Improving Dynamic Specification Inference with LLM-Generated Counterexamples

Agustín Balestra*, Agustín Nolasco*¶, Facundo Molina†,
Diego Garbervetsky‡¶‖, Renzo Degiovanni§, and Nazareno Aguirre*¶‖
*University of Rio Cuarto, Rio Cuarto, Argentina, {ebalestra, nolasco, naguirre}@dc.exa.unrc.edu.ar
†Complutense University of Madrid, Madrid, Spain, facundom@ucm.es
‡University of Buenos Aires, Buenos Aires, Argentina, diegog@dc.uba.ar
§Luxembourg Institute of Science and Technology, Luxembourg, renzo.degiovanni@list.lu
¶National Council for Scientific and Technical Research (CONICET), Argentina
‖Guangdong Technion-Israel Institute of Technology, Shantou, China

***Abstract*—Contract assertions, such as preconditions, postconditions, and invariants, play a crucial role in software development, enabling applications such as program verification, test generation, and debugging. Despite their benefits, the adoption of contract assertions is limited, due to the difficulty of manually producing such assertions. Dynamic analysis-based approaches, such as Daikon, can aid in this task by inferring expressive assertions from execution traces. However, a fundamental weakness of these methods is their reliance on the thoroughness of the test suites used for dynamic analysis. When these test suites do not contain sufficiently diverse tests, the inferred assertions are often not generalizable, leading to a high rate of invalid candidates (false positives) that must be manually filtered out.**

In this paper, we explore the use of large language models (LLMs) to automatically generate tests that attempt to invalidate generated assertions. Our results show that state-of-the-art LLMs can generate effective counterexamples that help to discard up to 11.68% of invalid assertions inferred by SpecFuzzer. Moreover, when incorporating these LLM-generated counterexamples into the dynamic analysis process, we observe an improvement of up to 7% in precision of the inferred specifications, with respect to the ground-truths gathered from the evaluation benchmarks, without affecting recall.

***Index Terms*—Specification inference. Contract assertions. Runtime analysis. Large language models.**

I. Introduction

Software specifications are abstract descriptions of the intended behavior of software systems [16]. Software specifications are crucial to relate user needs to software behavior, and to perform different software analyses, such as program verification [23], test generation [7], [10], [24], and program repair [25], [30]. While formal specifications have many advantages and can be exploited for powerful analyses, they are known to be difficult to produce, in particular due to difficulties in formally expressing program intent. Thus, specifications are rarely seen in practice, accompanying software.

To address this problem, researchers have developed techniques for automatically inferring specifications from other existing software elements. At the level of source code in particular, a family of approaches starting with Daikon [13], a foundational tool in this area, generate specifications in the form of program assertions by resorting to dynamic analysis [13], [27], [29], [33]. All these approaches follow a similar approach: a mechanism is employed for generating candidate assertions for different program points (preconditions, postconditions, etc.), which are then assessed against a given test suite. Failing assertions, i.e., assertions invalidated by at least one test case, are discarded, while those assertions that are found to be valid for the provided suite are *likely specifications* (or likely invariants). Likely invariants are either directly reported back to the user as likely specifications, or are the basis of what will be reported, as some tools perform *a posteriori* assertion filtering to reduce redundancy [27], [33].

All these techniques face a common problem, inherent to this kind of analysis: the quality of the inferred assertions greatly depends on the quality of the test suite, and even for thorough and diverse test suites, these techniques typically report *false positives*, i.e., spurious or invalid assertions which hold at the corresponding program points when observed in executions of the given test suite, but are invalid in the general case [13], [27], [29], [33]. A direct approach to deal with this issue is to inspect the produced assertions, and manually identify and discard invalid ones, before using the produced assertions for downstream analyses. However, this is time consuming and error-prone, and is exacerbated as tools for specification generation grow in specification expressiveness, and thus in the kinds of specifications they are able to report.

