Can Large Language Models Transform Natural Language Intent into Formal Method Postconditions?

MADELINE ENDRES*, University of Michigan, USA
SARAH FAKHOURY, Microsoft Research, USA
SAIKAT CHAKRABORTY, Microsoft Research, USA
SHUVENDU K. LAHIRI, Microsoft Research, USA

Informal natural language that describes code functionality, such as code comments or function documentation, may contain substantial information about a program's intent. However, there is typically no guarantee that a program's implementation and natural language documentation are aligned. In the case of a conflict, leveraging information in code-adjacent natural language has the potential to enhance fault localization, debugging, and code trustworthiness. In practice, however, this information is often underutilized due to the inherent ambiguity of natural language, which makes natural language intent challenging to check programmatically. The "emergent abilities" of Large Language Models (LLMs) have the potential to facilitate the translation of natural language intent to programmatically checkable assertions. However, it is unclear if LLMs can correctly translate informal natural language specifications into formal specifications that match programmer intent. Additionally, it is unclear if such translation could be useful in practice.

In this paper, we describe *nl2postcond*, the problem of leveraging LLMs for transforming informal natural language to formal method postconditions, expressed as program assertions. We introduce and validate metrics to measure and compare different *nl2postcond* approaches, using the correctness and *discriminative power* of generated postconditions. We then use qualitative and quantitative methods to assess the quality of *nl2postcond* postconditions, finding that they are generally correct and able to discriminate incorrect code. Finally, we find that *nl2postcond* via LLMs has the potential to be helpful in practice; *nl2postcond* generated postconditions were able to catch 64 real-world historical bugs from *Defects4J*.

CCS Concepts: • **General and reference** → *Metrics*; Validation; • **Software and its engineering** → *Correctness*; *Completeness*; *Software reliability*; *Software verification*; *Formal software verification*; • **Computing methodologies** → Artificial Intelligence.

Additional Key Words and Phrases: Large Language Models, Postconditions, Formal Specifications

ACM Reference Format:
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, Article 84 (July 2024), 24 pages. https://doi.org/10.1145/3660791

1 INTRODUCTION

Informal natural language specifications are omnipresent in modern software. For example, Pfeiffer [40] found natural language documentation in 98% of over 20,000 GitHub repositories, with 10%

*Work done while interning at Microsoft.

Authors' Contact Information: Madeline Endres, University of Michigan, Ann Arbor, MI, USA, endremad@umich.edu; Sarah Fakhoury, Microsoft Research, Redmond, WA, USA, sfakhoury@microsoft.com; Saikat Chakraborty, Microsoft Research, Redmond, WA, USA, saikatc@microsoft.com; Shuvendu K. Lahiri, Microsoft Research, Redmond, WA, USA, shuvendu@microsoft.com.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2024 Copyright held by the owner/author(s).
ACM 2994-970X/2024/7-ART84
https://doi.org/10.1145/3660791

of repository artifacts specifically for documentation. He [17] found over 20% of non-blank program lines contained in-file comments in their study of 150 of the most starred projects on GitHub. At the same time, it is well known that software bugs (unexpected exceptions, incorrect output) often arise from the weak association between the intended behavior (documented in natural language) and the behavior of the implementation [47, 49]. This issue is exacerbated with AI-assisted programming where users generate code from natural language intent [2, 15, 46], without a good way to ensure their association. Reliably translating informal natural language descriptions to formal specifications could help catch bugs before production and improve trust in AI-generated code [24].

Current approaches to translating natural language to formal specifications are heuristic-based and either rely on the input being in a structured format [3, 49] or can only generate a restricted class of specifications (e.g., regarding nullness or exceptions) [16, 47]. Further, most of these approaches are customized for only one specific programming language (such as Java). In the past, large-scale neural modeling for the problem of generating specifications has been difficult given the absence of large code corpora with matching natural language intent and corresponding specifications.

Large Language Models (LLMs) have generated interest in programming due to their ability to synthesize high-quality code from natural language intent in a surrounding context [6, 29, 34]. Given the limitations of current approaches for translating natural language to formal specifications, we explore the use of LLMs for this problem. Even though LLMs have not seen structured data matching natural language intent to specifications, larger models such as `GPT-4` have demonstrated "emergent abilities" to do well on tasks that they were not explicitly trained for [54]. This includes the capability to follow natural language instructions to perform reasoning tasks through prompting strategies like few-shot learning [25], chain-of-thought [55] and multi-step reasoning [58].

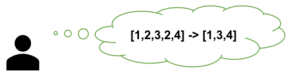
In this paper, we explore the feasibility of leveraging LLMs as a bridge between informal natural language and method postconditions (we term this approach *nl2postcond*). A *method postcondition* is an assertion that relates the method's input and output states, and holds true after any successful method execution. We assess this ability in a programming language-agnostic way, targeting postconditions expressed as assertions in the underlying programming language.

1.1 Motivating Examples

1.1.1 Formalizing User Intent. Consider the example in Fig. 1, adapted from the Python code generation benchmark, *HumanEval* [6]. A programmer intends to remove all numbers with duplicates from a list. For example, given the list `[1,2,3,2,4]` the function should return `[1,3,4]` without `2` as it appears more than once. The programmer describes the this behavior in a docstring (see Fig. 1b). However, the natural language specification is ambiguous; it does not indicate if all copies of a duplicated element should be removed, or if one copy should be retained. Here, the programmer intends the former, however, it is not uncommon to expect that the program should fulfill the latter.

Figure 1c shows two postconditions, one satisfying each possible intent. The programmer can verify that the second postcondition "`assert all(numbers.count(i)==1 for i in return_list)`" *correctly* matches their intent, by ensuring that all numbers in `return_list` occur exactly once in the input list. The first postcondition, however, incorrectly asserts that `return_list` is a set of the input list. As it is not always clear at first glance what the user intent is, generating such postconditions from natural language allows for checkable and unambiguous statements about a program's intended behavior, formalizing a user's intent. User-validated specifications can also be useful to prune incorrect code suggestions in interactive code generation settings [12, 24].

1.1.2 Detecting Real-World Functional Bugs. In practice, postconditions generated in the target programming language can be used in assertions, as demonstrated in fig. 1c, to check the correctness of a function, enabling the early detection of bugs or violations of a programmer's intent.



(a) Programmer intent for a function that removes all instances of numbers that have duplicates from a list.

```
def remove_duplicates(numbers: List[int]):
  """ From a list of integers, remove all elements that occur more than
      once. Keep order of elements left the same as in the input """
```

(b) *Ambiguous* natural language specification: it does not specify if all copies or all but one copy of a duplicated element should be removed. In this case, the programmer intends the former.

`assert len(set(numbers)) == len(set(return_list))`	✗
`assert all(numbers.count(i) == 1 for i in return_list)`	✓

(c) Postconditions generated by GPT-4. Note that while both could be correct with a literal reading of the natural language specification, only the second one is correct with respect to developer intent.

Fig. 1. Example of how postconditions could be used to clarify ambiguous natural language specifications.

```
/** @return a new line with reversed direction.*/
public Line revert() {
  final Line reverted = new Line(zero, zero.subtract(direction));
  return reverted; }
```

(a) Original function plus java doc.

> Line.revert() only maintains ~10 digits for the direction. This becomes an issue when the line's position is evaluated far from the origin.

(b) Bug report (not seen by the LLM.)

```
// Correct bug—finding postcondition 1
assert this.direction.negate().equals(returnValue.direction);
// Correct bug—finding postcondition 2
assert this.direction.dotProduct(returnVal.direction) == —1 * this.direction.getNormSq();
```

(c) Two bug-catching post conditions generated by GPT 4.

Fig. 2. Example of how postconditions or other formal specifications could catch bugs. This example is a historical bug from *Defects4J* (Math-9): the `Line` constructor does not return a new line with enough precision. The postconditions were generated by `GPT-4` in our evaluation, and both catch the bug.

The example in Figure 2 shows how formal specifications can be used to catch bugs in real-world programs. A bug from the `Apache Commons Math` project, the function `revert()` calls a constructor `Line()` that should return a new `Line` object with a reversed direction. The associated bug report [1] explains that `revert()` does not maintain enough precision, and fails in certain scenarios. Both of the provided postconditions in Figure 2c catch the bug by leveraging project-specific context and general mathematical knowledge about the specifications of a reversed line.

1.2 Overview

In these examples, we demonstrated that `GPT-4` can generate postconditions from natural language that closely capture informal intent and also detect program bugs. However, it is unclear to what extent LLMs are capable of the *nl2postcond* problem in general. We pose the high-level question:

> *Given a natural language description of a method and a candidate postcondition, how do we judge the quality of the postcondition?*

[1] https://issues.apache.org/jira//browse/MATH-938

We attempt to study this question through two high-level research questions:

- RQ1: How well do LLM-generated postconditions formalize informal natural language intent?
- RQ2: Can LLM-generated postconditions help catch real-world bugs?

To answer these questions, we define automated metrics for measuring the usefulness of LLM-generated postconditions, describe different ways to encode the problem statement for an LLM, explore different LLMs, and perform an empirical investigation (both quantitative and qualitative) on benchmarks across multiple programming languages. We first define automated evaluation metrics for the correctness and completeness (i.e., the discriminative power) of a postcondition (Section 2.1), and we propose a generic "prompt" and variants to transform an informal intent into an input for LLM (Section 2.2). We evaluate RQ1 using a Python programming dataset and present a detailed analysis of generated postconditions quality across different LLMs and prompt variants (Section 3). Next, we evaluate RQ2 on a benchmark of real-world Java defects and report on the ability of postconditions to find bugs by distinguishing the fixed version from the buggy version (Section 4). Finally, we articulate the limitations (Section 6) and discuss related works (Section 5).

