

Beyond Postconditions: Can Large Language Models infer Formal Contracts for Automatic Software Verification?

CEDRIC RICHTER and HEIKE WEHRHEIM, Carl von Ossietzky Universität Oldenburg, Germany

Automatic software verifiers have become increasingly effective at the task of checking software against (formal) specifications. Yet, their adoption in practice has been hampered by the lack of such specifications in real world code. Large Language Models (LLMs) have shown promise in inferring formal *postconditions* from natural language hints embedded in code such as function names, comments or documentation. Using the generated postconditions as specifications in a subsequent verification, however, often leads verifiers to suggest *invalid* inputs, hinting at potential issues that ultimately turn out to be *false alarms*.

To address this, we revisit the problem of specification inference from natural language in the context of automatic software verification. In the process, we introduce NL2CONTRACT, the task of employing LLMs to translate informal natural language into formal *functional contracts*, consisting of postconditions as well as *preconditions*. We introduce metrics to validate and compare different NL2CONTRACT approaches, using soundness, bug discriminative power of the generated contracts and their *usability* in the context of automatic software verification as key metrics. We evaluate NL2CONTRACT with different LLMs and compare it to the task of postcondition generation *nl2postcond*. Our evaluation shows that (1) LLMs are generally effective at generating functional contracts *sound* for all possible inputs, (2) the generated contracts are sufficiently expressive for discriminating buggy from correct behavior, and (3) verifiers supplied with LLM inferred functional contracts produce fewer false alarms than when provided with postconditions alone. Further investigations show that LLM inferred preconditions generally align well with developers intentions which allows us to use automatic software verifiers to catch real-world bugs.

CCS Concepts: • **General and reference** → *Metrics*; • **Software and its engineering** → *Formal software verification*; *Specification languages*; • **Computing methodologies** → *Artificial intelligence*.

Additional Key Words and Phrases: Contracts, Verification, LLMs, Specification Inference

ACM Reference Format:

Cedric Richter and Heike Wehrheim. 2018. Beyond Postconditions: Can Large Language Models infer Formal Contracts for Automatic Software Verification?. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 22 pages. <https://doi.org/XXXXXXX.XXXXXXX>

1 Introduction

Software verifiers [4–6, 19, 25, 36] have become increasingly effective at the task of showing that programs adhere to given specifications. With the right specification, they can rigorously guarantee the absence of software bugs, verify correctness properties, and significantly reduce the effort and cost associated with manual code inspection and debugging. Yet, the widespread adoption

Authors' Contact Information: Cedric Richter, cedric.richter@uol.de; Heike Wehrheim, heike.wehrheim@uol.de, Carl von Ossietzky Universität Oldenburg, Oldenburg, Germany.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Conference acronym 'XX, Woodstock, NY

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-XXXX-X/2018/06

<https://doi.org/XXXXXXX.XXXXXXX>

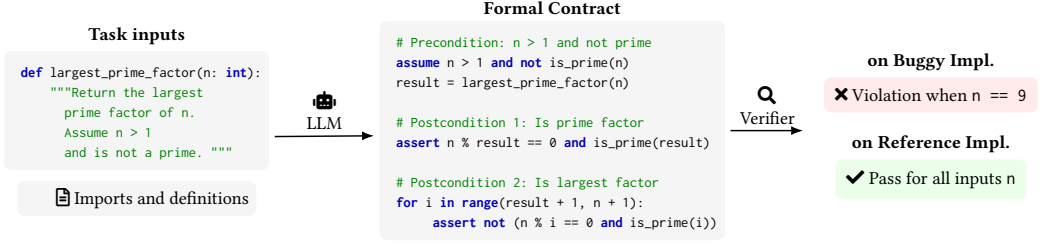


Fig. 1. Example task for NL2CONTRACT. The goal is to translate informal natural language descriptions into functional contracts which can be automatically checked by existing software verifiers.

of software verifiers in practice has been largely hindered by the fact that software developers must first of all provide some formal specification [44]. Manually writing these specifications is often highly nontrivial and hence most real world software lacks a *formal* specification of what it is intended to do.

In practice, *informal* natural language specifications describing the intended behavior of code, such as code comments or function documentation, are far more common [37]. Software developers use comments and documentation to communicate assumptions over inputs (preconditions) and over the expected behavior of their implementation (postconditions). At the same time, it is well known that software bugs often arise from *inconsistencies* between the documented intended behavior and the actual behavior of code [45–47]. These observations raise the question whether natural language descriptions, which are more common in practice, can be effectively translated into formal specifications, thereby enabling automatic bug detection. As Large Language Models (LLMs) have already proven to be able to synthesize code from natural language intent [9, 24], they are also natural candidates for the technological basis of such a translation.

Initial investigations [14, 59] have shown promise in the ability of LLMs to infer formal method postconditions from natural language descriptions of developer intent. Existing works such as *nl2postcond* [14] instruct a Large Language Model to write a “symbolic postcondition” for a function in the form of an `assert` statement. The generated postconditions can thus be programmatically checked allowing the automatic validation of their *soundness* and *completeness*. Still, in practice, we find that the generated postconditions that pass validation are *insufficient* when employed in the context of automatic software verification. More specifically, when tasked to generate postconditions, existing LLMs – as expected – typically ignore input assumptions leading to inferred postconditions which are only valid for a subset of inputs. A consecutively applied verifier then often produces counterexamples, i.e., inputs that violate the postcondition, which ultimately turn out to be *false alarms*.

To address this issue, we revisit in this paper the problem of specification inference from natural language descriptions in the context of automatic software verification. In this process, we propose the task NL2CONTRACT as an alternative to *nl2postcond*. To this end, we leverage LLMs for generating *functional contracts* in the form of pre- and postconditions from natural language information within code. Like for *nl2postcond*, we access NL2CONTRACT’s ability of generating meaningful contracts using soundness and completeness as our main quality indicators. In the process, we especially focus on NL2CONTRACT’s usage in the setting of software verification, checking for behavior not matching developer’s intents.

Example. In Figure 1, we provide an example of an NL2CONTRACT task adapted from HumanEval/59, a task from the Python code generation benchmark, HumanEval [9]. The programmer intends to implement a function (left) that computes the largest prime factor of n . In this, she assumes that n is an integer greater than 1 and not a prime number. Input validation is uncommon in scripting

languages such as Python¹ and hence the programmer expects that the user of the function ensures that the assumptions are fulfilled. The goal of NL2CONTRACT is now to translate the informal natural language specification into a formal contract that can be automatically checked by a verifier. Clearly, this contract should capture both input assumptions (as preconditions) and output constraints (as postconditions). In our example in Figure 1, we see that NL2CONTRACT has inferred preconditions (required clauses, given as *assumes*) and postconditions (ensures clauses, given as *asserts*) from the function’s signature, documentation and possible imports only. NL2CONTRACT evaluates the quality of such inferred contracts by supplying them to an automatic verifier which should then generate bug triggering inputs on buggy implementations. On correct reference implementations, the verifier should return “pass”. It in particular should avoid false alarms on correct code.

Limitations of Postcondition Inference. To further motivate NL2CONTRACT, we discuss the limitations of previous work that target postcondition inference. We use *nl2postcond* as an example. *nl2postcond* evaluates whether LLMs can generate sound postconditions that always hold after the execution of a function. Although postconditions are always dependent on the input and hence could encode precondition constraints, we find that current LLMs are *biased* towards simple (incomplete) postconditions that only encode output constraints. For our example in Figure 1 for HumanEval/59, *nl2postcond* generates the following postcondition (taken from *nl2postcond* evaluation):

```
assert n % return_value == 0 \
    and all(return_value % i != 0 for i in range(2, int(return_value**0.5) + 1))
```

The postcondition correctly encodes the expectation that the output of the `largest_prime_factor` should be a prime factor of n . However, it misses the precondition that $n > 1$ and not a prime number, and misses the postcondition that the result should be the largest prime factor. While the postcondition is test-set correct, i.e., all outputs of the function on test inputs of the evaluation set pass it, we find that software verifiers such as CrossHair [42] still report false alarms due to the missing input constraints. For example, CrossHair reports a bug for $n = 1$ when evaluated with the given postcondition on the non-buggy reference implementation. Our evaluation will show that this is a shortcoming of the task design of *nl2postcond* when employed within automatic verification, motivating the introduction of NL2CONTRACT.

Contributions. In this paper, we aim to systematically answer the following overarching research question:

Can LLMs translate informal constraints into function contracts useful for software verification?

Our interest in (and focus on within evaluation) is the inference of non-trivial and meaningful *preconditions*. Overall, we provide the following novel contributions:

- We propose the task of NL2CONTRACT for *inferring functional contracts* (consisting of pre- and postconditions) from developer intent via LLMs.
- We develop an *exception-based encoding* of functional contracts enabling the usage of automatic verifiers for checking.
- We provide sound formal definitions of *quality indicators* of functional contracts.
- We perform a thorough *experimental evaluation* of NL2CONTRACT itself, including
 - experiments with different LLMs (GPT-5, GPT-4o, CodeQwen 2.5) and
 - a comparison to *nl2postcond*, using both their and our own quality metrics.
- We empirically evaluate the *practicality* of generated contracts in the context of automatic software verification.
- We provide an evaluation of the usability of inferred specifications for finding *real bugs* in the Python-by-Contract benchmark set.

¹see PEP 20 - The Zen of Python

2 Background

Next, we introduce the background necessary for defining the NL2CONTRACT task.