In this paper, we assess to what extent Large Language Models (LLMs) are effective in automating specification invalidity detection. The motivation is straightforward: if LLMs are effective at detecting invalid assertions, then the burden on the engineer can be reduced, as the likely specifications to be manually examined can be significantly reduced. To analyze the effectiveness of LLMs for this task, we take a representative tool among the above-mentioned techniques, namely SpecFuzzer [27], and introduce an additional stage in the standard generate-filter-reduce pipeline of dynamic analysis specification generation techniques: after filtering assertions via dynamic analysis, we instruct an LLM to decide whether the assertion is valid (i.e., holds at the corresponding program point for every valid run of the software under analysis) or not. Since this process is likely to be inaccurate and thus lead us to discard valid assertions, we ask the LLM

to produce, when a likely invariant is deemed invalid, a test case witnessing the invalidity of the generated assertion. This approach introduces test generation *after* the assertions have been produced, thus focusing test generation on the task of invalidating produced assertions already validated by dynamic analysis. The purpose is to exploit dynamic analysis on the provided test suite to reliably discard assertions, and thus focus the task of the LLM in the more challenging assertions that dynamic analysis already "accepted". The LLM then acts as an automated intelligent oracle, that partially simulates the traditionally human task of inspecting produced assertions, identifying false positives (spurious assertions), and removing them. By requesting invalidating test cases in our queries to the LLM, we can verify the invalidity of the assertions by executing the produced test cases and confirming assertion failure. In this way, we only remove assertions that the LLM deemed invalid and could certify such invalidity via test cases.

We evaluate this LLM-enhanced technique on a combined benchmark from GAssert [33] and EvoSpex [29]. Our experiments consider three state-of-the-art LLMs from different families, namely, GPT-5.1 from Open AI, Llama 3.3 70B from Meta, and DeepSeek-R1. In order to more reliably assess the accuracy of the inferred specifications, we employ the curated ground truth statements provided in [27] for the GAssert and EvoSpex benchmarks, to implement automated ground truth checkers using SMT and SAT solvers. These checkers are used to assess accuracy in our experiments.

Our experimental results show that the counterexamples generated by GPT-5.1 help us to discard a total of 1,877 invalid specifications inferred by SpecFuzzer, while Llama 3.3 70B discards 1,048 invalid assertions, and DeepSeek-R1 discards 2,173. This LLM-based filtering reduces in 10.09%, 5.63%, and 11.68%, respectively, the noise in the inferred specifications, which otherwise would have required additional manual inspection and validation. In terms of accuracy, we observe that SpecFuzzer+GPT-5.1 obtains 74.17% precision, 54.57% recall, and 53.94% F1-score, with respect to the ground-truths gathered from the benchmarks that we implemented in automated SAT/SMT checkers, a $\sim$7% precision improvement with respect to SpecFuzzer. Moreover, Llama 3.3 70B and DeepSeek-R1 also improve SpecFuzzer's precision by $\sim$3.5% and $\sim$8%, respectively. These encouraging results suggest that LLMs have a significant potential in improving specification inference accuracy, thus reducing the engineering effort of the manual examination of generated specifications.

II. Background and Related Work

This work regards various different research areas, most notably dynamic specification inference, oracle assessment, and the use of LLMs in test generation and program analysis.

A. Dynamic specification inference

Dynamic specification inference aims at automatically producing program properties from execution scenarios. A remarkable contribution to this area, and a seminal technique and tool, is Daikon [13]. Daikon infers likely invariants by instantiating specification templates with program expressions, obtaining in this way candidate assertions, and then monitoring program executions at specific program points, such as method entrance and exit points, to assess which candidate expressions hold in all these observations at the considered program points. Daikon reports the properties that were not falsified by any test, as *likely invariants* (invariants in the sense that these were found to invariantly hold at the observed program points). Daikon is very useful and has enabled various other techniques for program analysis. However, the technique is known to have expressiveness limitations, and to have its precision subject to the thoroughness of the test suite used for inference.

Various techniques have been proposed to enhance Daikon's dynamic analysis based inference, tackling in particular expressiveness limitations and the number and quality of the reported likely invariants. GAssert [33] and EvoSpex [29] employ evolutionary approaches to generate postconditions that are validated against a given test suite. SpecFuzzer [27] provides higher flexibility and expressiveness by allowing specifications to be generated via grammar-based fuzzing from a user-provided grammar, and using dynamic analysis to validate the generated specifications in the style of Daikon and other previous tools. All these tools and techniques make attempts to improve the quality of reported specifications, e.g., employing mutation-based filtering or evolution towards stronger assertions, to reduce redundancy. Despite differences in candidate generation strategies, all these techniques fundamentally rely on the strength and thoroughness of the employed test suites for dynamic analysis, to reduce the number of invalid assertions generated. All these dynamic analysis based techniques suffer from the problem of generating false positives due to the inherently partial nature of test suites as behavior specifications.