1.3 Contributions

- Evaluating the feasibility of LLMs to facilitate *nl2postcond* via an empirical evaluation of the quality and usefulness of LLM generated postconditions on multiple benchmarks in multiple mainstream programming languages.
- Automated and semantics-based metrics (both correctness and completeness) for evaluating natural language generated postconditions, validated through an empirical and qualitative investigation. In particular, we believe this paper is the first to propose the use of LLMs to derive a natural distribution of *code mutants* to evaluate the completeness of specifications.
- The finding that with sufficiently robust natural language descriptions, LLMs can use *nl2postcond* to generate *correct* postconditions with high *discriminative power*. We illustrate that with `GPT-4` we can generate correct postconditions for up to 96% problems in the *EvalPlus* benchmark, with correct postconditions able to discriminate up to 81% of distinct buggy programs on average.
- The finding that LLM-generated *nl2postcond* postconditions are precise enough to capture real-world bugs in large industrial projects; *nl2postcond* postconditions detect 64 historical bugs from 70 buggy methods in industrial-scale Java projects.

2 *NL2POSTCOND*: OVERALL APPROACH

2.1 Problem Formulation and Metrics

We first formalize the *nl2postcond* problem through metrics to evaluate the quality of generated postconditions. Consider an example $\langle nl, r, T\rangle$, where nl is the natural language description of a problem, r is a reference code implementation, and T is a set of test inputs. For this section, we assume that each test $i \in T$ is an input that assigns a value to the input parameters and globals of r. We further assume that the reference solution is deterministic and returns a single output value *ret* containing the output. In this simple setting, it suffices to only have the set of inputs in T, as the desired output for each input i can be obtained by executing $r(i)$. For the purpose of postcondition generation through an LLM, the set of tests T is *hidden* from the LLM that generates a postcondition. The reference implementation r may or may not be present during the postcondition generation. However, both r and T are used to define the metrics for the offline evaluation for a benchmark set.

2.1.1 Test-set Correctness. Given an example $e \doteq \langle nl, r, T\rangle$, a candidate postcondition *post* is an assertion over the input and output states of r. For an expression *expr* and a state s that assigns valuation to variables, let $eval(expr, s)$ be the result of evaluating *expr* after replacing the variables

in *expr* with their values from s. A postcondition is *correct* if the reference implementation r satisfies it for every possible (legal) input. Therefore, a candidate postcondition *post* is correct if for every input i, if $r(i)$ is the output value, then $eval(post, (i, r(i)))$ is true, where $(i, r(i))$ is the joint state of the input parameters and output return variable. However, such a notion of correctness is difficult to establish in the absence of formal verification tools, and may further require manual effort to establish such proof even for verification-aware languages [26, 45]. We take a pragmatic approach, assuming that the test cases in T are sufficiently comprehensive to approximate the space of all legal inputs. Therefore, an expression *post* is *test-set-correct* w.r.t. T (denoted as $correct_T$) iff $\forall i \in T : eval(post, (i, r(i))) == true$. Henceforth, we may refer to "test-set-correct" as simply correct, since correctness in the remainder of the paper is with respect to the provided tests.

Given a set of m postconditions from an LLM, we define a metric $accept@k$ for $1 \leq k \leq m$ to capture the statistical expected value of containing at least one test-set-correct postcondition while sampling subsets of size k from the set of m conditions. This is inspired by the $pass@k$ metric proposed for evaluating the quality of correctness of generated code given a set of tests [6].

2.1.2 Test-set Completeness for Code Mutants: Bug-Completeness-Score. (Test-set) correctness is a *necessary* condition for a valid and useful postcondition, however, it is not *sufficient*. For example, the expression *true* vacuously satisfies any implementation r for any input $i \in T$, and is therefore correct. The value of a postcondition comes from how well it captures the desired intent expressed in the natural language intent *nl*. However, given that *nl* is informal, we cannot establish a check to ensure the association. Instead, we leverage the reference implementation and tests as the source of *ground truth* for what the user intends. However, this again poses the problem that the most desired postcondition is simply the *strongest postcondition* of r program, which is computationally intractable [8]. Instead, we use a concept of *completeness* that measures the degree to which the postcondition distinguishes the reference implementation r from other incorrect implementations.

Inspired by mutation-testing literature (c.f. Jia and Harman [20]) that assigns a score to a test t based on the fraction of code mutants "killed" or distinguished under t, we assign a measure of *bug-completeness* to a postcondition *post* as the fraction of code mutants that can be distinguished given the set of tests T. Unlike traditional mutation testing, we parameterize completeness with a *semantically distinct* code mutant set CM that are guaranteed to differ from r (and from each other) on at least one test in T. In other words, for each $c \in CM$, there exists a test input $i \in T$ that distinguishes from r (i.e., $r(i) \neq c(i)$) and (a possibly different) i that distinguishes from any other $c' \in CM \setminus \{c\}$ (i.e., $c(i) \neq c'(i)$). Given an example $e \doteq \langle nl, r, T \rangle$, a $correct_T$ postcondition *post* and a set of distinct code mutants CM, we define the *bug-completeness-score* of *post* as:

$$bug\text{-}completeness\text{-}score(post, CM, T) \doteq |\{c \in CM \mid \exists i \in T : eval(post, (i, c(i))) == false\}|/|CM|$$

That is, *bug-completeness-score* measures the fraction of distinct code mutants that fail the correct postcondition (via associated distinct buggy input/output tests). If the *bug-completeness-score* of a postcondition is 1, we say that the postcondition is *bug-complete*. The metric can be lifted to a set of postconditions P via the union of all code mutants "killed" by all correct postconditions in the set:

$$bug\text{-}completeness\text{-}score(P, CM, T) \doteq |\bigcup_{post \in P} \{c \in CM \mid \exists i \in T : eval(post, (i, c(i))) == false\}|/|CM|$$

We propose the use of LLMs to sample mutants from the natural distribution of implementations to the problem described by the natural language intent *nl*. In other words, we enumerate a set of likely implementations *Impls* for *nl* using a LLM (GPT-3.5), and define CM to be the subset of *Impls* that differ from r on at least one test $i \in T$, using the tests in T that are pairwise distinct. We now discuss why we use a parameterized set of code mutants instead of creating variants of r by mutating different operators. We believe that such a fixed set of mutation operators does not

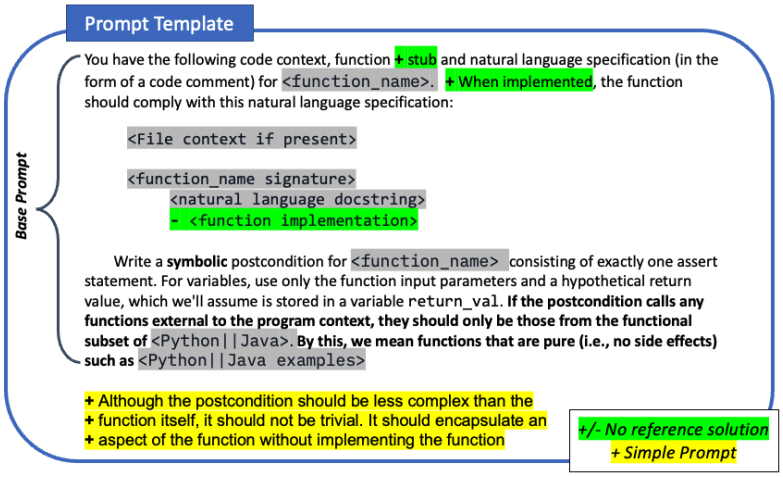


Fig. 3. Prompt template for generating postconditions from natural language via chat models (including changes for the `simple` and *no reference* variations). We found that the **bold** text greatly improved postcondition quality: without it, the model tended to return point-based tests or code blocks with side effects. While modified here slightly for clarity, our full prompts are included in our associated materials (see Section 8.)

approximate real-world bugs for two reasons: (a) first, since code mutants only differ from the reference implementation in one or two operators at a time, it may not cover mutations that are further away in the edit distance, and (b) it may not cover subtle bugs that a human would write using different syntactic constructs (e.g., a while loop instead of a for loop) or APIs.

2.2 Prompt Design for LLM-based Postcondition Generation

LLM performance has been shown to be impacted by small changes in prompts for the same problem task, and designing the optimal prompt is not always straightforward. We explore several prompt templates, i.e. varied textual representations of the problem description *nl*, and reference solution *r*, optimizing for a number of outcomes. First, the prompts should work with *chat-based models*, and the generated postconditions should be *symbolic* (e.g., not point-wise tests), directly executable, and side-effect free. Also, the prompt should encourage the LLM to produce expressions that are syntactically and semantically valid while being as programming language agnostic as possible. Several prompt iterations were considered until we observed satisfactory performance on a subset of example problems, though we acknowledge further prompt tuning may result in different outcomes. Figure 3 outlines our prompt template. This template shows four possible prompt iterations along two orthogonal axes (a) whether the reference *r* is included, and (b) requested postcondition complexity. We now discuss each axis in more detail.

Including Reference Code: Our default prompt includes only the *nl*, not the reference code *r*. This is useful for *specification-driven AI-based programming* scenarios [24] where the user first accepts a few specifications that are used to constrain AI generated code suggestions. However, we also provide a prompt variant that includes the reference code *r* along with *nl*. This allows us to assess if natural language alone can be as effective as code in conveying programming intent to an LLM.

