rem)`</td></tr>
<tr><td>5</td><td>Refine Assertions</td><td>... let's refine our assertions to make them more concise and meaningful...
`» assert (need <= rem && ret[0] == num + need && ret[1] == rem - need) || (need > rem && ret[0] == num + rem && ret[1] == 0)`</td></tr>
<tr><td>6</td><td>Submit Final Solution</td><td>We have several assertions that check different properties of the function... Let's try to finalize our solution by submitting a single assertion that combines all the checks...
`» assert (need <= rem && ret == [num + need, rem - need]) || (need > rem && ret == [num + rem, 0])`</td></tr>
</table>

It then transitions to *Check Edge Cases*, explicitly reasoning about conditional scenarios such as whether `need` exceeds `remaining`. Next, the model enters *Refine Assertions*, improving earlier checks by strengthening semantic constraints on the return values.

The reasoning subsequently moves to *Postcondition Combination*, where multiple previously explored assertions are consolidated. This is followed by another *Refine Assertions* step, reflecting iterative improvement toward a more concise and comprehensive specification. Finally, the trajectory concludes with *Submit Final Solution*, where the model submits a single assertion that captures all explored conditions.

Overall, this example illustrates a systematic progression through reasoning categories, starting from high-level function understanding, moving through edge case analysis and iterative refinement, and culminating in the submission of a complete postcondition.

A.6 Usage of LLM Assistance

We used large language models (LLMs) to aid in polishing the writing and improving clarity. All research ideas, experiments, and conclusions are the work of the authors.