Recently, various specification inference techniques have been proposed that directly incorporate LLMs in the specification generation process [12], [17], [22], [28], or use LLMs to generate other forms of specifications, such as test assertions or metamorphic relations [32], [38]. Unlike these techniques, our approach does not modify the invariant generation or validation mechanisms themselves. Instead, it introduces a supplementary post-processing step that actively targets invalid assertions (false positives) by generating focused counterexamples, guided by an LLM, after candidate postconditions have already been inferred.

B. Oracle Assessment

The problem of assessing and improving test oracles has been widely studied in software testing research [19], [21], [26]. OASIs [21] systematically detects these deficiencies using evolutionary search. False positives are revealed by generating test cases that violate the oracle, while false negatives are identified via mutation analysis. OASIs [21] has been used both as a quality metric and as a support tool for oracle refinement in several studies. Previous research has also focused on oracle quality, with techniques and tools such as OraclePolish [19], which employs dynamic taint analysis

to detect unused inputs and brittle assertions (assertions that depend on uncontrolled inputs and thus may be invalidated).

Our work is conceptually aligned with OASIs and OraclePolish in that we aim to generate counterexamples that expose false positives in inferred specifications. However, we differ substantially in methodology: rather than relying on evolutionary search or dynamic taint analysis, we leverage the reasoning and code generation capabilities of LLMs to directly synthesize counterexamples. Moreover, our approach is specifically tailored to dynamically inferred assertions and integrates seamlessly with existing specification inference pipelines.

C. Large Language Models for Software Testing and Analysis

Recent years have seen growing interest in leveraging LLMs for software engineering, including their application in tasks such as code generation, program repair, and test generation [18]. LLMs have been shown to be effective at various complex software engineering tasks, including generating unit tests, understanding program semantics, and reasoning about program behavior from source code [34]. In particular, various studies have explored the use of LLMs to generate test cases with the aim of improving code coverage or exposing bugs [8], [9], [14], [31], [35], [37]. Additional recent work investigates the effectiveness of LLMs for extracting software specifications from software documentation [36].

Our work differs from prior LLM-based testing and test generation methodologies in two fundamental ways. First, rather than employing LLMs for the general problem of generating test cases, we leverage them to produce tests that serve as targeted counterexamples to falsify specific inferred specifications, such as candidate program postconditions. Second, we constrain the model to output executable code that serves as a concrete witness of incorrectness, as opposed to providing unstructured natural language assessments of specification validity. This design philosophy enables the automated validation of LLM-generated responses and ensures that the resulting artifacts are executable. This implies that these responses can eventually be integrated into a dynamic loop to refine and improve an underlying specification inference process.

III. Illustrative Example

This section presents an illustrative example to demonstrate how LLMs can enhance dynamic specification inference. Specifically, we show how LLMs can identify false positives, incorrect assertions that despite being valid in the observations considered by the dynamic analysis (restricted to the test suite being considered), are invalid in the general case. As mentioned previously, the LLMs will be instructed to provide test cases that confirm the invalidity of the identified assertions.

As a simple example, consider an array-based implementation of a queue data structure, which includes a method named `getFront` (see Figure 1(a)). The method's intended behavior is to return the least recently inserted element—the item at the logical front of the queue—while preserving the queue's state. Formally, this constitutes a postcondition: for any execution of `getFront`, if the queue is not empty, the front

```java
public class QueueAr {
  private Object [] theArray;
  private int currentSize, front, back;

  /**
  * Get the least recently inserted item in the queue.
  * Does not alter the queue.
  */
  public Object getFront() {
    Object result;
    if(isEmpty()) {
      result = null;
    } else {
      result = theArray[front];
    }
    return result;
  }
  ...
}
```

(a) Target method.

```java
// Postcondition: (currentSize == front implies front < 1)
assert (this.currentSize != this.front || this.front < 1)
```

(b) Postcondition inferred by SpecFuzzer.

```java
@Test
public void testGetFront_1() throws Throwable {
    // Spec: (currentSize = front) implies (front < 1)
    QueueAr queue = new QueueAr(10);

    queue.enqueue("Item 1");
    queue.dequeue(); // This makes the queue empty again

    // Check the postcondition when the queue is empty
    assertTrue(queue.getFront() == null);

    queue.enqueue("Item 2");
    // Test that getFront() returns the correct item when
     the queue is not empty
    assertTrue(queue.getFront().equals("Item 2"));
}
```

(c) LLM-generated counterexample.