Postcondition Complexity: we also consider a `simple` prompt variation that explicitly instructs the LLM to generate postconditions that capture an aspect of a function, rather than the whole function. When using the `base` prompt, LLMs have a tendency to construct complex postconditions,

```
Given a string text, replace all spaces in it with underscores, and if a
string has more than 2 consecutive spaces, then replace all consecutive
spaces with - . For example: fix_spaces(“ Example 1”) == “_Example_1”,
fix_spaces(“ Example   2”) == “_Example-2”
```

(a) Informal natural language specification for problem 12 from *HumanEval*

```
# Base prompt: postcondition that incorrectly attempts to fully specify the problem
assert all(map(lambda x: x == "_" or x == "-", re.split(r'\w+', return_value))) and " " not in
      return_value and "__" not in return_value and "--" not in return_value

# Simple prompt: postcondition that correctly checks that return_value does not contain any spaces.
assert not re.search(r' {1,}', return_value), "The return value contains one or more spaces"
```

(b) `base` vs. `simple`: the `base` postcondition tries to capture all intended functionality, but does so incorrectly. The `simple` postcondition is less complex (capturing less functionality), but is correct.

Fig. 4. Example of how the `base` and `simple` prompt variations can impact postcondition construction. Both postconditions were generated for *HumanEval* problem 12 using `GPT-4`.

often approaching a fully functional implementation of the problem. While useful, we observe these postconditions are likely to be incorrect. To illustrate this, Fig. 4 compares postconditions produced by the `base` and `simple` prompt for a problem from the *HumanEval* benchmark.

We combine these two prompt variations into four distinct prompts in our evaluation:

(1) `Base` prompt with only natural language description *nl* (no reference solution *r*)
(2) `Base` prompt with both reference *r* and natural language description *nl*
(3) `Simple` prompt with only natural language description *nl* (no reference solution *r*)
(4) `Simple` prompt with both the reference *r* and natural language description *nl*

3 RQ1: HOW WELL DO LLM-GENERATED POSTCONDITIONS FORMALIZE INFORMAL NATURAL LANGUAGE INTENT?

To assess if LLMs can generate high-quality postconditions that capture and formalize intent, we report a detailed empirical study of LLM-generated postconditions on a popular benchmark.

3.1 RQ1 Experimental Setup

3.1.1 Evaluation Benchmark. We use the benchmark *EvalPlus*. It has 164 Python problems, each with a function stub, natural language description, reference implementation, and validation tests [30]. *EvalPlus* updates the popular *HumanEval* benchmark [6], containing the same problems but with a more extensive test suite (775 tests per problem on average). We choose *EvalPlus* because each example has (a) a descriptive natural language intent, (b) a set of extensive test inputs, and (c) a reference solution. Using these three components, we can evaluate if a postcondition formalizes the user intent expressed in the natural langauge, *nl*, while also satisfying the reference solution.

3.1.2 Large Language Models. We generate postconditions using three chat-based models (both closed and open-source), that have shown state-of-the-art performance on programming tasks:

- *OpenAI:* ***`GPT-3.5`*** *and* ***`GPT-4`*** are based on the pre-trained GPT-3 model, which is fine-tuned using Reinforcement Learning with Human Feedback (RLHF) [37]. While `GPT-3.5` and `GPT-4` are not explicitly fine-tuned for code generation, they have demonstrated strong capabilities on several related tasks [11, 35]. We use OpenAI APIs for the `gpt-3.5-turbo` and `gpt-4` endpoints.
- *The BigCode Project:* ***`StarChat`***. StarCoder [29] is an open-access 16B parameter model pre-trained on The Stack [23], one trillion tokens sourced from 80+ programming languages, GitHub

issues, Git commits, and Jupyter notebook. We use *StarChat* [2], a StarCoder version fine-tuned for helping coding. StarChat is one of the few open-access chat model alternatives to `GPT-3.5` and `GPT-4`, permitting replication of and comparison with our results. This model allows us to use the same prompt we used for the OpenAI models, rendering a fairer comparison.

3.1.3 Postcondition Generation. For each *EvalPlus* problem, we generate 10 postconditions for each of the 4 prompt variants (Section 2.2) per LLM model. We use a temperature of 0.7 as it is the default for both `GPT-3.5` and `GPT-4`, and has been found to be a reasonable temperature for code generation tasks.[3] As we consider four prompt variants, we generate 40 postconditions per problem per model. This results in 19,680 postconditions across all variants, models, and *EvalPlus* problems.

3.1.4 Code Mutant Generation. To generate the set of code mutants *CM* needed for *bug-completeness*, we use an LLM (`GPT-3.5` with temperature 0.9) to generate a set of codes that satisfy the natural language intent *nl*. We generate 200 code solutions to each problem and then save only those that produce a bad output for at least one test in T. We term these bugs as *natural* unique code mutants, as they represent natural yet buggy implementations for the problem description. However, we noticed that for some examples, the number of such natural code mutants is fairly small. To amplify the set of buggy codes, we generate 100 additional buggy codes by explicitly instructing `GPT-3.5` to include an error in its solution. As mentioned in Section 2.1.2, we only retain distinct buggy codes so that no two mutants fail the same tests in the same way. That is, we only retain mutants and associated inputs that contribute distinct buggy input/output pairs. The number of unique buggy codes varies per problem, ranging from 4 to 233 with a median of 55. While we combine the two bug sources, we also consider the *natural mutants* alone in our evaluation to see if the source of the bug impacts the efficacy of our metrics. We make available the set of mutants for use by the broader research community and to support the reproducibility of our results.

3.2 RQ1-Results: Do LLM-generated Postconditions Formalize User Intent?

We now discuss the results of our empirical and qualitative evaluations, structured around postcondition correctness, postcondition completeness, and qualitative insights.

3.2.1 Postcondition Correctness. Table 1 has our *test-set-correctness* (Section 2.1.1) results. Overall, we find that for *EvalPlus*, LLM-generated postconditions are likely to be test-set correct; in our best-performing prompt variation, 77% of postconditions were test-set-correct and a test-set-correct postcondition was generated for 96% of problems (158/164). As we show later in this section, test-set-correctness on *EvalPlus* largely corresponds to true correctness. Our results indicate that LLMs have the potential to reliably generate correct postconditions from natural language specifications.

Regardless of the prompt variation, `GPT-4` postconditions were the most likely to be correct ($0.63 \leq$ `accept@1` ≤ 0.77) followed by `GPT-3.5` ($0.46 \leq$ `accept@1` ≤ 0.56). `StarChat` postconditions were consistently the least correct, with `accept@1` between 0.21 and 0.25. While the raw number of correct `StarChat` postconditions was low, the number of benchmark problems with at least one correct postcondition was relatively high, ranging from 77% to 86% depending on the prompt.

As described in section 2.2, we consider both a `base` postcondition prompt and a `simple` prompt for generating simpler postconditions that capture only an aspect of program behavior. Regardless of LLM model, `simple` postconditions are more likely to be correct than `base` postconditions. Using a paired students t-test [18] between `accept@1` ablation pairs where the only difference is the prompt complexity type, `simple` prompt postconditions are significantly more likely to be correct

[2] HuggingFace model identifier `HuggingFaceH4/starchat-alpha`

[3] We also considered temperatures of 0.2 and 1.2, finding high-level trends were the same regardless of temperature.

Table 1. Test-set correctness on *EvalPlus* for three models (`GPT-3.5`, `GPT-4`, and `StarChat`), differing prompt complexities (`base` vs. `simple`), and including or omitting the reference solution in the prompt. Darker highlighted cells are more correct. Bolded values are the largest for a specific model.

Model	Prompt	Prompt has: NL Only=✗ ref code=✓	Accept @ 1	Accept @ 5	Accept @ 10	x/164 correct
GPT-3.5	base	✗	0.46	0.80	0.87	143
GPT-3.5	base	✓	0.49	0.81	**0.88**	**145**
GPT-3.5	simple	✗	0.55	**0.82**	0.87	143
GPT-3.5	simple	✓	**0.56**	**0.82**	**0.88**	144
GPT-4	base	✗	0.63	0.83	0.88	144
GPT-4	base	✓	0.71	0.89	0.91	150
GPT-4	simple	✗	**0.77**	**0.94**	**0.96**	**158**
GPT-4	simple	✓	0.76	0.92	**0.96**	157
StarChat	base	✗	0.21	0.61	0.82	134
StarChat	base	✓	0.20	0.59	0.77	126
StarChat	simple	✗	**0.25**	**0.69**	0.85	139
StarChat	simple	✓	0.23	0.67	**0.86**	**141**

with $p = 0.008$, a large effect (standardized Cohen's $d = 1.73$). This indicates that when prioritizing correctness, using a prompt that explicitly asks for simpler postconditions improves the result.

We also compared the efficacy of generating postconditions from natural language alone to generating when a reference solution is included in the prompt. We did not observe a significant difference in `accept@1` between postconditions generated with natural language specifications alone and those including a reference solution ($p = 0.42$). This indicates that the presence of a reference solution does not necessarily enhance postcondition correctness when compared to natural language alone. Therefore, in some situations, it might be feasible to rely solely on natural language intent (when comprehensive enough) without needing to provide a reference solution.

Correctness False Positives. EvalPlus has more comprehensive tests than its predecessor *HumanEval*, but it is still possible that the tests do not capture all possible inputs. If so, our test-set correctness metric may have false positives. We validate our metric for *EvalPlus*: we find only one problem (# 122) with false positives relating to negative inputs. Overall, only 1.1% of the 900 postconditions that we manually annotated were affected (see Section 3.2.3 for our annotation process). In contrast, we also compare our results to the hypothetical results if using *HumanEval* (which has the same problems, but many fewer tests). The *HumanEval* results contain 7% false positives for `accept@10` for `GPT-4`, much higher than our results. Thus, we find that test-set correctness is a reasonable approximation for true correctness on *EvalPlus* (but perhaps less so for *HumanEval*).