Programs. Throughout this work, we consider simple programs containing functions and statements. We assume that the semantics of a program $P \in \mathbb{P}$ can be modeled as a function $f_P : \Sigma \rightarrow \Sigma$ that maps a program state $\sigma \in \Sigma$, e.g. variable assignments, input values etc., to another state $\sigma' \in \Sigma$. For each program P , we assume that there exists an *unknown* ground truth function $f_{gt} : \Sigma \rightarrow \Sigma$ and set of all *valid* input states $\Sigma_{gt} \subseteq \Sigma$. In our evaluation, we work with a benchmark set containing known (f_{gt}, Σ_{gt}) pairs. The ground truth function f_{gt} models the *correct* intended behavior of P on its domain Σ_{gt} . A program P is then only *correct* iff:

$$\forall \sigma \in \Sigma_{gt} : f_P(\sigma) = f_{gt}(\sigma)$$

The program P is *buggy* iff it deviates from the intended behavior for at least one valid input state $\sigma_{\mathbf{x}} \in \Sigma_{gt}$:

$$\exists \sigma_{\mathbf{x}} \in \Sigma_{gt} : f_P(\sigma_{\mathbf{x}}) \neq f_{gt}(\sigma_{\mathbf{x}}).$$

Specifications. Program specifications are a way to define the correct intended behavior of programs, either in its entirety or in parts. In this work, we focus on *program contract* based specifications [33, 38]. A program contract defines *pre-* and *postconditions* that the program has to adhere to. Formally, we define a program contract as a pair $(Pre, Post)$. A precondition $Pre : \Sigma \rightarrow \mathbb{B}$ is a predicate specifying for which input states the contract holds. A postcondition $Post : \Sigma \times \Sigma \rightarrow \mathbb{B}$ specifies the expected output behavior of the given program dependent on the given input. A program contract based specification $Spec$ consists of a set of program contracts, i.e. $Spec \subseteq PRE \times POST$ where PRE is the set of preconditions and $POST$ is the set of postconditions. We let $\mathbb{S}$ be the set of all specifications.

Soundness. We define a program contract $(Pre, Post)$ to be *sound* wrt. an (f_{gt}, Σ_{gt}) pair iff:

$$(S1) \quad \forall \sigma \in \Sigma : Pre(\sigma) \Rightarrow \sigma \in \Sigma_{gt} \quad (S2) \quad \forall \sigma \in \Sigma_{gt} : Pre(\sigma) \Rightarrow Post(\sigma, f_{gt}(\sigma))$$

In other words, a program contract is sound iff every input state σ satisfying the precondition is (1) a valid input state and (2) leads to an output state (when applying the ground truth function) jointly satisfying the postcondition. A specification $Spec$ is *sound* wrt. f_{gt} iff it consists of sound program contracts only.

A sound contract can be used to detect bugs in programs via checking for specification violations. A *specification violation* of a program P wrt. a specification $Spec$ is an input state $\sigma_{\mathbf{x}} \in \Sigma$ for which there exists a program contract $(Pre, Post) \in Spec$ such that:

$$Pre(\sigma_{\mathbf{x}}) \wedge \neg Post(\sigma_{\mathbf{x}}, f_P(\sigma_{\mathbf{x}}))$$

If there exists no such input state, the program P is said to be *correct* wrt. $Spec$.

A specification violation of a *sound* specification implies the existence of a bug in the program P . More specifically, soundness ensures that the following holds for all $(Pre, Post) \in Spec$:

$$\forall P \in \mathbb{P}, \sigma \in \Sigma : Pre(\sigma) \wedge \neg Post(\sigma, f_P(\sigma)) \Rightarrow \sigma \in \Sigma_{gt} \wedge f_P(\sigma) \neq f_{gt}(\sigma),$$

In other words, a specification violation $\sigma_{\mathbf{x}}$ always implies the existence of a bug in program P with $\sigma_{\mathbf{x}}$ as a *valid* bug-triggering input.

Completeness. Soundness of a contract thus ensures that specification violations actually hint to bugs. A different quality criterion for contracts is their completeness. Completeness aims at measuring the discriminative power of contracts, i.e., their ability of distinguishing buggy from non-buggy programs via specification violations. Soundness *does not* ensure that every bug leads to

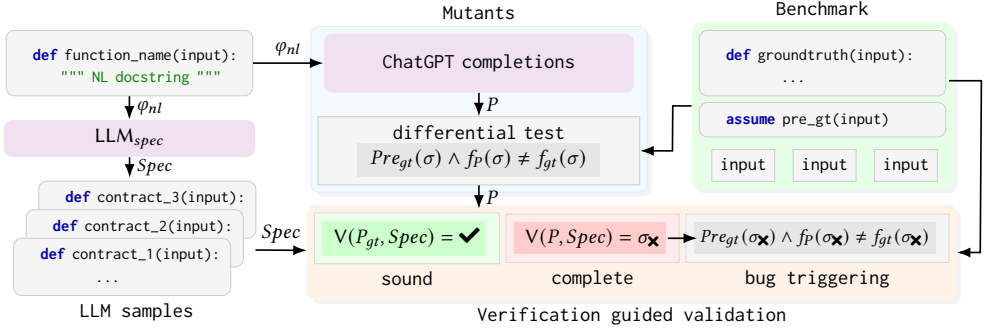


Fig. 2. Overview of NL2CONTRACT's specification validation process.

a specification violation. For this, we additionally need the specification to be *sufficiently complete*. Formally, a specification $Spec$ is sufficiently complete wrt. (f_{gt}, Σ_{gt}) iff:

$$(C) \quad \forall P \in \mathbb{P} : \exists \sigma \in \Sigma : \sigma \in \Sigma_{gt} \wedge f_P(\sigma) \neq f_{gt}(\sigma) \Rightarrow \exists \sigma_{\mathbf{x}} \in \Sigma : Pre(\sigma_{\mathbf{x}}) \wedge \neg Post(\sigma_{\mathbf{x}}, f_P(\sigma_{\mathbf{x}})),$$

for at least one $(Pre, Post) \in Spec$. Completeness thus ensures that whenever a program P is buggy there exists at least one specification violation. A *good* specification is both sound and sufficiently complete. In the sequel, we employ soundness and sufficient completeness (and variants thereof) as *quality indicators* of specifications.

Software Verifier. To find specification violations or prove their absence, we use software verifiers [6]. A software verifier checks the correctness of a program P wrt. a specification $Spec$. Formally, we view a software verifier as a function $V : \mathbb{P} \times \mathbb{S} \mapsto \{\checkmark\} \cup \Sigma$ which either returns $\checkmark$ (program correct) or an input state for which the program cannot establish the postcondition under the given precondition. Formally, a (sound) verifier should guarantee

$$\begin{aligned} V(P, Spec) = \checkmark & \Rightarrow P \text{ is correct wrt. } Spec \\ V(P, Spec) = \sigma & \Rightarrow Pre(\sigma) \wedge \neg Post(\sigma, f_P(\sigma)) \end{aligned}$$

We first of all employ verifiers for evaluating inferred specifications with respect to soundness and completeness, and ultimately for finding bugs in programs.

3 NL2CONTRACT

In this section, we introduce the NL2CONTRACT task and how we leverage it to evaluate the capabilities of LLMs to generate *sound* and *complete* specifications. Figure 2 provides an overview of our evaluation approach.

3.1 Task Formulation

The goal of NL2CONTRACT is to evaluate the ability of LLMs to generate specifications $Spec$ consistent with a natural language description φ_{nl} . Formally, we model the LLM as a function $LLM_{spec} : \Phi_{nl} \rightarrow \mathbb{S}$ which maps a natural language description $\varphi_{nl} \in \Phi_{nl}$ to a formal specifications $Spec \in \mathbb{S}$. The key question is how to measure that $Spec$ is consistent with φ_{nl} . For this, we propose the following task design.

Task Design. An NL2CONTRACT task is a triple $(\varphi_{nl}, P_{gt}, Pre_{gt})$ of natural language description φ_{nl} , reference implementation P_{gt} and reference precondition Pre_{gt} , the latter for instance given as assertions within programs. The program P_{gt} implements the ground truth function f_{gt} and the precondition Pre_{gt} describes its set of valid inputs Σ_{gt} .

The goal of LLM_{spec} when given an NL2CONTRACT task is to infer a specification Spec from the natural language description φ_{nl} . Naturally, the LLM neither has access to the reference implementation nor to the precondition. We use P_{gt} and Pre_{gt} to evaluate the quality of the inferred specification $\text{Spec} = \text{LLM}_{\text{spec}}(\varphi_{\text{nl}})$. As detailed in Section 2, a high quality contract has to be sound and sufficiently complete wrt. an $(f_{\text{gt}}, \Sigma_{\text{gt}})$ pair which our task contains in the form of $(P_{\text{gt}}, \text{Pre}_{\text{gt}})$.

Verification Guided Validation. Looking at the definitions of soundness and (sufficient) completeness, we see two challenges when actually evaluating contracts: Soundness needs to be checked on *all* states and completeness additionally investigates *all* programs. In particular the latter is infeasible in practice. Here, we address the first challenge by employing a formal verifier, ideally *proving* specification soundness, and a combination of verifier and mutations for the second challenge.