Fig. 1: Situation where an LLM-generated counterexample contradicts the inferred postcondition. Figure a) shows the target method under analysis. Figure b) shows an incorrect postcondition inferred by SpecFuzzer for this method. Figure c) shows a test case generated by an LLM that contradicts the inferred postcondition, thus showing it is a false positive.

of the queue is returned, otherwise `null` is returned. Besides, the queue's observable state after the call must be identical to its state before the call (also part of the postcondition). The implementation logic is straightforward: it first checks if the queue is empty, in which case it returns null; otherwise, it accesses and returns the element stored at the front index of the underlying array, without modifying any structural fields.

To illustrate the automated specification inference, consider the application of SpecFuzzer [27] to infer postconditions for the `getFront` method of our `QueueAr` example. Given a representative test suite, the tool generates a substantial set of 260 candidate postconditions. One such candidate, shown in Figure 1(b), asserts a relationship between internal fields: if the

queue is empty (`currentSize == front`), then `front < 1`. Although this candidate postcondition may not be considered a direct characterization of the method's intended behavior, it definitely provides relevant information regarding the internal implementation of the data structure, observable at the exit point of the method under analysis.

While the inferred postcondition is indeed informative, it is not a valid postcondition assertion for `getFront`, despite being valid for all program executions corresponding to the test suite employed for dynamic analysis. Since this is not self-evident, let us analyze the implementation of the `dequeue` method, a method that updates the `front` index, and therefore indirectly affects the possible states after executing `getFront`. As this is a circular array-based implementation of a queue, the `dequeue` operation does not reset the `front` index to 0, but rather increments it circularly:

```
public Object dequeue() {
  if(isEmpty()) return null;
  currentSize--;
  Object frontItem = theArray[front];
  theArray[front] = null;
  if (++front == theArray.length)
    front = 0;
  return frontItem;
}
```

This circular increment has a direct implication: each dequeue operation advances `front` by one (modulo capacity). Consider the following execution trace, starting from an initial empty state where `currentSize = 0` and `front = 0`:

1) Enqueue an item: `currentSize` becomes 1; `front` remains 0.
2) Dequeue the item: `currentSize` returns to 0; `front` is incremented to 1.

The queue is now logically empty (`currentSize = 0`), but `front = 1`. Now, perform another `enqueue`:

3) Enqueue a second item: `currentSize` becomes 1; `front` remains 1.

In this final state, we have `currentSize == front == 1`. This invalidates the candidate postcondition, which states that when `currentSize == front`, `front` must be less than 1, providing a concrete counterexample.

Figure 1(c) presents a test case generated by a large language model (LLM) that encodes precisely the sequence of operations described in the previous analysis. Consequently, this test case acts as a verifiable counterexample, demonstrating the invalidity of the spurious postcondition. The generation process involved providing GPT-5.1 with three key inputs: the complete source code of the `QueueAr` class, the specific implementation of the `getFront` method, and the target candidate postcondition. We then prompted the model to first evaluate the logical validity of the postcondition and, upon determining it to be false, to synthesize an executable JUnit test case that serves as a concrete counterexample. The specific prompt structure and reasoning guidelines are detailed in Section V-C.

The difficulty of automatically detecting spurious assertions and generating counterexamples highlights the potential of using large language models (LLMs) to improve dynamic specification inference. Our example demonstrates that LLMs can effectively identify and invalidate such flawed specifications. In the next section, we present an empirical study to systematically evaluate the effectiveness of LLMs in this concrete role.

IV. Research Questions

Our evaluation assesses the effectiveness of LLMs in improving dynamic specification inference. We begin by investigating whether LLM-generated tests enhance the quality of inferred specifications, posing the following research question:

RQ1 Does LLM-based counterexample generation improve dynamic specification inference?

To answer this question, we first prompt the LLM to judge the validity of a given postcondition for a target method; if deemed invalid, we ask the LLM to generate a test case as a counterexample. This counterexample is then integrated into the dynamic specification inference process to measure its impact on the accuracy relative to the benchmark's ground-truth specifications.