RQ1 Summary: Postcondition Correctness on *EvalPlus*

On *EvalPlus*, LLMs can produce *correct* postconditions from informal natural language specifications. All prompt variants generate a correct postcondition for at least 77% and up to 96% of problems. `GPT-4` outperformed `GPT-3.5` and `StarChat`. Asking for `simple` postconditions leads to more correct postconditions, but including a reference solution does not; using (descriptive) natural language alone can be sometimes just as powerful.

3.2.2 Postcondition Completeness. While our test-set correctness results are encouraging, test-set correctness is necessary but not sufficient for assessing if a postcondition meaningfully captures the natural language specification. To capture a notion of completeness, we measure *bug-completeness*

Table 2. Table of bug-completeness for *EvalPlus*. `% bug-complete` is the % of postconditions that detect all buggy code mutants. `% problems with bug-complete` is the % of *EvalPlus* problems with at least one `bug-complete` postcondition. `% problems union bug-complete` is the % of problems where the union of correct postconditions is `bug-complete`. The last two columns are the average `bug-completeness-score`, a fraction between 0 and 1, for all correct postconditions, normalized by *EvalPlus* problem. We report this for both *natural* and *all* generated code mutants. Bolded values are the largest value per column per model.

Model	Prompt	Prompt has: NL Only=✗ ref code=✓	% bug-complete	% problems with bug-complete	% problems union bug-complete	Avg bug-completeness-score for correct postconditions	
						Natural bugs	*All bugs*
GPT-3.5	base	✗	15.4	42.1	48.2	0.62	0.72
GPT-3.5	base	✓	**18.5**	**47.0**	**49.4**	**0.70**	**0.76**
GPT-3.5	simple	✗	8.1	29.3	33.5	0.44	0.55
GPT-3.5	simple	✓	14.0	37.2	41.5	0.58	0.62
GPT-4	base	✗	**35.1**	**61.6**	**62.2**	**0.81**	**0.85**
GPT-4	base	✓	34.9	58.0	61.6	0.78	0.82
GPT-4	simple	✗	9.2	26.2	29.3	0.40	0.52
GPT-4	simple	✓	8.9	29.3	36.0	0.47	0.56
StarChat	base	✗	0.8	7.3	8.5	0.13	0.24
StarChat	base	✓	1.4	9.1	11.0	**0.23**	0.30
StarChat	simple	✗	1.5	6.7	7.3	0.16	0.24
StarChat	simple	✓	**3.0**	**17.1**	**17.7**	**0.23**	**0.36**

for all test-set correct postconditions (see Section 2.1). Table 2 contains our results. We report both the percentage of postconditions that are *bug-complete* (kill all distinct code mutants) and the average *bug-completeness score* (fraction of code mutants killed). The results indicate that `GPT-4` postconditions can kill all the code mutants for up to 62.2% of examples in *EvalPlus*. Overall, both `GPT-3.5` and `GPT-4` generate relatively bug-complete postconditions, with average bug-completeness scores of up to 0.76 and 0.85 respectively. That is, the average correct postcondition generated by these models discriminates over three-quarters of distinct buggy codes. The bug-completeness scores for `StarChat` were lower but still substantial, catching up to one-third of mutants. Our bug-completeness results suggest that LLMs, especially larger models like `GPT-3.5` and `GPT-4`, can use natural language to produce postconditions that meaningfully capture desired aspects of program behavior.

In contrast to the correctness results (Section 3.2.1), `base` postconditions generally have higher bug-completeness scores than `simple` postconditions (up to a 30% difference). Thus, the `simple` prompt may generate more correct postconditions at the expense of bug-catching power. Even so, as shown in Fig. 4, `simple` postconditions still meaningfully capture aspects of program behavior: correct `simple` `GPT-4` and `GPT-3.5` postconditions discriminate over half of unique buggy mutants.

We also compare the bug-completeness of postconditions generated from natural language intent alone to those generated with a reference solution in the prompt. While the difference was not quite significant, the average bug-completeness score was 5% higher for the case with the reference code included ($p = 0.06$). From our qualitative investigation, this seems to be caused by an increase in the number of postconditions that are functional re-implementations of the reference solution.

Natural vs. Artificial bugs. To help validate our proposed bug-completeness metric (see Section 2.1), we examine the impact of using *natural* or *artificial* LLM code generation bugs. As shown in Table 2, our completeness metric was consistently (though not always substantially) lower when only considering natural bugs; naturally occurring LLM code generation bugs are harder to kill via *nl2postcond* than artificially seeded bugs. This finding highlights a potential limitation in using artificially seeded faults to assess postcondition correctness as it may artificially inflate the metric. However, generating unique natural bugs is more expensive than using artificial bugs. To ensure metric robustness, augmenting the evaluation metric with artificial bugs may still be useful.

Table 3. Atomic categories of *nl2postcond* postconditions that are often combined by LLMs via && (logical and). `return_val` refers to the function's return value. *% test-set correct* and *bug-completeness* columns are defined in Section 2.1. Example postconditions are adapted from our *EvalPlus* results, only modified for space.

Category	Example Postconditon	% Prevalent	Avg. Bug-complete-score (*Natural*/*All*)
Type Check	`isinstance(return_val, int)`	**47.4**	*0.14* / *0.27*
Format Check	`return_val.startswith("ab")`	11.2	0.43 / 0.57
Arithmetic Bounds	`return_val >= 0`	30.8	0.23 / 0.34
Arithmetic Equality	`return_val[0] == 2 * input_val`	17.5	**0.82** / **0.89**
Container Property	`len(return_val) > len(input_val)`	27.0	0.45 / 0.57
Element Property	`return_val[0] % 2 == 0`	12.6	0.39 / 0.53
Forall-Element Property	`all(ch.isalpha() for ch in return_val)`	8.3	0.23 / 0.44
Implication	`(return_val==False) if 'A'not in string`	12.7	0.58 / 0.64
Null Check	`return_val is not None`	4.4	0.40 / 0.50
Average			0.32 / 0.46

RQ1 summary: Postcondition Completeness on *EvalPlus*

We find that for the benchmark *EvalPlus*, *nl2postcond* postconditions generated by `GPT-3.5` and `GPT-4` can meaningfully capture program intent especially when using our `base` prompt: the average correct postcondition generated by these models can discriminate three-quarters of unique buggy code mutants depending on the prompt variation.

3.2.3 Qualitative Analysis of Generated Postconditions. Evaluating postcondition correctness and completeness tells us how well LLMs can generate specifications that capture program intent, however it does not give us insight into the kinds of generated specifications, and how they differ in terms of performance. We ask two questions: 1) **Are there patterns within LLM generated postconditions** and 2) **How do these categories differ in terms of correctness and completeness?** These insights can help to inform future improvements around LLM generated specifications, and may guide ranking or selection strategies when using generated postconditions in practice.

To determine what program aspects *nl2postcond* postconditions verify, we conduct a manual qualitative analysis. We first select 230 postconditions generated for 23 *EvalPlus* problems. We use the best-performing prompt version for correctness: `GPT-4` with the `simple` prompt and no reference solution. The first two authors developed a set of qualitative coding categories for postcondition structure and jointly labeled all 230 postconditions. The first author then used this set of categories to label an additional 670 postconditions for a total of 900 labeled postconditions from 139 *EvalPlus* problems We present these categories in Table 3 and report prevalence and completeness.

We observe that postconditions can take the form of either atomic or conjoined statements. For example, an LLM may generate a single postcondition that checks several distinct properties about a program, conjoined with logical && operators. Results of the classification process show that 33% of LLM-generated postconditions consist of multiple *atomic postconditions*, conjoined using &&.

We categorize nine basic types of atomic properties. Table 3 contains an example of each, along with its dataset prevalence and completeness measures. Prevalence is counted across both atomic and conjoined statements, e.g. if an assertion conjoins specifications across two categories, both are counted. As a result, prevalence adds to over 100%. `Type Checks` enforce a constraint on the type of a return value using `isinstanace`. `Format Checks` ensure that the return value follows a certain string format constraint. `Arithmetic Bounds` and `Arithmetic Equality` enforce a numeric bounding or equality constraint against another expression. `Container Property` checks an aspect

of a complex type or object (e.g., the length of an array). `Element Property` and `Forall-Element Property` enforce some constraint on one or all elements of a collection. `Implications` include conditional logic, and `Null Check` ensures that the return value is not `None`.

We did not observe a significant relation between postcondition type and correctness. However, we do observe significant differences in bug-completeness across categories. For example, postconditions labeled as `Type Checks`, i.e. specifications enforcing the type of the return value, were the weakest, only killing 27% of bugs on average. This difference was particularly pronounced for *natural bugs* (see Sections 2.1 and 3.1.4), where `Type Checkers` only killed 14% of bugs on average. Interestingly, `Type Checks` are also the most prevalent category, indicating LLM preference towards generating such constraints. Low completeness scores indicate that, for the studied dataset, type-mismatch errors is not a common bug source. This may be explained by the inclusion of type hints in the *EvalPlus* dataset, which appear in function stubs provided to the LLM.

On the other hand, `Arithmetic Equality` checks, i.e. specifications that assert that parts of the return value must be equivalent to another expression, provide a strong postcondition. On average, this category of postcondition kills 89% of all bugs and appears in 17.5% of labeled postconditions.

Using our categorization, we can partially explain the lower completeness scores of `StarChat` postconditions in section 3.2.2. While we do not perform a systematic qualitative analysis, we observe that the majority of correct `StarChat` postconditions are atomic `Type Checks`, which is the weakest postcondition type (see Table 3). This hypothesis is also validated by results of `GPT-4`, where in contrast, only 16% of generated postconditions are atomic `Type Checks` alone. Instead, the majority of `Type Checks` in `GPT-4` are in conjoined statements with other atomic checks, which may explain the relatively higher average completeness scores between the two models.