Soundness. For soundness we evaluate requirements (S1) and (S2) jointly. More specifically, let $(\text{Pre}, \text{Post}) \in \text{Spec}$ be one contract of a specification inferred by LLM_{spec} . Instead of separately checking (S1) and (S2), we derive a new specification $\text{Spec}_* = \{(\text{Pre}, \text{Post}_*) \mid (\text{Pre}, \text{Post}) \in \text{Spec}\}$ with $\text{Post}_*(\sigma, \sigma') = \text{Pre}_{\text{gt}}(\sigma) \wedge \text{Post}(\sigma, \sigma')$ for our evaluation. We then ask the verifier V to check whether P_{gt} satisfies Spec_* . If $V(P_{\text{gt}}, \text{Spec}_*) = \checkmark$, we have

$$\forall \sigma \in \Sigma : \text{Pre}(\sigma) \Rightarrow \text{Pre}_{\text{gt}}(\sigma) \wedge \text{Post}(\sigma, f_{\text{gt}}(\sigma))$$

for all $(\text{Pre}, \text{Post}) \in \text{Spec}$ and thus know soundness of Spec . By this construction, we can check soundness within a single verifier call.

Completeness. While the verifier will often be able to provide a proof of properties (S1) and (S2) for all states, it will – naturally – not reason about *all* possible programs. Instead of checking property (C) on all programs, we thus employ mutations, more precisely buggy variants of the reference implementation (alike Endres et al. [14]), to approximate completeness. Here, the key idea is to generate a large set of *buggy* code mutants CM that show different behavior than the reference implementation, i.e.:

$$\text{CM}(P_{\text{gt}}) \subseteq \{P \in \mathbb{P} \mid \exists \sigma \in \Sigma : \text{Pre}_{\text{gt}}(\sigma) \wedge f_P(\sigma) \neq f_{P_{\text{gt}}}(\sigma)\}$$

We then evaluate completeness on this set of code mutants CM , i.e., for every $P \in \text{CM}$, the verifier is tasked to find specification violations. If we get $V(P, \text{Spec}) = \sigma_{\mathbf{x}}$ (for some $\sigma_{\mathbf{x}} \in \Sigma$) for all $P \in \text{CM}$, we consider Spec to be sufficiently complete. Recall that in case of sound specifications this moreover guarantees $\sigma_{\mathbf{x}}$ to be a bug triggering input. In practice, $\text{CM}(P_{\text{gt}})$ can be constructed by modifying the reference implementation. In our evaluation, we use code mutants generated by ChatGPT tasked to find code completions for the natural language description φ_{nl} .

3.2 Prompt Design for LLM-Based Contract Inference

We next discuss our prompt design for instructing LLMs to generate program contracts from natural language descriptions. Figure 3 shows our prompt approach. The LLM is instructed to generate function contracts from developer hints and given context. A function contract specifies the behavior of a single function or method. Here, we specifically look at function contracts that specify the behavior of Python functions.

Prompt Construction. We construct the prompt for the LLM directly from existing code. For this, we take as input the function signature, a natural language docstring, and the surrounding file context (if available). We then use this information to automatically construct the prompt shown in Figure 3(a). In practice, such information is often not readily available. Therefore, when employing NL2CONTRACT approaches for finding bugs in real world code with automatic software verifiers, we parse this information directly from source code. Given a function under test, we first identify all functions (methods) called and all variables accessed in the global or class context. Then, we

```
You have the following code, including a function stub and docstring
for <function_name>:
...
<File context if present>

<function_name signature>
<natural language docstring>
...

Write a function contract for checking that the function
<function_name> is implemented correctly. Write the function
contract in Python by using Python assert statements. Specify
preconditions in a try-catch block and postconditions
as assert statements. Include everything in a contract function
called '<function_name>_contract'. Infer necessary type
hints for all input arguments.
```

(a) Prompt Template

```
def <function_name>_contract(... TYPED PARAMETERS ...):
    try:
        ... PRECONDITIONS USING ASSERTS ...
    except AssertionError as e:
        raise ValueError(f"Precondition failed: {e}") from e

    result = <function_name> (... ARGUMENTS ...)

    ... POSTCONDITIONS USING ASSERTS ...

    return result
```

(b) Function Contract

Fig. 3. Prompt template for generating functional contracts from natural language descriptions. Dark gray parts are automatically filled in during prompt construction. Violet parts are inferred by the LLM from the given context.

construct the file context by (1) including variable definitions of variables defined in the global or class context, (2) signatures and documentation of called methods and functions, and (3) the header of the class (if included in a class). We have found that including this information is important for LLMs to generate valid contracts – otherwise LLMs tend to hallucinate non-existing code.

Function Contracts in Python. While there are many attempts to bring formal contracts into the Python language such as `icontract` [40], `deal` [52] or the abandoned [PEP 316](#), there is still only limited support from automatic verification tools. Note in particular that there is no natural support of assume statements like we used for illustration in the example of Figure 1. For this reason, we decided to encode pre- and postconditions as pure Python assertions which allows us to use software verifiers subsequently checking for assertion violations. Our encoding is shown in Figure 3(b). The contract is encoded as Python function that takes in the same parameters as the original function (and calls it). The encoding should guarantee that an assertion exception (hinting to a specification violation) occurs for inputs which satisfy the precondition but where the function's returned value (in Figure 3(b): `result`) does not meet the postcondition. To this end, preconditions are encoded as Python assert statement that are wrapped in a try-catch block (try-except). The purpose of the try-catch block is to prevent precondition violations to lead to assertion errors. The function then instead raises a `ValueError` and stops. The postconditions can directly be encoded as assert statements which are dependent on the input and the result of the original function. The function hence raises an assertion error only for function inputs that satisfy the precondition (and are hence valid) and violate the postcondition which exactly coincides with our definition of specification violation.

In practice, we provide this function contract template as part of the prompt by filling in the function name of the original function. The LLM then infers the missing preconditions (which also includes the types of function parameters) and postconditions from the given natural language context.

Function Contracts in other programming languages. Our encoding of functional contracts can be easily adopted for programming languages that support exception handling. In particular in Java, pre- and postconditions can be encoded as Java assertions (which are available since Java 1.4) and the whole contract as a Java method. `IllegalArgumentException` can be raised for precondition violations which is a native way to indicate invalid inputs. In general, we see our encoding of functional contracts as a feasible way to evaluate LLM's ability to generate functional

contracts across many programming languages. We however focus in our evaluation on Python, and leave the evaluation in other languages open for future work.

4 Evaluation

We perform a systematic study to evaluate the ability of LLMs to infer specifications from natural language intent on popular benchmarks. During our evaluation, we thoroughly evaluate whether LLMs can generate *sound* and *complete* specifications that enable the automatic detection of software bugs with automatic software verifiers. Our evaluation is guided by the following research questions:

RQ1 How effective are LLMs in generating *sound* functional contracts?

RQ2 How effective are LLM-generated specifications in *discriminating* buggy and correct behavior?

RQ3 Are LLM-generated specifications useful for finding real bugs *automatically*?

In **RQ3**, we explore the usability of LLM-generated specifications in different verification scenarios, including their potential for automatic testing.

4.1 Experimental Setup

In the following, we describe the main experimental setup for **RQ1** and **RQ2**. We see *nl2postcond* as our main baseline, but we compare also with other baselines on the task of bug detection in our experiments for **RQ3**. Our experimental setup for **RQ3** is provided in Section 5.3.

Benchmark. We adopt HUMANEVAL⁺ [29] as our main benchmark. HUMANEVAL⁺ includes 164 Python problems, each consisting of a function stub, a natural language description encoded as a Python docstring, a reference implementation, and some validating tests. We choose HUMANEVAL⁺ as it updates the popular HUMANEVAL benchmark [9] with (1) a more extensive test suite (775 tests per problem on average), (2) corrected reference implementations, and most importantly (3) *reference preconditions* implemented as Python asserts for each problem in the benchmark. The preconditions are non-trivial: 92 out of 164 tasks contain preconditions that are not type related (e.g. `assert not is_prime(n)`). A total of 46 out of 164 tasks are annotated with preconditions that use functions from the Python standard library (e.g. `assert set(a).issubset({"0", "1"})`) and 5 out of 164 tasks are annotated with loops, conditional statements, or introduce helper functions to encode preconditions. In addition, 104 out of 164 tasks do not provide a type hint for input parameters in the function stub, which makes them generally challenging for automatic software verification. Based on these observations, we see HUMANEVAL⁺ as a demanding benchmark that allows to evaluate whether LLMs can generate functional contracts from natural language descriptions that are also *sound* with respect to the reference solution and precondition.

Large Language Models. We generate functional contracts with recent large language models that have shown strong performance on various programming tasks. We include both open-source and closed source models:

GPT-5 (Chat) and **GPT-4o** are the most recent version GPT series of chat models provided by OpenAI. GPT-5 shows strong performance on existing coding benchmarks [24].

CodeQwen 2.5 CodeQwen 2.5 (32B) is the currently best 32B parameter open-source coding model² that still fits on consumer hardware. We use a variant of CodeQwen 2.5 that is tuned for instruction following which supports the chat format for assistance in coding.

We use the same prompt for all coding models and use the OpenRouter API³ to query all models. We see the performance of these models as a baseline performance which can likely be improved by further prompt tuning or other more advanced techniques [53].

²<https://evalplus.github.io/leaderboard>, accessed in September 2025.

³openrouter.ai

Contract Generation. For each HUMANEVAL⁺ problem, we generate 10 functional contracts per problem and model. We use the default temperature of 0.7 as this has been found to be reasonable for code generation tasks. As our baseline prompt, we employ *nl2postcond* [14]. As *nl2postcond* instructs the LLM to generate postconditions, we wrap the generated postconditions in a contract function to be evaluated in our setup:

```
def <function_name>_contract(<original function_signature>):
    return_value = <function_name> (<arguments>)
    ... POSTCONDITION GENERATED BY nl2postcond ...
    return return_value
```

We automatically fill in the function name and signature based on the given programming task. Note that postconditions generated by LLMs with the *nl2postcond* prompt expect the variable `return_value` to contain the result of the computation which we compute by calling the original function. We view the function signature as a *trivial precondition* that can easily be computed from the function code. The verifier will use the type hints to restrict the search space for finding specification violations. Finally, we generate in total 9,840 functional contracts across all models, prompt variants (NL2CONTRACT and *nl2postcond*), and programming problems.