Since LLMs can make mistakes or produce tests irrespective of their relevance, accuracy improvement may be coincidental. Therefore, we shift our focus to directly evaluating the quality of the LLMs' outputs. Specifically, we assess their ability to *(i)* correctly identify spurious assertions from dynamic inference output, and *(ii)* generate valid counterexamples that invalidate them. This leads to the following questions:

RQ2 How effective are LLMs in identifying spurious assertions inferred by dynamic analysis?

RQ3 How effective are LLMs in generating counterexamples that invalidate the identified spurious assertions?

To address RQ2, we investigate the LLMs' ability to accurately identify spurious assertions that are invalid for the corresponding target method. For precise verification, we use constraint solving to determine the correctness of each inferred postcondition relative to the ground truth. For RQ3, we focus on the LLMs' capacity to generate valid counterexamples that demonstrate assertion invalidity. Here, we verify whether the generated counterexamples successfully lead to the removal of the spurious assertions during the dynamic filtering stage.

V. Empirical Study Workflow

A. Overview

We now provide a detailed overview of our empirical study workflow. As illustrated in Figure 2, our process evaluates the LLMs' potential to enhance specification inference. The workflow begins with a target Java class, aiming to infer its method postconditions as executable contract assertions. For initial inference, we employ SpecFuzzer [27], a state-of-the-art dynamic specification inference tool.

Our process continues with a target method and the set of likely postconditions dynamically inferred by SpecFuzzer. For each candidate postcondition, we query an LLM to perform a

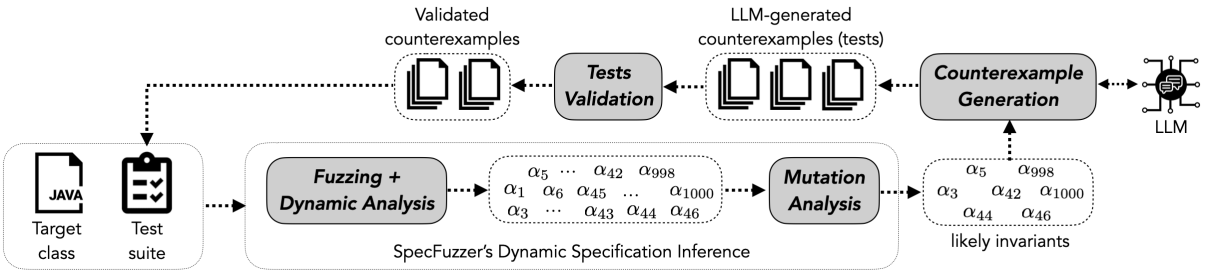

Fig. 2: Overview of our empirical study workflow.

validity analysis. Specifically, we construct a prompt containing the source code of the enclosing class, the implementation of the target method, and the postcondition under scrutiny. The LLM is instructed to first reason about the postcondition's correctness. If it judges the postcondition to be invalid, the model must then generate a concrete, executable counterexample in the form of a JUnit test case that demonstrates the violation. If the postcondition is deemed valid, the process proceeds without test generation. All LLM-generated counterexample tests are subsequently aggregated and added to the original test suite. SpecFuzzer is then re-executed using this augmented suite to produce a refined—and ideally more accurate and reduced—set of inferred postconditions. The subsequent sections detail the implementation and rationale for each of these steps.

B. Dynamic Specification Inference

We use SpecFuzzer [27] to infer postconditions for target Java methods. The tool operates in two phases. First, it employs grammar-based fuzzing to generate candidate postconditions and validates them against an existing test suite. Second, it performs a mutation-based filtering step to refine the candidates. During this latter phase, postconditions are grouped into clusters based on the mutants they kill. Finally, only a single representative postcondition is selected from each cluster, to form the final reported specification.