RQ1 summary: Qualitative Analysis of Postconditions for *EvalPlus*

We qualitatively identify nine atomic component categories of LLM-generated postconditions. While we observe minimal correctness differences, bug completeness varied significantly; the weakest postcondition type, `Type Checks`, killed only 14% of natural bugs on average while the strongest, `Arithmetic Equality` check, killed 82%, a 6x difference.

4 RQ2: CAN *NL2POSTCOND* HELP CATCH REAL WORLD BUGS?

Beyond understanding whether LLMs can capture natural language intent via executable postconditions, we also want to understand *nl2postcond*'s real-world potential. To do so, we investigate the second motivating use case in Section 1.1: finding bugs in an existing code base. We evaluate *nl2postcond*'s bug-catching potential using *Defects4J* [21], a benchmark of historical Java bugs.

4.1 RQ2–Research Methodology and Experimental Setup

We outline our methodology for evaluating the capabilities of postconditions to catch real-world bugs: we describe the target benchmark *Defects4J*, discuss prompt variations for Java, and provide our criteria for bug-discriminating postconditions. We model our approach after TOGA's approach [9], where the goal is to find specifications/tests that a user could have used to catch a bug as they fail on the buggy version, and succeed on the fixed version.

4.1.1 Benchmark: Defects4J. We use *Defects4J* 2.0 [21], a well-known benchmark of 835 manually curated real-world bugs gathered from 17 Java projects. For each bug, the dataset has a set of bug-reproducing test cases (trigger tests), and regression test cases which load the class in which the method under test is contained. Each bug in *Defects4J* contains buggy and fixed versions of the code. We consider a postcondition to be test-set-correct if it passes all trigger and regression

tests on the fixed version. As our prompt leverages functional syntax introduced in Java 8 (see the postcondition in fig. 5c as an example), we only consider the subset pf 525 bugs that are reproducible when allowing this syntax. Each bug may involve changes to multiple functions, for which we each generate postconditions. In total 840 functions are modified across the 525 bugs.

4.1.2 Bug Discriminating Postconditions. To evaluate whether LLM-generated postconditions are capable of catching real-world bugs, we instrument the buggy and fixed function versions with each associated postcondition. We consider a generated postcondition to be *bug-discriminating* if it satisfies the following criteria:

(1) The postcondition **passes** all the trigger and regression tests, on the fixed version of a function.
(2) The postcondition **fails** a a trigger test or regression test on the buggy version of a function.

The *Defects4J* benchmark ensures that the difference between the buggy and fixed versions is minimized to only changes related to the bug-fix. Therefore, assuming a comprehensive test suite, any discriminating postcondition satisfying the above criteria is related to the change for the example (bug-related). Finally, similar to our qualitative evaluation for RQ1 (see Section 3.2.1) we qualitatively analyze bug-discriminating postconditions to gain greater insight.

4.1.3 Prompt Design and Ablations. To generate postcondtions for buggy functions in the dataset, we use the same prompt as in RQ1 (see fig. 3). Designed as language agnostic, the only change needed to adapt the prompt for *Defects4J* is including additional code context. Given that *Defects4J* problems are extracted from real-world projects, functions are comparatively more complex than those in *EvalPlus* and are often tightly coupled with other project functions. Our initial investigations found that without some file-level context, LLMs rarely generate meaningful postconditions that also compile. Therefore, we include additional class and type-related context in the prompt. Given the limited context window of the LLMs used, we greedily include methods in the call graph for the buggy function (ordered by in-file placement) until the prompt tokens are exhausted. The call graph and in-file dependencies are determined using the Java language binding for `Tree-sitter`[4].

For each buggy function, we combine of function and class-level in-file comments to formulate a natural language specification. In practice, this is primarily the buggy function's JavaDoc. We do not generate additional natural language (i.e., through code summarization) nor do we use external documentation: all natural language is pulled directly from the buggy function's source code file.

We choose to use only the `simple` prompt from RQ1, as it had more correct postconditions than did the `base` version. Following the approach in RQ1, we report two variants of the prompt: 1) that only includes the natural language of the function and 2) that includes both the natural language and the code of the buggy function body. For each variant, we generate 10 postconditions for every function modified between the buggy and fixed projects (840 functions across 525 unique bugs).

We choose to generate postconditions using two of the three models used in RQ1: `GPT-4` and `StarChat`. Given that `GPT-4` and `GPT-3.5` are comparable, closed-access chat models from OpenAI, we choose to focus on `GPT-4` as it shows superior performance in RQ1. We choose to use `StarChat` as it is one of the few open-source chat-based models available. In total, we evaluate 33,600 postconditions (2 models $*$ 2 ablations $*$ 10 postconditions $*$ `840` functions).

4.2 *RQ2*–Results: Can LLM-generated Postconditions Catch Real-World Bugs?

We detail our findings on if *nl2postcond* postconditions are test-set correct, and if they can catch bugs in real-world industrial-scale projects. We find that even with the increased complexity over *EvalPlus*, `GPT-4` is still able to produce correct postconditions for *Defects4J* at a high rate. In addition, both `GPT-4` and `StarChat` are able to generate bug-discriminating postconditions for a subset of

[4]https://tree-sitter.github.io/tree-sitter/

Table 4. Table containing our *Defects4J* results for postconditions generated for 840 methods across 525 historical bugs. We report the likelihood of generated postconditions to compile, and the `accept@k` likelihood that they pass all tests when instrumenting the fixed function (*test-set correct* columns). *# distinguishable bugs* is the number of bugs for which at least one generated postcondition was discriminating (see Section 4.1.2).

Model	Prompt has: NL Only = ✗ buggy code = ✓	Compiles			Test-set correct			Number distinguishable bugs
		@1	@5	@10	@1	@5	@10	
`GPT-4`	✗	0.65	0.86	0.89	0.32	0.57	0.66	35
`GPT-4`	✓	**0.73**	**0.90**	**0.93**	**0.39**	**0.66**	**0.75**	**47**
`StarChat`	✗	0.25	0.68	0.83	0.11	0.38	0.55	19
`StarChat`	✓	**0.29**	**0.72**	**0.84**	**0.12**	**0.39**	**0.56**	**24**

Defects4J bugs. Bug-discriminating postconditions were further analyzed via a qualitative analysis to gain insight into the ability of LLMs to catch bugs via *nl2postcond*.

4.2.1 Test-set Correctness. Our full test-set correctness results for *Defects4J* are in table 4. We find that while lower than the results from *EvalPlus*, `GPT-4` still generate a significant number of test-set correct postconditions with respect to the fixed version of a function (e.g., correct with respect to programmer intent), achieving `accept@1` of up to 0.39 and `accept@10` of up to 0.75. `StarChat` performs worse, with `accept@1` and `accept@10` of 0.12 and 0.56 respectively. We note that these numbers may be higher in practice if postconditions are filtered by those that compile (see table 4, `Compiles` column). In general, including the buggy code in the prompt leads to more test-set correct postconditions. This contrasts with the results from *EvalPlus*, where we did not observe a difference. We hypothesize that this is the case because of (a) the comments not being completely descriptive, and (b) the increased program and object complexity in *Defects4J*, as supported by the fact that postconditions are also less likely to compile when the buggy code is omitted from the prompt.

4.2.2 Bug-discriminating Postconditions. We find that LLMs can generate postconditions that distinguish between buggy and fixed code in real-world projects with respect to regression and trigger tests. As seen in Table 4, `GPT-4` was able to generate discriminating postconditions for up to 47/525 (9%) bugs. `StarChat` caught fewer, but still generated postconditions that distinguished up to 25 bugs. Across all prompt variants and models, we were able to generate a bug-discriminating postcondition for 70 buggy methods from 64 unique bugs in *Defects4J*, 12.2% of all bugs considered.

RQ2 summary: Correctness and Bug Catching Power on *Defects4J*

We find that *nl2postcond* postconditions are often test-set correct for real-world functions (`accept@10` up to 0.75) and can be powerful enough to catch real-world bugs (*nl2postcond* discriminates 70 buggy methods from 64 bugs in *Defects4J*).

4.2.3 Qualitative Analysis of Bug-discriminating Postconditions. We conduct a qualitative evaluation of the bug-discriminating postconditions to gain insight into how *nl2postcond* postconditions discriminate real-world bugs. We observed additional evidence both motivating the potential usefulness of *nl2postcond* and examples of why LLMs may be a good tool to solve this problem. To communicate these findings, we detail two cases.

The first case is a historical bug from the Apache Commons CLI project.[5] As shown in fig. 5, the program should render multi-line text such that 1) white space padding is added at the beginning of every line after the first one and 2) that no line length exceeds a specified `width`. The requirement

[5]Project page: https://commons.apache.org/proper/commons-cli/, Bug: https://issues.apache.org/jira/browse/CLI-151

```
/** Render the text and return the rendered Options in a StringBuffer.
 * @param width The number of characters to display per line
 * @param nextTab The position on the next line for the first tab.
 * @param text The text to be rendered.*/
StringBuffer renderWrappedText(StringBuffer sb, int width, int nextTab, String text);
```

(a) Buggy function stub and javadoc.

> The method... has couple of bugs in the way that it deals with the [nextTab] variable. This causes it to format every line beyond the first line by [nextTab] too many characters **beyond the specified width**.