Verification. To verify the soundness and completeness of the generated specifications, we employ automatic verification tools. There are currently only a few formal software verifiers [13, 15, 42] that support the verification of Python code. In our experiments, we specifically decided for CrossHair [42] which we run with the analysis option `--analysis_kind=asserts`⁴. CrossHair is a symbolic execution engine for Python and it supports the verification of programs with basic Python types. We choose CrossHair because it is capable enough for verifying HUMANEVAL tasks and because it is generally effective in finding assertion errors in Python programs. However, CrossHair – as most automatic verifiers for Python – is *incomplete*, i.e., it reports unknown results if it cannot prove the complete input space to be safe. Therefore, we adopt a more pragmatic approach and declare a specification as unsound if CrossHair finds a specification violation on the reference implementation in 60s and sound otherwise.

Code Mutants. To evaluate specification completeness, we employ the same set of code mutants *CM* as used in the evaluation of *nl2postcond* postconditions. The set contains 4 to 233 code mutants per HUMANEVAL task with a median of 55 mutants per task. The mutants are generated as code completions from an earlier version of ChatGPT. Each code mutant comes with a set of bug triggering inputs, i.e. *valid* inputs for which the mutant behaves differently than the reference implementation.

Metrics. Our key metrics for the evaluation of LLM inferred specifications are soundness and completeness, checked via verifiers according to our description in Section 3.1 (thus called *verification soundness* and *verification completeness*). For a set of n randomly generated specifications, we compute *sound@k* as a *pass@k* style metric [9] for $1 \leq k < n$ that measures the statistical expected value that a random sample of k specifications contains at least one verification sound specification. This is in line with the *accept@k* metric used in the *nl2postcond* evaluation to estimate test-set correctness. For completeness, we compute *%complete* as the expected value that a verification sound specification is also verification complete, i.e., the verifier kills all mutants given the specification. For our comparison with *nl2postcond*, we furthermore employ *test-set correctness* and *bug*

⁴CrossHair expects that all functions to be verified are marked with an assert statement in the beginning and it reports all types of exceptions as errors. Hence, when we run our contracts with CrossHair, we preprocess them by adding a trivial assertion (`assert True`) in the beginning and we replace the raising of `ValueError` with a return statement.

Table 1. Comparison of different LLMs evaluated with the NL2CONTRACT and *nl2postcond* prompt with respect to test-set correctness and verification soundness, all in %. The best results per model are highlighted in bold.

Model	Prompt	Test-set correctness			Verification soundness		
		<i>accept@1</i>	<i>accept@5</i>	<i>accept@10</i>	<i>sound@1</i>	<i>sound@5</i>	<i>sound@10</i>
GPT-5 (Chat)	<i>nl2postcond</i>	86.6	94.0	95.7	13.0	14.3	15.2
	NL2CONTRACT	89.0	94.6	95.1	81.1	88.7	90.0
GPT-4o	<i>nl2postcond</i>	75.9	92.2	95.1	13.4	16.6	18.3
	NL2CONTRACT	80.5	93.5	96.3	72.8	87.9	91.5
CodeQwen 2.5	<i>nl2postcond</i>	59.1	80.1	85.4	11.5	15.5	17.1
	NL2CONTRACT	73.6	94.8	97.0	66.7	88.7	93.9

completeness as defined by Endres et al. [14]. These metrics are computed by using the set of test inputs associated with each HUMAN-EVAL⁺ task instead of employing a verifier to check *all* inputs.

5 Results

We now discuss the results of our experimental evaluation per research question.

5.1 RQ1: Ability of LLMs to Generate Sound Functional Contracts

To answer **RQ1**, we evaluate the effectiveness of LLMs to generate sound functional contracts. We compare models and prompts with respect to test-set correctness and verification soundness.

Results. Table 1 summarizes our experimental results. Overall, we find that for HUMAN-EVAL⁺:

LLMs are effective in generating test-set correct specifications from natural language. We observe that the evaluated LLMs produce test-set correct functional contracts in between 73.6% to 89.0% of all cases (*accept@1*). If we sample more specifications, it is highly likely that the generated functional contract is test-set correct (up to 97% for *accept@10*). When comparing the same LLM with different prompt variations, we find that the performance of the OpenAI models do not suffer from performing the more complex tasks of contract generation. In fact, for *accept@1*, they perform significantly better in generating test-set correct functional contracts with NL2CONTRACT than generating a test-set correct postcondition with *nl2postcond*. Surprisingly, the open-source model CodeQwen 2.5 performs in general better in generating test-set correct contracts than generating test-set correct postconditions. We find that LLMs prompted to generate functional contracts tend to split postconditions into multiple simpler assertions which are more often test-set correct.

Test-set correctness does not imply verification soundness. We observe that the task of generating verification sound specifications (right hand side of Table 1) is significantly more challenging than producing test-set correct specifications. The performance of LLMs prompted with *nl2postcond* drop drastically from 73.6% – 89.0% *accept@1* test-set correctness to 11.5% – 13.0% *sound@1* when evaluated with respect to verification soundness. LLMs prompted to generate functional contracts with NL2CONTRACT achieve a verification soundness score *sound@1* of 66.7% to 81.1%. By increasing the sampling budget, LLMs become more effective in generating verification sound functional contracts (*sound@5* of 87.9% to 88.7% and *sound@10* of 90.0% to 93.9%). To better understand why LLMs are more effective in generating verification sound functional contracts, we analyze the distribution of soundness violations. Figure 4a shows the percentage of specifications (out of all generated specifications) that are unsound due to a precondition violation for each model and prompt. We observe that most specifications generated with *nl2postcond* (75.8% to 82.0%) are unsound due to violations of condition (S1) of Section 2, i.e. they allow *invalid* inputs that violate the reference precondition. LLMs prompted with NL2CONTRACT are better in capturing the ground

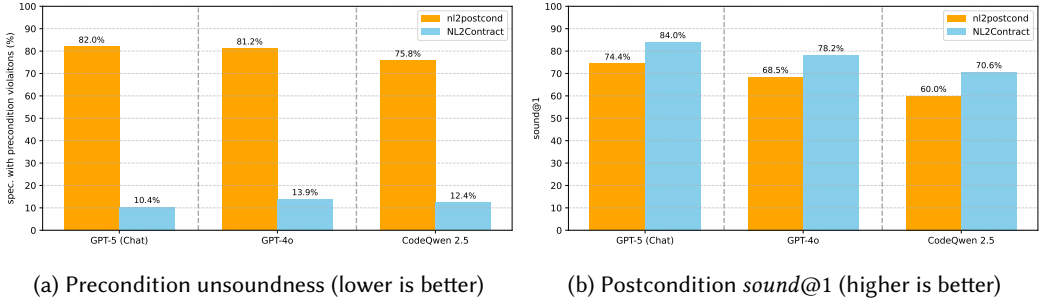


Fig. 4. Comparison of LLMs under different prompts. Left (a) shows the percentage of specifications that are unsound due to violations of reference precondition. Right (b) shows *sound@1* of the postconditions.

truth precondition. Only 10.4% to 12.4% of the NL2CONTRACT specifications are unsound due to violations of the reference precondition. We will see in Section 5.3 that this has a significant practical impact: Software verifiers that are supplied with verification unsound specification often produce a significant higher number of false alarms. Only verification soundness indicates this shortcoming of *nl2postcond* postconditions. This shows the importance of quality metrics in software verification that evaluate the soundness of specifications for *all* possible inputs.

Postcondition soundness. We also explore and compare the verification soundness of the generated postconditions. To this end, we add the reference precondition as an additional precondition to the generated contracts, both for NL2CONTRACT and *nl2postcond*. This ensures that the verifier only generates *valid* inputs when validating the contracts. Hence, the contracts can only be unsound due to violations of condition (S2), i.e. the postcondition fails on the reference implementation given a valid input. Our *sound@1* results for this experiment are shown in Figure 4b. LLMs prompted for NL2CONTRACT achieve a postcondition soundness score *sound@1* score of 70.6% to 84.0%. This is on average 9,6% to 10,6% higher than the *sound@1* score achieved by LLMs prompted with *nl2postcond*.

Based on these results, we conclude for **RQ1**:

On HUMANEVAL⁺ tasks, LLMs are effective in generating *test-set correct* and *verification sound* functional contracts from informal natural language specifications. LLMs with NL2CONTRACT are significantly better in producing verification sound specifications than LLMs prompted for *nl2postcond*. LLMs prompted with NL2CONTRACT are effective in capturing input assumptions.

5.2 RQ2: Ability to Discriminate Buggy and Correct Behavior

To answer **RQ2**, we evaluate whether the generated *verification sound* specifications can discriminate buggy and correct behavior. We compare models and prompts with respect to (test-set) bug completeness and verification completeness.

Results. Table 2 summarizes our experimental results. Overall, we find that for HUMANEVAL⁺ mutants:

LLMs can generate bug discriminating functional contracts from natural language. We find that LLMs prompted with NL2CONTRACT can generate at least one verification sound functional contract that is able to kill all code mutants for 80 to 92 HUMANEVAL⁺ tasks (48.8% to 56.1% of all tasks). Surprisingly, GPT-4o performs better in generating bug complete specifications for a diverse set of tasks, producing at least one verification sound bug complete functional contract for 92 tasks.