(b) Bug report indicating that the function sometimes erroneously renders text with more than `width` characters per line, behavior that directly conflicts with the javadoc.

```
// Checks if the rendered text does not exceed the specified width per line
assert rVal.toString().lines().allMatch(line -> line.length() <= width);
```

(c) Bug catching *nl2postcond* postcondition generated by `GPT-4`. `rVal` is the function return value. This postcondition was generated without the buggy function code in the prompt.

```
public void test27() throws Throwable {
  HelpFormatter helpFormatter0 = new HelpFormatter();
  MockPrintWriter mockPrintWriter0 = new MockPrintWriter("—");
  helpFormatter0.printUsage((PrintWriter) mockPrintWriter0, 0, "[ Options: [ sh6ort "); }
```

(d) Bug-catching test prefix from TOGA where TOGA expects this test prefix to throw a `RuntimeException`. While this catches the bug, it is semantically removed from the bug's root cause.

Fig. 5. Example from *Defects4J* (Cli project, bug 8) where the bug can be caught via *nl2postcond*. Cli 8 is a bug in the implementation for calculating the width of lines when wrapping output text. The natural language function description specifically says that each line must be at most `width` characters long. `GPT-4` translates this intent into the provided postcondition, which correctly catches the bug.

that each line should be `width` characters long is clearly specified in the Javadoc. However, the program sometimes incorrectly rendered lines longer than `width` due to a bug in the white space padding implementation. In our evaluation, `GPT-4` generated multiple postconditions that catch this bug, including the example in 5c. These bug-catching postconditions were generated by both prompt variations. This example evidences both that 1) informal natural language can meaningfully telegraph code bugs and 2) modern LLMs, such as `GPT-4`, can sufficiently formalize natural language intent to capture the disagreement. Overall, this example shows the potential of *nl2postcond* to unearth coding inconsistencies solely from informal natural language documentation.

For our second example, we refer to one of our initial motivating examples in Section 1.1, fig. 2. This example was adapted from *Defects4J*, and consists of a historical bug in another popular Apache library project, Commons Math.[6] In this bug, a method returning a reversed instance of a mathematical `Line` object does not retain sufficient precision in its internal state. `GPT-4` is able to generate multiple postconditions that correctly detect this bug: both examples in fig. 2 are actual postconditions from our evaluation generated using the prompt with the buggy code included. This example also demonstrates the potential of LLMs to generate postconditions powerful enough to capture real-world bugs. In addition, it provides evidence that LLMs in particular are helpful for realizing *nl2postcond*. Both postconditions detect the bug by leveraging general mathematical knowledge about the properties of a reversed line. The second postcondition in particular exemplifies the ability of LLMs to dynamically combine methods such as `dotProduct` from the project file's context with algebraic world knowledge that is external to the project's code.

[6]Project page: https://commons.apache.org/proper/commons-math/, Bug: https://issues.apache.org/jira/browse/MATH-938

4.3 Baseline Comparison: TOGA, Daikon

We provide an empirical and qualitative comparison of the effectiveness of *nl2postcond* with respect to two other popular methods of inferring test oracles and invariant specifications. We choose a state-of-the-art technique for each: (a) TOGA [9], a neural approach to generating test oracles, and (b) Daikon [10], a popular technique to infer program invariants (including method postconditions) from multiple dynamic executions. There exists related efforts on generating unit tests neurally such as AthenaTest [51], but no public release exists for evaluating it for our setup.[7] We focus our comparison on *Defects4J*: to the best of our knowledge, neither TOGA nor Daikon support Python (and thus are not compatible with *EvalPlus*).

4.3.1 TOGA. TOGA is a neural approach to generating test oracles for a method. Given a test prefix, TOGA generates an assertion or expected exception that the test prefix is expected to satisfy. Although both can generate assertions that may not agree with the implementation of a method, there are fundamental differences between test oracle generation (as in TOGA) and specification generation (as in *nl2postcond*). *nl2postcond* infers method postconditions that are expected to hold for all inputs. These can not only be checked during testing, but also at runtime on unseen inputs and trigger assertion failures instead of producing corrupted values; TOGA assertions can only be applied at testing time, since the assertions apply to the specific test prefix that reaches the buggy location. However, there are (algebraic) specifications that are best expressed over multiple method calls (e.g., `s.pop(s.push(5)) == 5` for a stack `s`); expressing such a specification as a method postcondition (for either `s.pop` or `s.push` for even a single value 5) will require adding auxiliary ghost variables. Finally, test oracle assertions are often equalities (to match the expected output value on the specific input), whereas method postcondition assertions can be arbitrary Boolean expressions to capture all possible output values (see Table 3 for examples).

Setup. We compare *nl2postcond*'s results on *Defects4J* with the results reported on TOGA [9]. To enable TOGA to catch bugs without access to the failing trigger test, Dinella *et al.* integrated TOGA with EvoSuite [13], a popular automated testing tool. We used the set of 57 bugs found by TOGA (by reproducing their experimental setup released as a docker), each accompanied by a EvoSuite-generated test prefix and the corresponding test oracle. Of the 57 bugs reported by TOGA, 15 bugs were excluded from our *nl2postcond* evaluation due to Java limitations (Section 4.1.1).

Evaluation Results. Overall, we find that the bug finding capabilities of TOGA and *nl2postcond* are complementary. Of the 101 distinct bugs caught by at least one approach, only 5 are caught by both *nl2postcond* and TOGA.[8] 37 are only caught by TOGA, while 59 are only caught by *nl2postcond*. To better understand the differences between the two techniques, we conduct a qualitative evaluation of bugs caught by at least one approach. We note the following observations:

- A majority of *nl2postcond*-caught bugs (52/64) could not be found by TOGA, due to the lack of any EvoSuite generated test prefix that reaches the bug location. This includes *Math 9*, one of the two *Defects4J* bugs we use to motivate *nl2postcond* (see fig. 2). This demonstrates the usefulness of *nl2postcond*'s ability to be checked at runtime on unseen inputs.
- Most TOGA-caught bugs are "exceptional" bugs where the code either throws an *unexpected exception* or fails to throw an *expected exception*. Since we do not model exceptional postconditions in *nl2postcond*, we fail to find most such bugs. Fig. 6 shows how leveraging a model predicting exceptional postconditions helps TOGA catch bugs that *nl2postcond* does not. Incorporating exceptional postconditions into *nl2postcond* is an intriguing direction for future work. Beyond

[7] Personal communication with the author of AthenaTest.
[8] The 5 in common are Cli 8, Cli 32, JacksonCore 8, Jsoup 88, and Math 99.

```
public void test16() throws Throwable {
  byte[] byteArray0 = new byte[179];
  ByteArrayInputStream inputStream0 = new ByteArrayInputStream(byteArray0);
  ArchiveStreamFactory archiveStreamFactory0 = new ArchiveStreamFactory();
  try {
    archiveStreamFactory0.createArchiveInputStream((InputStream) inputStream0);
    fail("Expecting exception: Exception");
  } catch(Exception e) {
    verifyException("org.apache.commons.compress.archivers.ArchiveStreamFactory", e);
  }}
```

(a) TOGA test oracle that catches Compress 11. TOGA finds the bug by simulating a small file and then explicitly catching the resulting exception.

```
\old(in.getClass().getName()) == java.io.BufferedInputStream.class.getName()
```

(b) Daikon postcondition that catches Compress 11. It does so as the buggy function involves a using a `ArchiveStream` factory function that can change the class name of the Input Stream class.

Fig. 6. TOGA test oracle and Daikon postcondition for a historical bug caught by both TOGA and Daikon, but not by *nl2postcond* (Compress 11). This bug involves incorrectly processing files less than 512 bytes as tar archives, and it was fixed by throwing an exception.

exceptional postconditions, we also find that TOGA can model test prefixes that involve objects from different classes and methods (similar to the stack example with `push` and `pop`).

- For the 5 common bugs, we observe that TOGA and *nl2postcond* find the same underlying bug with different means. For example, one of our motivating examples for *nl2postcond*, *Cli 8*, is also caught by a TOGA test oracle. While both are helpful, *nl2postcond*'s assertion directly captures the semantics of the root cause of the bug (useful for both fault localization and patch construction). TOGA, however, provides a higher-level end-to-end test that is more removed from the buggy method, necessitating the developer spend additional time for root cause analysis. We present both bug catches for *Cli 8* in fig. 5d.

4.3.2 Daikon. Daikon [10] uses multiple program runs to dynamically infer program invariants, including postconditions. Unlike *nl2postcond*, Daikon invariants are always implementation-consistent (it only retains expressions that are true across tested executions) and can only be generated from testable code (e.g., can not be generated from natural language alone).

Setup. We used Daikon to generate likely postconditions from running the set of regression tests (without any failing trigger tests) on the buggy version. We then check if these specifications are bug-discriminating. We run Daikon using standard parameters for each buggy method to generate a set of likely postconditions. Due to challenges integrating Daikon with several of the projects in *Defects4J*, we scope our evaluation to the 101 bugs found by either *nl2postcond* or TOGA.

Results. Overall, we find that while Daikon generates many postconditions that are consistent with all tests, bug-discriminating postconditions are rare. Daikon generated postconditions for an associated buggy method for the majority of tested bugs (78/101). For the rest, Daikon either failed to generate any method postconditions on the buggy version using just the regression tests (17/101), or timed out after 10 minutes (6/101). The number of postconditions generated for any given method varied widely. However, we observed only three instances of a Daikon-generated postcondition that is bug-discriminating. Daikon finds one bug that is not found by *nl2postcond*, but the other two specifications are incorrect. Fig. 6b shows the case where Daikon is able to catch a bug that *nl2postcond* does not, by detecting a class name change instigated through a factory function. For the remaining two cases, the bug-discriminating postcondtions overfit the regression

```
// Checks if the returnValue is greater than or equal to zero
assert returnValue >= 0;
```

(a) Example bug-catching post conditions generated by *nl2postcond* which correctly asserts that the domain of a continuous distribution function should be greater than or equal zero. This postcondition catches a large number of bug-triggering inputs for this method.

```
daikon.Quant.fuzzy.eq(\result, 1.000020000400008) || daikon.Quant.fuzzy.eq(\result, 1.5)
```

(b) Daikon postcondition that distinguishes Math 9, but overfitts to the regression tests.

```
  /** Access the initial domain value, based on <code>p</code>, used to
   * bracket a CDF root. This method is used by
   * {@link #inverseCumulativeProbability(double)} to find critical values.
   * @param p the desired probability for the critical value
   * @return initial domain value */
  protected double getInitialDomain (double p) {
      double ret = 1.0;
      double d = getDenominatorDegreesOfFreedom();
      if (d > 2.0) {
          ret = d / (d — 2.0);
      }
      return ret; }
```

(c) Math 95 from *Defects4J*: This function returns a domain for use by an Inverse Cumulative Probability function. The buggy version did not have sufficient bounds on `getDenominatorDegreesOfFreedom`, leading to a potential negative domain (impossible for a cumulative Probability function) or a division by zero error. Highlighted tokens are those that were added for the fixed version.

Fig. 7. Comparison of *nl2postcond*, TOGA, and Daikon on *Math 95*.

tests and do not hold for all inputs. For example, the specification for Math 95 in Fig. 7b) states that a return value should be close to one of two values $\{1.0, 1.5\}$. However, the fixed program admits many more positive return values; this is correctly reflected by the *nl2postcond* postcondition in Fig. 7a. In general, we observe that Daikon generated invariants are either very weak (e.g., a field is not modified), or are incorrect (do not generalize to all inputs). To the best of our understanding of *Defects4J*, this is in contrast to the majority of bug-discriminating *nl2postcond* postconditions.

Baseline Comparison with TOGA and Daikon

Compared to two other approaches, we find that *nl2postcond* postconditions are either more widely applicable or find more bugs. We also note that the bugs found via *nl2postcond* are largely non-overlapping with those found by TOGA, indicating that the two approaches may be complementary. *nl2postcond* finds many more bugs compared to Daikon, which often generates invariants that are not bug-discriminating or overfit the observed executions.

5 RELATED WORK

Specification Generation. A specification provides a comprehensive description of a program's intended behavior, encompassing the functional relationships between inputs and outputs, as well as the internal state dynamics. Specifications may vary in formality, ranging from informal descriptions such as API documentation to formal representations like test cases or runtime assertions. The applications of program specifications are extensive, and include bug identification [1, 19], verification [5, 32], specification-driven development [27, 36, 41], and code comprehension [4]. Our

goal is to generate formal and functional specifications in the form of postconditions, articulating the desired input-output relationship of a code given the informal natural language description. There has been a long line of work for automatically inferring specifications using static analysis [44], abstract interpretation [7], dynamic analysis [10], and so on. While most of these existing works rely on a code implementation inferring the specification of existing code, our approach is to infer the desired behavior of the code from natural language. Similar to us, several approaches attempted to generate specification by analyzing API documentation or code comments using different natural language processing techniques such as named pattern matching [38, 47–49], text normalization [3], entity recognition [57], natural language parsing [59], etc. Being dependent on mostly hand-crafted rules and heuristics, most of these techniques only work on the semi-structured natural language format of the input and are not easily extensible across different programming languages and domains. In contrast, our technique relies on LLMs for world knowledge and our experiment shows the extensibility of our technique in two different languages – Python and Java.

Machine Learning for Specifications. Machine Learning approaches for specification generation have shown promise in several directions, including synthesizing test oracles [9, 31, 52], improving test coverage [28], generating unit tests [24, 50]. Depending on the scenario, the specifications generated by these approaches are dependent on different inputs. AthenaTest [50] generates both the input and the oracle of a unit test from the implementation of the focal method (recall TOGA only generates test oracle). Closer to our work, TiCoder [24] leverages LLM to generate test input and output to formalize the user intent. While these approaches focus on generating concrete test cases (and potentially oracles), our approach is geared toward generating abstract functional relationships between the input and output of a procedure, which allows us to reason about a range of inputs. Similar to our work, EvoSpex [33] generates functional relationships of input-output with evolutionary learning. While their approach is aimed at summarizing existing program behavior (and therefore cannot be used to find bugs), our approach contributes towards generating formal specifications of desired input-output behavior. Recent work by Vikram et al. [53] proposes to leverage LLM for generating property-based tests (PBTs). Speculyzer [22] uses LLMs to enumerate likely properties and inputs similar to PBT, but use them as heuristics to improve code generation. Unlike our work, they do not seek to evaluate the correctness and completeness of these specifications. In addition to the input-output specification generation, machine learning has been applied to generate intermediate specifications of code such as invariants, using traditional machine learning [14, 43], deep learning [42, 56], and LLMs [39].

6 LIMITATIONS AND THREATS TO VALIDITY

LLM-Related Approach Limitations. We note that there are several inherent weaknesses of our approach relating to the use of LLMs. In particular, we note that as we are using popular LLMs as a black box, the underlying model is not well understood. This can lead to a lack of interpretability of the results, as well as raise questions regarding result generalizability. In addition, due to the quickly evolving AI landscape, the results may become obsolete quickly. We consider specific instances of these limitations.

Data Leakage. One potential concern to generalizability is the use of the benchmarks *EvalPlus* and *Defects4J* which are included in The Stack [23], the dataset used to train StarChat, and may have been included as part of training datasets for both `GPT-3.5` and `GPT-4`. The risk of data leakage could pose a threat to the internal validity of our study. Nevertheless, this concern is partially mitigated by the target task: the use of models to produce postconditions. To our knowledge, postconditions have not been previously generated as part of any public-facing dataset.

Stability of Models' Output. Two of the models used in the experiments are accessed using OpenAI web APIs. OpenAI models are not open-access and are often updated or deprecated. This poses a threat to the replicability of our study. To mitigate this threat, we make available all postconditions generated by the closed-access models. We also use the open-access `StarChat`, and share all generated artifacts. In addition, we report results using the widely adopted metric *accept@k*, which accounts for the stochasticity of model output.

Generalization of Findings. Given the relatively small number of bugs (525) considered in the *Defects4J* benchmark, our findings may not generalize to arbitrary bugs across different languages and repositories. We partially mitigate this threat by using real-world bugs from open-source projects and evaluating the capabilities of LLMs on both Python and Java benchmarks. In addition, the proposed taxonomy of postconditions (Section 3.2.3) is representative of only the programs in the *EvalPlus*benchmark and may not generalize across languages or program complexities.

Measure of Postcondition Completeness. Our metric for postcondition completeness relies on a set of generated code mutants. The code mutants are generated to cover the space of possible bugs in the target function, however, the set of code mutants generated per problem will never represent a comprehensive set of possible bugs. Therefore, our measure of completeness is dependent on the range and quality of bugs covered in the set of mutants. This poses a threat to the internal validity of our study. To mitigate this threat we maximize the diversity of bugs by retaining only distinct mutants and associated distinct tests, and generate up to 233 buggy codes per problem.

7 CONCLUSION

In this paper, we introduce and define *nl2postcond* as the problem of translating natural language comments into programmatically checkable postconditions via LLMs. Our work proposes and validates metrics for assessing the correctness and completeness of postconditions derived from natural language, offering an initial step in systematizing the *nl2postcond* problem. Through an empirical and qualitative evaluation on two benchmarks, we find that LLMs are adept at translating natural language descriptions to formulate non-trivial postconditions that accurately capture programming intent. Our study also finds that LLM-generated postconditions can exhibit high discriminative power: we generate postconditions via *nl2postcond* that are able to discriminate 64 real-world historical bugs from industrial-scale Java projects. These findings underscore the feasibility and promise of leveraging natural language documentation into executable specifications. Our research highlights the possibility of LLMs acting as a bridge between informal language descriptions and formal code specifications, such that natural language comments can be used effectively to improve software validation and bug detection.

8 DATA AVAILABILITY

Our code and artifacts will be publicly available at http://github.com/microsoft/nl-2-postcond. This package contains our postcondition generation scripts and prompts, *EvalPlus* postcondition evaluation harness, and our qualitative codebook. We also make all generated postconditions available. Finally, we include the unique natural and artificial LLM-generated mutants for *EvalPlus*.

REFERENCES

[1] Andrea Arcuri. 2008. On the automation of fixing software bugs. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*, Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn (Eds.). ACM, 1003–1006. https://doi.org/10.1145/1370175.1370223

[2] Amazon AWS. 2023. Amazon CodeWhisperer. Accessed September 27, 2023. https://aws.amazon.com/codewhisperer/.

[3] 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, Amsterdam, The Netherlands, July 16-21, 2018, Frank Tip and Eric Bodden (Eds.). ACM, 242–253. https://doi.org/10.1145/3213846.3213872

[4] Jonathan P Bowen, Peter T Breuer, and Kevin C Lano. 1993. Formal specifications in software maintenance: From code to Z++ and back again. *Information and Software Technology* 35, 11-12 (1993), 679–690. https://doi.org/10.1016/0950-5849(93)90083-F

[5] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. 2005. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures (Lecture Notes in Computer Science, Vol. 4111)*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever (Eds.). Springer, 342–363. https://doi.org/10.1007/11804192_16

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

[7] Patrick 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 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, Gary T. Leavens and Matthew B. Dwyer (Eds.). ACM, 213–232. https://doi.org/10.1145/2384616.2384633

[8] Edsger W. Dijkstra and Carel S. Scholten. 1990. Predicate Calculus and Program Semantics. (1990). https://doi.org/10.1007/978-1-4612-3228-5

[9] Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu K. Lahiri. 2022. TOGA: A Neural Method for Test Oracle Generation. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2130–2141. https://doi.org/10.1145/3510003.3510141

[10] 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 1999 International Conference on Software Engineering, ICSE' 99, Los Angeles, CA, USA, May 16-22, 1999*, Barry W. Boehm, David Garlan, and Jeff Kramer (Eds.). ACM, 213–224. https://doi.org/10.1145/302405.302467

[11] Sarah Fakhoury, Saikat Chakraborty, Madan Musuvathi, and Shuvendu K Lahiri. 2023. Towards Generating Functionally Correct Code Edits from Natural Language Issue Descriptions. *arXiv preprint arXiv:2304.03816* (2023).