Table 2. Comparison of different LLMs evaluated with the NL2CONTRACT and *nl2postcond* prompt with respect to bug completeness and verification completeness. *%complete* is the percentage of verification sound specification that are also complete (kills all mutants). $x/164$ is the number of tasks (out of 164) with at least one complete specification. *%killed* is the percentage of mutants killed by the verifier with at least one verification sound specification. The best results per model are highlighted in bold.

Model	Prompt	Bug complete		Verification complete		Mutants
		% complete	$x/164$	% complete	$x/164$	% killed
GPT-5 (Chat)	<i>nl2postcond</i>	7.3	15	4.8	11	11.3
	NL2CONTRACT	39.5	83	29.9	68	73.0
GPT-4o	<i>nl2postcond</i>	4.8	15	3.8	11	12.8
	NL2CONTRACT	31.4	92	25.2	69	73.7
CodeQwen 2.5	<i>nl2postcond</i>	4.6	16	2.8	11	11.4
	NL2CONTRACT	18.4	80	12.9	58	69.1

However, our results also indicate that GPT-5 (Chat) is more reliable in generating verification sound bug complete specifications: 39.5% of the GPT-5 (Chat) generated functional contracts are verification sound *and* bug complete. While the open-source model CodeQwen 2.5 achieves a lower bug completeness score, 18.4% of the generated specifications are still verification sound and bug complete. When prompted with *nl2postcond*, LLMs perform significantly worse producing a verification sound bug complete postcondition in 4.6% to 7.3% of all cases. This is expected as LLMs prompted with *nl2postcond* are in general less effective in producing verification sound formal specifications (as we have seen in **RQ1**).

Bug completeness overestimates verification completeness. Recall that completeness evaluates the effectiveness of specifications to catch code mutants via specification violations. Bug completeness evaluates completeness by testing the specification with a set of bug-triggering inputs. Verification completeness, in contrast, uses a software verifier to generate bug triggering inputs. Because of this difference, we find that bug completeness overestimates verification completeness in practice: LLMs produce less verification sound specifications that are useful for the software verifier to detect all code mutants. Still, LLMs prompted with NL2CONTRACT are more effective in generating verification sound and verification complete specifications than LLMs prompted with *nl2postcond*. In fact, between 12.9% and 29.9% of the LLM generated functional contracts are verification sound and verification complete while only 2.8% to 4.8% of *nl2postcond* postconditions are verification complete. In addition, LLMs prompted with NL2CONTRACT can generate verification sound functional contracts that let the software verifier detect all code mutants for 58 to 69 HUMANEVAL⁺ tasks (35.4% to 42.1% of all tasks).

Mutation score. We also evaluate the percentage of mutants killed by the verifier when provided with a verification sound specification. Our results are also shown in Table 2 (right hand side). We find that the software verifier given a verification sound LLM-generated functional contract can discriminate over three-quarters of the buggy code mutants.

Postcondition completeness. For a fair comparison between *nl2postcond* and NL2CONTRACT, we evaluate the completeness of the generated postconditions. For this, we again add the reference precondition to the generated contracts as described before. We thus consider all contracts that are verification sound for *valid* inputs, i.e. those that are verification sound under the reference precondition. Our results for postcondition completeness are shown in Figure 5. In Figure 5a, we compare the bug completeness of the postconditions generated by LLMs prompted with *nl2postcond*

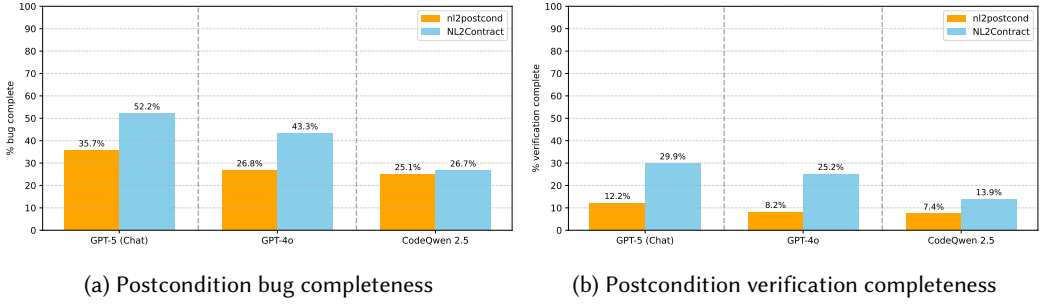


Fig. 5. Comparison of LLMs under different prompts with respect to postcondition completeness.

and NL2CONTRACT. We find that LLMs prompted to generate functional contracts are more effective than the same LLMs prompted to generate postconditions (with up to 18.5% more specifications being bug complete). This also carries over to verification completeness (see Figure 5b). Here, 13.9% to 29.9% of the postconditions generated by LLMs prompted for NL2CONTRACT are verification complete while only 7.4% to 12.2% of postconditions generated by LLMs prompted for *nl2postcond* are verification complete.

Based on our results, we conclude for **RQ2**:

On the HUMANEVAL⁺ benchmark, LLMs are more effective in generating functional contracts that can discriminate between buggy and correct behavior. In addition, when provided with NL2CONTRACT contracts, the software verifier can *automatically* detect up to 73% of all mutants, allowing an automatic verification of up to 69 out of 164 tasks.

5.3 RQ3: Finding Real Bugs with Automatic Verifiers

For **RQ3**, we evaluate whether LLM generated functional contracts can be used to identify real software bugs with the help of automatic verification techniques. We are not only considering formal verification tools, but we also evaluate the potential of the generated contracts for automatic testing.

Python-by-Contract. To evaluate on real world bugs, we employ the *Python-by-Contract* dataset [60]. The dataset collects Python solutions for 55 introductory tasks from Advent of Code 2020 and the introductory programming course at ETH Zurich in Fall 2019. Each solution is manually annotated with reference functional contracts. In addition, the authors provide 59 *buggy* solutions that represent mistakes made during the development or specification process. We find that 31 bugs are due to implementation mistakes in functions and methods from which 19 can be confirmed by CrossHair with the reference specification. We choose Python-by-Contract because (1) it provides *realistic* buggy solutions together with a reference implementation, (2) the tasks are often non-trivial implementing between 1 to 10 functions per problem, and most importantly (3) they provide a reference specification which we use to derive the ground truth preconditions from. To construct the dataset, we pair each buggy solution with its reference implementation and a reference precondition computed from the reference specification. In the process, we remove the original specifications and all comments hinting at the bug location.

Experimental setup. We compare specifications generated by LLMs prompted with NL2CONTRACT and *nl2postcond*. To construct the prompts, we follow the process described in Section 3.2. In particular, we remove the body of all functions and methods implemented in the buggy solution and prompt the LLMs based on the provided natural language hints (that are already available in

Table 3. Comparison of different LLMs and prompts on the Python-by-Contract dataset. %detecting is the percentage of specifications that enable the verifier (or tester) to detect the bug by reporting a bug triggering specification violation. #bugs is the number of bugs (out of 31) where at least one bug triggering input is found by the verifier or tester.

Model	Prompt	<i>sound@1</i>	CrossHair		Pynguin	
			% detecting	# bugs	% detecting	# bugs
GPT-5 (Chat)	<i>nl2postcond</i>	20.0	4.2	3	16.1	6
	NL2CONTRACT	71.3	39.4	14	15.8	8
GPT-4o	<i>nl2postcond</i>	19.7	5.8	5	15.8	9
	NL2CONTRACT	67.7	34.8	14	14.8	7
CodeQwen 2.5	<i>nl2postcond</i>	21.0	6.8	4	16.1	7
	NL2CONTRACT	64.8	35.2	14	15.2	7

code). LLMs are prompted to generate specifications for functions that are modified in the solution process. We use the reference implementation and precondition to compute the *sound@1* score. To find real bugs automatically, we employ different *automatic* tools with the generated specifications. In the process, we say that a bug is detected if the tool given a generated specification reports a *bug triggering* violation $\sigma_{\mathbf{x}}$ on the buggy solution. A violation is bug triggering if the buggy implementation behaves differently than the reference implementation for the given input, i.e. $Pre_{gt}(\sigma_{\mathbf{x}}) \wedge f_P(\sigma_{\mathbf{x}}) \neq f_{gt}(\sigma_{\mathbf{x}})$.

Verifiers and Testers. We are not only considering CrossHair as our main verifier, but also evaluate Pynguin [30] which is a popular automatic testing tool for Python. Pynguin implements a search-based coverage-guided testing strategy and is typically used for generating regression tests. To utilize Pynguin, we exploit the properties of our functional contracts (see Section 3.2): The contract defines a contract function that only raises assertion errors when the test input leads to a specification violation. Hence, when the program is instrumented with the generated contract function, we can detect specification violations by running the contract function on different test inputs. In this case, we employ Pynguin to generate random test inputs and report any test case that leads to an assertion error for the contract function. We run Pynguin with a 600s timeout.

Results. Our results are shown in Table 3. Overall, we find that for the Python-by-Contract bugs:

LLMs can generate specifications that help to catch real bugs automatically. Supplied with NL2CONTRACT generated specifications, CrossHair is able to automatically detect 14 out of 19 bugs that it is capable to find. In general, the LLMs prompted with NL2CONTRACT produce in 34.8% to 39.4% of all cases a specification that enables the verifier to detect the bug. GPT-5 (Chat) is more effective than the other LLMs in generating sound specifications, which more often enables generation of a bug triggering inputs. CodeQwen 2.5 is surprisingly competitive with the other larger LLMs, allowing the detection of the same number of bugs with a model that can run locally.

Functional contracts are necessary for avoiding false alarms. In comparison with *nl2postcond* postconditions, LLMs with NL2CONTRACT produce specifications that are significantly more useful in software verification: The software verifier supplied with NL2CONTRACT based specifications finds 9 to 11 bugs more than when supplied with *nl2postcond* postconditions. The key reason for this is the higher verification soundness of the NL2CONTRACT specifications. Therefore, in practice, verifiers supplied with NL2CONTRACT produce a significant lower number of false alarms, as shown in Figure 6a. Here, we report the percentage of specifications that lead to an alarm raised by CrossHair on the buggy solutions. We find that most alarms reported by CrossHair given the

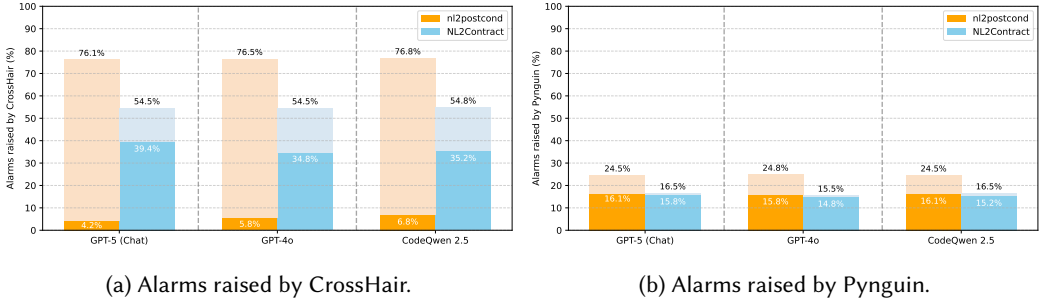


Fig. 6. Analysis of the impact of LLM generated specification on the performance of automatic tools. The lighter color shows the percentage of alarms raised by the different tools. The darker color shows the percentage of alarms (specification violations) that identify a real bug.

nl2postcond specifications are *false alarms*, i.e. specification violations that are not bug triggering. While CrossHair also reports false alarms with NL2CONTRACT specifications, most of the found specification violations uncover a real bug. This showcases the higher usability of NL2CONTRACT specifications in the context of software verification.

Contract functions for random testing. Recall that we express functional contracts as contract functions which can be evaluated by executing the code on different input. We utilize this property to evaluate whether the generated contract functions are also useful for automated random testing (right hand side of Table 3). By injecting the contract functions into the program, we find that Pynguin (without any modification) can effectively find up to 9 out of 31 bugs. LLMs prompted with *nl2postcond* perform slightly better in the context as Pynguin spends a significant amount of its time in covering the NL2CONTRACT preconditions. Still, when Pynguin finds a specification violation, we find that they are more often bug triggering for NL2CONTRACT contracts, as shown in Figure 6b. We see this as an indication that NL2CONTRACT contracts can be promising in the context of automated testing, but we leave the exploration of specification formats tailored for testing and evaluations with property-based testing methods [32] open for future work.

Finally, we conclude for **RQ3**:

For Python-by-Contract bugs, the NL2CONTRACT specifications are more *useful* for finding bugs automatically with software verifiers than raw *nl2postcond* postconditions. Software verifiers supplied with NL2CONTRACT specifications can not only detect up to 14 out of 31 bugs automatically, but also produce significantly less false alarms in the process.

6 Discussion

To gain further insights into how NL2CONTRACT function contracts enable the detection of real bugs with automatic tools, we conduct a qualitative evaluation of specifications generated by LLMs that led to successful bug detection and that allow false alarms. We discuss two cases in the following that are representative for the strengths of NL2CONTRACT and LLM-based specification inference in general.

Preconditions are important for real bug detection. While we already showcased an example where preconditions matter for bug detection in our introduction, we now consider a real bug from the Python-by-Contract dataset in Figure 7. The goal of the benchmark task is to implement a conversion from a 24h clock into angles of clock hands. Figure 7(a) shows a buggy solution where the implementer forgot to handle the time after 12 clock hours and Figure 7(b) provides the contract

```
def compute_angles(hour: int, minute: int, second: int)
    -> Tuple[float, float, float]:
    """Compute the angles of the clock hands for a given time of the
    day."""
    angle_second = second / 60 * 360
    angle_minute = (minute + second / 60) / 60 * 360

    # ERROR
    # I forgot to distinguish between <12 and >= 12 clock hours.

    angle_hour = (hour + minute / 60 + second / 3600) / 12 * 360

    return angle_hour, angle_minute, angle_second
```

(a) Buggy solution

```
def compute_angles_contract(hour: int, minute: int, second: int)
    -> Tuple[float, float, float]:
    try:
        # Preconditions
        assert 0 <= hour <= 23
        assert 0 <= minute <= 59
        assert 0 <= second <= 59
    except AssertionError as e:
        raise ValueError(f"Precondition failed: {e}") from e
    result = compute_angles(hour, minute, second)
    ...
    # Check angles are within valid bounds
    for angle in result: assert 0.0 <= angle < 360.0
    ...
    return result
```

✗ Violation when hour == 12

(b) Function Contract

Fig. 7. Example of buggy solution from the Python-by-Contract dataset and a simplified contract generated by GPT-5 (Chat) with NL2CONTRACT. The grayed out part is *not* provided to the LLM. Given the specification, CrossHair correctly identifies the bug when calling `compute_angles(12, 0, 0)`.

generated by GPT-5 (Chat) which we simplified for demonstration. During specification inference, the LLM is only provided with the function signature and docstring, while the implementation is hidden from the LLM. Given the specification, CrossHair correctly reports a bug triggering specification violation for the input `compute_angles(12, 0, 0)`. We find that this exemplifies a case where generating preconditions is of particular importance: The precondition ensures that all inputs follow the format of 24h clock. GPT-5 (Chat) prompted with *nl2postcond* does not encode this information in the generated postcondition:

```
assert all(0.0 <= angle < 360.0 for angle in return_value)
```

Although the postcondition is bug discriminating for valid inputs, CrossHair reports a postcondition violation for the input `compute_angles(-1, -1, -1)`. However, the behavior of `compute_angles(-1, -1, -1)` is undefined which results into a false alarm, i.e. the buggy solution and reference implementation behave identical for the given input. In practice, such bug reports by automatic tools are unhelpful and might lead the developer away from detecting the real bug. Therefore, to really support the developer in bug detection, we need tools that only report bugs for valid inputs. As determining the validity of inputs is non-trivial, the example demonstrates the importance of preconditions for real bug detection with automatic verification tools and the potential of NL2CONTRACT to generate sound preconditions from natural language hints.

LLM inferred specifications can help to uncover inconsistency bugs automatically. During our investigation, we found an inconsistency bug which we reported to the Python-by-Contract dataset developers and which is shown in Figure 8. In this example, the developer implements an approximation of the `sqrt` function which should be precise up to `eps`. Both NL2CONTRACT and *nl2postcond* correctly capture this property with the postcondition `abs(result * result - c) < eps`. CrossHair however reports a specification violation for the reference implementation because the reference implementation only ensures that `abs(result * result - c) <= eps` holds, i.e. it computes `approximate_sqrt(2, 0.25) = 1.5` which is too imprecise. This showcases the potential of specifications inferred from natural language descriptions. When confronted with this inconsistency bug, the developer can decide whether it represents a simple mistake in the documentation or a serious implementation flaw. This can help the developer to make the natural language description more consistent with the implementation or fix the bug. Both tasks are important for designing effective and reliable software systems [46]. Overall, we see NL2CONTRACT as an important step towards verifying software with respect to natural language specifications already available in code which cannot only help us to uncover real bugs, but also inconsistencies with the documentation.

```

"""
Approximate the square root of non-zero positive integer ``c``...
The result should be precise up to ``eps``: ``abs(sqrt(c) - c) < eps``.
"""

# Reference postcondition: abs(result * result - c) <= eps
def approximate_sqrt(c: int, eps: float) -> float:
    """Approximate the square-root of c up to the precision eps."""

```

(a) Reference docstring and function stub

```
assert abs(result * result - c) < eps
```

(b) Relevant postcondition generated by NL2Contract and nl2postcond

✗ Violation when $c == 2$ and $\text{eps} == 0.25$

(c) Violation reported by CrossHair

Fig. 8. An inconsistency bug in the Python-by-Contract dataset.

7 Threats to Validity

We have conducted our evaluation on two popular benchmarks: HUMANEVAL⁺ and Python-by-Contract. Although used frequently in the evaluation of LLMs (especially HUMANEVAL⁺), the benchmarks focus on simple Python programs which often do not have dependencies beyond the scope of a single file. Therefore, our evaluation results might not generalize to other more complex (and potentially undocumented) real world Python code. Another potential threat is the risk of data leakage. Because the Python-by-Contract dataset and HUMANEVAL⁺ are public benchmarks, the underlying LLM might have seen and memorized the benchmark tasks. Our evaluation partially mitigates this risk by generating contracts in custom Python-based format. Python-by-Contract uses a more specific annotation language and we are not aware of a dataset with complete specification-annotated HumanEval tasks. For Python-by-Contract, there is still the risk that we measure the transfer of memorized specifications to our custom format. Another potential limitation is the underlying software verifier. The software verification community has in the past mostly focused on statically typed compiled languages such as C or Java. Therefore, software verification tools for more dynamic languages such as Python are currently not as mature as their static counterpart. Our evaluation results still show that using an existing software verifier can help for finding real bugs in Python programs. We still expect that usability of NL2CONTRACT contracts will increase with future, more mature verification tools for Python.