[12] Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, and Shuvendu K. Lahiri. 2024. LLM-based Test-driven Interactive Code Generation: User Study and Empirical Evaluation. (2024). arXiv:2404.10100 [cs.SE]

[13] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, Tibor Gyimóthy and Andreas Zeller (Eds.). ACM, 416–419. https://doi.org/10.1145/2025113.2025179

[14] 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 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 499–512. https://doi.org/10.1145/2837614.2837664

[15] GitHub. 2023. GitHub Copilot. Accessed September 27, 2023. https://github.com/features/copilot/.

[16] Alberto Goffi, Alessandra Gorla, Michael D. Ernst, and Mauro Pezzè. 2016. Automatic generation of oracles for exceptional behaviors. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, Andreas Zeller and Abhik Roychoudhury (Eds.). ACM, 213–224. https://doi.org/10.1145/2931037.2931061

[17] Hao He. 2019. Understanding source code comments at large-scale. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 1217–1219. https://doi.org/10.1145/3338906.3342494

[18] Henry Hsu and Peter A Lachenbruch. 2014. Paired t test. *Wiley StatsRef: statistics reference online* (2014).

[19] Daniel Jackson. 1992. *Aspect, a formal specification language for detecting bugs*. Ph. D. Dissertation. Citeseer.

[20] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Trans. Software Eng.* 37, 5 (2011), 649–678. https://doi.org/10.1109/TSE.2010.62

[21] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, Corina S. Pasareanu and Darko Marinov (Eds.). ACM, 437–440. https://doi.org/10.1145/2610384.2628055

[22] Darren Key, Wen-Ding Li, and Kevin Ellis. 2022. I speak, you verify: Toward trustworthy neural program synthesis. *arXiv preprint arXiv:2210.00848* (2022).

[23] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, et al. 2022. The stack: 3 tb of permissively licensed source code. *arXiv preprint arXiv:2211.15533* (2022).

[24] Shuvendu K Lahiri, Aaditya Naik, Georgios Sakkas, Piali Choudhury, Curtis von Veh, Madanlal Musuvathi, Jeevana Priya Inala, Chenglong Wang, and Jianfeng Gao. 2022. Interactive code generation via test-driven user-intent formalization. *arXiv preprint arXiv:2208.05950* (2022).

[25] Andrew K. Lampinen, Ishita Dasgupta, Stephanie C. Y. Chan, Kory Matthewson, Michael Henry Tessler, Antonia Creswell, James L. McClelland, Jane X. Wang, and Felix Hill. 2022. Can language models learn from explanations in context? arXiv:2204.02329 [cs.CL]

[26] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 6355)*, Edmund M. Clarke and Andrei Voronkov (Eds.). Springer, 348–370. https://doi.org/10.1007/978-3-642-17511-4_20

[27] Andreas Leitner, Ilinca Ciupa, Manuel Oriol, Bertrand Meyer, and Arno Fiva. 2007. Contract driven development = test driven development - writing test cases. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*, Ivica Crnkovic and Antonia Bertolino (Eds.). ACM, 425–434. https://doi.org/10.1145/1287624.1287685

[28] 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 *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 919–931. https://doi.org/10.1109/ICSE48619.2023.00085

[29] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).

[30] 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. *37th conference on Neural Information processing Systems (NeurIPS), 2023* (2023). https://arxiv.org/abs/2305.01210

[31] 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 Trans. Software Eng.* 49, 4 (2023), 1580–1598. https://doi.org/10.1109/TSE.2022.3183297

[32] B Mike, K Rustan M Leino, and S Wolfram. 2004. The Spec# programming system: An overview. In Construction and Analysis of Safe, Secure, and Interoperable Smart devices (CASSIS)" volume 3362 of Lecture Notes in Computer Science.

[33] Facundo Molina, Pablo Ponzio, Nazareno Aguirre, and Marcelo F. Frias. 2021. EvoSpex: An Evolutionary Algorithm for Learning Postconditions. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 1223–1235. https://doi.org/10.1109/ICSE43902.2021.00112

[34] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).

[35] Theo X Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2023. Demystifying GPT Self-Repair for Code Generation. *arXiv preprint arXiv:2306.09896* (2023).

[36] Jonathan S. Ostroff, David Makalsky, and Richard F. Paige. 2004. Agile Specification-Driven Development. In *Extreme Programming and Agile Processes in Software Engineering, 5th International Conference, XP 2004, Garmisch-Partenkirchen, Germany, June 6-10, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 3092)*, Jutta Eckstein and Hubert Baumeister (Eds.). Springer, 104–112. https://doi.org/10.1007/978-3-540-24853-8_12

[37] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems* 35 (2022), 27730–27744.

[38] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit M. Paradkar. 2012. Inferring method specifications from natural language API descriptions. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, Martin Glinz, Gail C. Murphy, and Mauro Pezzè (Eds.). IEEE Computer Society, 815–825. https://doi.org/10.1109/ICSE.2012.6227137

[39] Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2023. Can Large Language Models Reason about Program Invariants? 202 (2023), 27496–27520. https://proceedings.mlr.press/v202/pei23a.html

[40] Rolf-Helge Pfeiffer. 2020. What constitutes Software?: An Empirical, Descriptive Study of Artifacts. In *MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*, Sunghun Kim, Georgios Gousios, Sarah Nadi, and Joseph Hejderup (Eds.). ACM, 481–491. https://doi.org/10.1145/3379597.3387442

[41] Richard Rutledge, Sheryl Duggins, Dan Lo, and Frank Tsui. 2014. Formal specification-driven development. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP)*. The Steering Committee of The World Congress in Computer Science, Computer . . . , 1.

[42] Gabriel Ryan, Justin Wong, Jianan Yao, Ronghui Gu, and Suman Jana. 2020. CLN2INV: Learning Loop Invariants with Continuous Logic Networks. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. https://openreview.net/forum?id=HJlfuTEtvB

[43] Rahul Sharma and Alex Aiken. 2016. From invariant checking to invariant inference using randomized search. *Formal Methods Syst. Des.* 48, 3 (2016), 235–256. https://doi.org/10.1007/S10703-016-0248-5

[44] Sharon Shoham, Eran Yahav, Stephen Fink, and Marco Pistoia. 2007. Static specification mining using automata-based abstractions. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2007, London, UK, July 9-12, 2007*, David S. Rosenblum and Sebastian G. Elbaum (Eds.). ACM, 174–184. https://doi.org/10.1145/1273463.1273487

[45] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. 2011. Secure Distributed Programming with Value-Dependent Types. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming* (Tokyo, Japan) *(ICFP '11)*. Association for Computing Machinery, New York, NY, USA, 266–278. https://doi.org/10.1145/2034773.2034811

[46] Tabnine. 2023. Tabnine Code Completion. Accessed September 27, 2023. https://www.tabnine.com/.

[47] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. 2007. /*icomment: bugs or bad comments?*/. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, Thomas C. Bressoud and M. Frans Kaashoek (Eds.). ACM, 145–158. https://doi.org/10.1145/1294261.1294276

[48] 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 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic (Eds.). ACM, 11–20. https://doi.org/10.1145/1985793.1985796

[49] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T. Leavens. 2012. @tComment: Testing Javadoc Comments to Detect Comment-Code Inconsistencies. In *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012*, Giuliano Antoniol, Antonia Bertolino, and Yvan Labiche (Eds.). IEEE Computer Society, 260–269. https://doi.org/10.1109/ICST.2012.106

[50] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2020. Unit Test Case Generation with Transformers and Focal Context. https://doi.org/10.48550/ARXIV.2009.05617

[51] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2021. Unit Test Case Generation with Transformers and Focal Context. arXiv:2009.05617 [cs.SE]

[52] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, and Neel Sundaresan. 2022. Generating Accurate Assert Statements for Unit Test Cases using Pretrained Transformers. In *IEEE/ACM International Conference on Automation of Software Test, AST@ICSE 2022, Pittsburgh, PA, USA, May 21-22, 2022*. ACM/IEEE, 54–64. https://doi.org/10.1145/3524481.3527220

[53] Vasudev Vikram, Caroline Lemieux, and Rohan Padhye. 2023. Can Large Language Models Write Good Property-Based Tests? *arXiv preprint arXiv:2307.04346* (2023).

[54] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. 2022. Emergent Abilities of Large Language Models. *Transactions on Machine Learning Research* (2022).

[55] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. arXiv:2201.11903 [cs.CL]

[56] 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 International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 106–120. https://doi.org/10.1145/3385412.3385986

[57] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. 2009. Inferring Resource Specifications from Natural Language API Documentation. In *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009*. IEEE Computer Society, 307–318. https://doi.org/10.1109/ASE.2009.94

[58] Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, and Ed Chi. 2023. Least-to-Most Prompting Enables Complex Reasoning in Large Language Models. arXiv:2205.10625 [cs.AI]

[59] Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald C. Gall. 2017. Analyzing APIs documentation and code to detect directive defects. In *Proceedings of the 39th International Conference on Software*

Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE / ACM, 27–37. https://doi.org/10.1109/ICSE.2017.11

Received 2023-09-28; accepted 2024-04-16