8 Related Work

In this work, we revisit the problem of specification inference from natural language descriptions, already available in code. In the process, we propose the task NL2CONTRACT which evaluates the effectiveness of LLMs to generate sound and bug discriminating specifications useful for software verification. In the following, we discuss the most closely related approaches.

Specification Generation. Formal specifications play a critical role in many areas of software engineering, including code generation [2, 28, 56], software testing [17, 32, 51], and formal verification [4–6, 19, 25, 36]. They are used to describe the intended behavior of a system, including functional relationships of input and outputs, as well as invariants over the internal state. However, writing or obtaining formal specifications for a system under test is generally considered a hard task [10]. Therefore, there exists a large body of research automating the generation of formal specifications [1, 3, 11, 16, 18, 23, 34, 41, 43, 54]. Many existing works focus on loop invariant generation [16, 23, 41, 43], while others have attempted the generation of preconditions [3, 11, 18], postconditions [1, 34, 54], and assertion-based invariants [48]. While useful for building regression oracles [57], most of these techniques focus on capturing the existing implemented behavior, instead of the intended behavior, limiting their application for bug detection. Our goal in contrast is to generate formal specifications of the intended program’s behavior from natural language descriptions, which are useful for bug detection via verification. Several approaches have been proposed that, similar to our work, aim to generate specifications from natural language [7, 8, 26, 35, 46, 47, 58, 61].

Many of these techniques rely on pattern-matching and hand-crafted rules. For example, Jdoctor [7], icodecomment [46], and @tComment [47] use pattern-matching techniques to extract specifications from Javadoc comments. This often restricts their application to semi-structured natural language formats, making their adoption for general specification inference from natural language hints in code challenging. In contrast, NL2CONTRACT utilizes the code understanding capabilities of LLMs to infer functional contracts from the few natural language hints that are already available in code.

LLMs for Specification Generation. Several approaches have been proposed to synthesize program specifications using LLMs [12, 14, 20, 31, 39, 49, 50, 55]. Most of these approaches are predominantly used for testing. AthenaTest [49] uses machine learning to generate both the input and the oracle of a unit test. TOGA [12], TOGLL [21], and Doc2OracLL [22] generate exception and assertion based test oracles. TiCoder [27] leverages LLMs to generate test inputs and expected outputs. While useful for testing, the generated test oracles are often limited to describing the expected behavior for a specific test input (test prefix). Our goal are specifications of the expected behavior that are valid for all inputs. There also exist more recent approaches that utilize LLMs for generating formal specifications such as AutoSpec [55] and SpecGen [31]. They use LLMs in combination with automatic software verifiers to infer and refine specifications until the specification is consistent with the current implementation. As a consequence, the generated specifications capture the implemented behavior of the existing (buggy) code precisely, making bug detection impossible. In contrast, we propose NL2CONTRACT as a way to measure the ability of LLMs to infer the desired and expected behavior from natural language. Our experiments demonstrate that functional contracts derived from natural language hints allows to find novel bugs, which are inconsistencies between the documentation and the actual implementation, with the help of automatic verification tools. Most closely related to our work is *nl2postcond* [14] which evaluates the ability of LLMs to generate postconditions describing the intended code behavior from natural language elements. NL2CONTRACT extends this task to the generation of formal contract. Our experiments show that LLMs prompted with NL2CONTRACT can generate sound functional contracts which are more useful than raw postconditions for the detection of real bugs with an automatic software verifier.

9 Conclusion

In this work, we introduce NL2CONTRACT as the task to infer functional contracts from natural language descriptions via LLMs. We evaluate the capabilities of LLMs to generate sound functional contracts – consisting of both pre- and postconditions – useful for finding software bugs with automatic verification tools. We formally define key quality indicators of the specifications such as verification soundness and verification completeness that we use for evaluating the quality of the generated specifications. Our evaluation shows that LLMs can translate natural language descriptions into non-trivial contracts that accurately capture the developer’s intent. Our study further reveals that LLMs can generate meaningful preconditions which allows an automatic software verifier to find 14 real world bugs in the Python-by-Contract benchmark, while emitting significantly fewer false alarms. Our research shows the potential of LLMs to generate formal specifications from natural language, which ultimately can enable software validation and bug detection with automatic verification tools.

10 Data Availability

We plan to make all implementations and data publicly available and archived at Zenodo.

References

- [1] Anoud Alshnakat, Dilian Gurov, Christian Lidström, and Philipp Rümmer. 2020. Constraint-Based Contract Inference for Deductive Verification. In *Deductive Software Verification: Future Perspectives - Reflections on the Occasion of 20 Years of KeY*, Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, and Mattias Ulbrich (Eds.). Lecture Notes in Computer Science, Vol. 12345. Springer, 149–176. https://doi.org/10.1007/978-3-030-64354-6_6
- [2] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. IEEE, 1–8. <https://ieeexplore.ieee.org/document/6679385/>
- [3] Angello Astorga, P. Madhusudan, Shambwaditya Saha, Shiyu Wang, and Tao Xie. 2019. Learning stateful preconditions modulo a test generator. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 775–787. <https://doi.org/10.1145/3314221.3314641>
- [4] Thomas Ball and Sriram K. Rajamani. 2002. The SLAM project: debugging system software via static analysis. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, John Launchbury and John C. Mitchell (Eds.). ACM, 1–3. <https://doi.org/10.1145/503272.503274>
- [5] Dirk Beyer and M. Erkan Keremoglu. 2011. CPAchecker: A Tool for Configurable Software Verification. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6806)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer, 184–190. https://doi.org/10.1007/978-3-642-22110-1_16
- [6] Dirk Beyer and Jan Strejček. 2025. Improvements in software verification and witness validation: SV-COMP 2025. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 151–186.
- [7] 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*. 242–253.
- [8] Arianna Blasi, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Antonio Carzaniga. 2021. MeMo: Automatically identifying metamorphic relations in Javadoc comments for test automation. *J. Syst. Softw.* 181 (2021), 111041. <https://doi.org/10.1016/J.JSS.2021.111041>
- [9] 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).
- [10] Jennifer A. Davis, Matthew A. Clark, Darren D. Cofer, Aaron Fifarek, Jacob Hinchman, Jonathan A. Hoffman, Brian W. Hulbert, Steven P. Miller, and Lucas G. Wagner. 2013. Study on the Barriers to the Industrial Adoption of Formal Methods. In *Formal Methods for Industrial Critical Systems - 18th International Workshop, FMICS 2013, Madrid, Spain, September 23-24, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8187)*, Charles Pecheur and Michael Dierkes (Eds.). Springer, 63–77. https://doi.org/10.1007/978-3-642-41010-9_5
- [11] Elizabeth Dinella, Shuvendu K. Lahiri, and Mayur Naik. 2024. Inferring Natural Preconditions via Program Transformation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024, Porto de Galinhas, Brazil, July 15-19, 2024*, Marcelo d’Amorim (Ed.). ACM, 657–658. <https://doi.org/10.1145/3663529.3663865>
- [12] Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu K Lahiri. 2022. Toga: A neural method for test oracle generation. In *Proceedings of the 44th International Conference on Software Engineering*. 2130–2141.
- [13] Marco Eilers and Peter Müller. 2018. Nagini: A Static Verifier for Python. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018. Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10981)*, Hana Chockler and Georg Weissenbacher (Eds.). Springer, 596–603. https://doi.org/10.1007/978-3-319-96145-3_33
- [14] Madeline Endres, Sarah Fakhoury, Saikat Chakraborty, and Shuvendu K Lahiri. 2024. Can large language models transform natural language intent into formal method postconditions? *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1889–1912.
- [15] Bruno Farias, Rafael Menezes, Eddie B. de Lima Filho, Youcheng Sun, and Lucas C. Cordeiro. 2024. ESBMC-Python: A Bounded Model Checker for Python Programs. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, Maria Christakis and Michael Pradel (Eds.). ACM, 1836–1840. <https://doi.org/10.1145/3650212.3685304>
- [16] 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 Bodik and Rupak Majumdar

- (Eds.). ACM, 499–512. <https://doi.org/10.1145/2837614.2837664>
- [17] Marie-Claude Gaudel. 2007. Software Testing Based on Formal Specification. In *Testing Techniques in Software Engineering, Second Pernambuco Summer School on Software Engineering, PSSE 2007, Recife, Brazil, December 3-7, 2007, Revised Lectures (Lecture Notes in Computer Science, Vol. 6153)*, Paulo Borba, Ana Cavalcanti, Augusto Sampaio, and Jim Woodcock (Eds.). Springer, 215–242. https://doi.org/10.1007/978-3-642-14335-9_7
 - [18] Sandip Ghosal, Bengt Jonsson, and Philipp Rümmer. 2023. An Active Learning Approach to Synthesizing Program Contracts. In *Software Engineering and Formal Methods - 21st International Conference, SEFM 2023, Eindhoven, The Netherlands, November 6-10, 2023, Proceedings (Lecture Notes in Computer Science, Vol. 14323)*, Carla Ferreira and Tim A. C. Willemse (Eds.). Springer, 126–144. https://doi.org/10.1007/978-3-031-47115-5_8
 - [19] Matthias Heizmann, Yu-Fang Chen, Daniel Dietsch, Marius Greitschus, Jochen Hoenicke, Yong Li, Alexander Nutz, Betim Musa, Christian Schilling, Tanja Schindler, and Andreas Podelski. 2018. Ultimate Automizer and the Search for Perfect Interpolants - (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 10806)*, Dirk Beyer and Marieke Huisman (Eds.). Springer, 447–451. https://doi.org/10.1007/978-3-319-89963-3_30
 - [20] Soneya Binta Hossain and Matthew Dwyer. 2024. Togl: Correct and strong test oracle generation with llms. *arXiv preprint arXiv:2405.03786* (2024).
 - [21] Soneya Binta Hossain and Matthew B. Dwyer. 2025. TOGL: Correct and Strong Test Oracle Generation with LLMS. In *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025*. IEEE, 1475–1487. <https://doi.org/10.1109/ICSE55347.2025.00098>
 - [22] Soneya Binta Hossain, Raygan Taylor, and Matthew B. Dwyer. 2025. Doc2OracLL: Investigating the Impact of Documentation on LLM-Based Test Oracle Generation. *Proc. ACM Softw. Eng.* 2, FSE (2025), 1870–1891. <https://doi.org/10.1145/3729354>
 - [23] Christian Janßen, Cedric Richter, and Heike Wehrheim. 2024. Can ChatGPT support software verification?. In *Fundamental Approaches to Software Engineering - 27th International Conference, FASE 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings (Lecture Notes in Computer Science, Vol. 14573)*, Dirk Beyer and Ana Cavalcanti (Eds.). Springer, 266–279. https://doi.org/10.1007/978-3-031-57259-3_13
 - [24] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net. <https://openreview.net/forum?id=VTF8yNQm66>
 - [25] Alexey V. Khoroshilov, Vadim S. Mutilin, Alexander K. Petrenko, and Vladimir Zakharov. 2009. Establishing Linux Driver Verification Process. In *Perspectives of Systems Informatics, 7th International Andrei Ershov Memorial Conference, PSI 2009, Novosibirsk, Russia, June 15-19, 2009. Revised Papers (Lecture Notes in Computer Science, Vol. 5947)*, Amir Pnueli, Irina B. Virbitskaite, and Andrei Voronkov (Eds.). Springer, 165–176. https://doi.org/10.1007/978-3-642-11486-1_14
 - [26] Myeongsoo Kim, Davide Corradini, Saurabh Sinha, Alessandro Orso, Michele Pasqua, Rachel Tzoref-Brill, and Mariano Ceccato. 2023. Enhancing REST API Testing with NLP Techniques. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, René Just and Gordon Fraser (Eds.). ACM, 1232–1243. <https://doi.org/10.1145/3597926.3598131>
 - [27] Shuvendu K Lahiri, Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, Madanlal Musuvathi, Piali Choudhury, Curtis von Vech, Jeevana Priya Inala, Chenglong Wang, et al. 2022. Interactive code generation via test-driven user-intent formalization. *arXiv preprint arXiv:2208.05950* (2022).
 - [28] Yixuan Li, Julian Parsert, and Elizabeth Polgreen. 2024. Guiding Enumerative Program Synthesis with Large Language Models. In *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 14682)*, Arie Gurfinkel and Vijay Ganesh (Eds.). Springer, 280–301. https://doi.org/10.1007/978-3-031-65630-9_15
 - [29] 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. *Advances in Neural Information Processing Systems* 36 (2023), 21558–21572.
 - [30] Stephan Lukaczyk and Gordon Fraser. 2022. Pynguin: Automated unit test generation for python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 168–172.
 - [31] Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. 2024. Specgen: Automated generation of formal program specifications via large language models. *arXiv preprint arXiv:2401.08807* (2024).
 - [32] David Maciver and Zac Hatfield-Dodds. 2019. Hypothesis: A new approach to property-based testing. *J. Open Source Softw.* 4, 43 (2019), 1891. <https://doi.org/10.21105/JOSS.01891>
 - [33] Bertrand Meyer. 1992. Applying ‘design by contract’. *Computer* 25, 10 (1992), 40–51.

- [34] Facundo Molina, Marcelo d'Amorim, and Nazareno Aguirre. 2023. SpecFuzzer: A Tool for Inferring Class Specifications via Grammar-Based Fuzzing. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 2094–2097. <https://doi.org/10.1109/ASE56229.2023.00024>
- [35] 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>
- [36] Zvonimir Pavlinovic, Akash Lal, and Rahul Sharma. 2016. Inferring annotations for device drivers from verification histories. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 450–460. <https://doi.org/10.1145/2970276.2970305>
- [37] Rolf-Helge Pfeiffer. 2020. What constitutes software? An empirical, descriptive study of artifacts. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 481–491.
- [38] Reinhold Plosch. 1997. Design by contract for Python. In *Proceedings of Joint 4th International Computer Science Conference and 4th Asia Pacific Software Engineering Conference*. IEEE, 213–219.
- [39] Severin Primbs, Benedikt Fein, and Gordon Fraser. 2025. AsserT5: Test Assertion Generation Using a Fine-Tuned Code Language Model. In *IEEE/ACM International Conference on Automation of Software Test, AST@ICSE 2025, Ottawa, ON, Canada, April 28-29, 2025*. IEEE, 12–23. <https://doi.org/10.1109/AST66626.2025.00008>
- [40] Marko Ristin. 2021. icontract documentation - Recipes. <https://icontract.readthedocs.io/>.
- [41] 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>
- [42] Phillip Schanely. 2017. CrossHair: Symbolic Execution for Python. <https://github.com/pschanely/CrossHair>.
- [43] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. 2018. Learning Loop Invariants for Program Verification. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett (Eds.). 7762–7773. <https://proceedings.neurips.cc/paper/2018/hash/65b1e92c585fd4c2159d5f33b5030ff2-Abstract.html>
- [44] Colin Snook and Rachel Harrison. 2001. Practitioners' views on the use of formal methods: an industrial survey by structured interview. *Information and Software Technology* 43, 4 (2001), 275–283.
- [45] Marcel Steinbeck and Rainer Koschke. 2021. Javadoc violations and their evolution in open-source software. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 249–259.
- [46] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. 2007. /* icomment: Bugs or bad comments?*. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. 145–158.
- [47] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T Leavens. 2012. @ tcomment: Testing javadoc comments to detect comment-code inconsistencies. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 260–269.
- [48] Valerio Terragni, Gunel Jahangirova, Paolo Tonella, and Mauro Pezzè. 2020. Evolutionary improvement of assertion oracles. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 1178–1189. <https://doi.org/10.1145/3368089.3409758>
- [49] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2020. Unit test case generation with transformers and focal context. *arXiv preprint arXiv:2009.05617* (2020).
- [50] 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>
- [51] Arjan P. van der Meer, Rachid Kherrazi, and Marc Hamilton. 2014. Using Formal Specifications to Support Model Based Testing ASDSpec: A Tool Combining the Best of Two Techniques. In *Proceedings Ninth Workshop on Model-Based Testing, MBT 2014, Grenoble, France, 6 April 2014 (EPTCS, Vol. 141)*, Holger Schlingloff and Alexander K. Petrenko (Eds.). 1–13. <https://doi.org/10.4204/EPTCS.141.1>
- [52] Nikita Gram Voronov. 2018. deal - Design by contract for Python. <https://github.com/life4/deal>.
- [53] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.
- [54] Yi Wei, Carlo A. Furia, Nikolay Kazmin, and Bertrand Meyer. 2011. Inferring better contracts. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu, HI, USA, May 21-28, 2011*, Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic (Eds.). ACM, 191–200. <https://doi.org/10.1145/1985793.1985820>

- [55] Cheng Wen, Jialun Cao, Jie Su, Zhiwu Xu, Shengchao Qin, Mengda He, Haokun Li, Shing-Chi Cheung, and Cong Tian. 2024. Enchanting program specification synthesis by large language models using static analysis and program verification. In *International Conference on Computer Aided Verification*. Springer, 302–328.
- [56] Michael W. Whalen and Mats Per Erik Heimdahl. 1999. On the Requirements of High-Integrity Code Generation. In *4th IEEE International Symposium on High-Assurance Systems Engineering (HASE '99)*, 17-19 November 1999, Washington, D.C, USA, *Proceedings*. IEEE Computer Society, 217–226. <https://doi.org/10.1109/HASE.1999.809497>
- [57] Tingting Yu, Xiao Qu, Mithun Acharya, and Gregg Rothermel. 2013. Oracle-based Regression Test Selection. In *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013*. IEEE Computer Society, 292–301. <https://doi.org/10.1109/ICST.2013.34>
- [58] Juan Zhai, Yu Shi, Minxue Pan, Guian Zhou, Yongxiang Liu, Chunrong Fang, Shiqing Ma, Lin Tan, and Xiangyu Zhang. 2020. C2S: translating natural language comments to formal program specifications. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 25–37. <https://doi.org/10.1145/3368089.3409716>
- [59] Gehao Zhang, Zhenting Wang, and Juan Zhai. 2025. Breaking the Myth: Can Small Models Infer Postconditions Too? *CoRR* abs/2507.10182 (2025). <https://doi.org/10.48550/ARXIV.2507.10182> arXiv:2507.10182
- [60] Jiyang Zhang, Marko Ristin, Phillip Schanely, Hans Wernher Van De Venn, and Milos Gligoric. 2022. Python-by-contract dataset. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1652–1656.
- [61] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. 2011. Inferring specifications for resources from natural language API documentation. *Autom. Softw. Eng.* 18, 3-4 (2011), 227–261. <https://doi.org/10.1007/S10515-011-0082-3>

Received 11 September 2025