C. LLM-based Counterexample Generation

To identify spurious inferred assertions and generate counterexamples that discard them, we employ state-of-the-art LLMs, with strong capabilities in code understanding and generation. To interact with the LLMs, we use a carefully designed prompt template, shown in Figure 3. The prompt aims at guiding the LLM towards determining the validity of a given postcondition for a given method, and to generate a counterexample test case evidencing the violation of the postcondition, if applicable. The prompt is composed of a system prompt and a user prompt. The texts labeled with ①, ②, ③, and ④ form the system prompt. Part ① sets the role of the LLM as an expert in program verification and testing, and describes the overall task of determining the validity of a postcondition and generating counterexamples, if needed. Parts ② and ③ provide detailed instructions on the input format and the expected output format, respectively. Part ② specifies the order in which the class code, method under test, and postcondition will be provided, while part ③ outlines how the LLM should structure its output, indicating that a test case should be provided only if the postcondition is violated, in the JUnit format.

```
You are an expert in program verification and testing. You are given a Java method and a postcondition assertion for that method. Based on the method behavior, you will need to confirm that the method meets the postcondition. If indeed the method meets the postcondition, you MUST output "OK". If, on the other hand, you determine that the method does NOT meet the postcondition, you MUST output a counterexample, in the form of a jUnit test case. The counterexample test case MUST demonstrate the violation of the postcondition, using input values for the method that, after the execution of the method, make the postcondition invalid. You may provide the reasoning behind your output. ①

You will receive the input in the following way:
- The source code of the class comes first, after a line containing the text [[CODE]]
- The method under test, that the postcondition refers to, comes after the code, preceded by text [[METHOD]]
- The postcondition follows after the method, and is preceded by text [[POSTCONDITION]] (note: postconditions may include quantifiers like size(X) – returns the size of the collection X; pairwiseEqual(seq1, seq2) - True iff seq1 and seq2 have the same length, and every seq1[i] == seq2[i]; isReverse(seq1, seq2) - True iff seq1 is the reverse of seq2, typeArray(X) – returns the type of the array X; getElement(X, i) – returns the i-th element of the array X; old(X) – returns the value of X before executing the method under test; ②

The output must be produced as follows:
- After text [[VERDICT]], you will output "OK" if no counterexample for the method and postcondition exist, and "FAILED" otherwise.
- If verdict was "[[FAILED]]", output the counterexample after the verdict, preceded by text "[[TEST]]", in JUnit format. ③

Here are some examples of inputs and corresponding outputs:
[[CODE]]:
    {sample_class_1}
[[METHOD]]:
    {sample_method_1}
[[POSTCONDITION]]:
    {sample_postcondition_1}
[[VERDICT]]:
    {sample_verdict_1}
[[TEST]]:
    {sample_test_1}
... ④

[[CODE]]:
    {target_class}
[[METHOD]]:
    {target_method}
[[POSTCONDITION]]:
    {target_postcondition} ⑤
```

Fig. 3: Prompt template used for the LLM-based counterexample generation.

Part ④ provides examples of inputs and corresponding outputs, to illustrate the expected behavior of the LLM. For space reasons, we do not show here the complete examples, although they are available in our replication package [5].

Finally, part ⑤ forms the user prompt, consisting of the

actual class code, method under test, and postcondition to be analyzed by the LLM.

D. Counterexample Validation

The last step in our workflow validates the test cases generated by the LLM. Essentially, each generated test case is validated simply by compiling it. If the LLM-generated test does not compile, we instruct the LLM to fix it, with up to three attempts. If after three attempts the test case still does not compile, we discard it. At this point, we do not execute the generated test cases, as they will be executed at a later stage by SpecFuzzer.

Once we have validated the generated test cases, we incorporate all compiling test cases into the test suite used by SpecFuzzer. Then, we execute SpecFuzzer again to obtain a new set of postconditions. In this paper, we only perform one SpecFuzzer re-execution, after counterexample tests have been added, we do not iterate the process. Multiple inference iterations until convergence is a research direction we plan to explore in future work.

VI. Empirical Setup

A. Subjects

We consider a set of 43 Java methods that were part of the evaluation of SpecFuzzer [27], and have also been used in several related works on specification inference [13], [15], [29], [33]. This set includes methods from array-based implementations of stack and queue data structures, methods from popular Java libraries like Apache Commons Math, Apache Commons Lang, and Google Guava, and JTS Topology Suite, and methods from classes using or implementing data structures including linked lists, n-ary trees, and heaps. We select these subjects because the ground truth of postconditions for these methods is already available, which we use as a basis to assess the correctness of the inferred postconditions. We implemented ground truth checkers for these cases using SAT and SMT Solving. The ground truths we use are publicly available in our replication package [5].

B. Large Language Models (LLMs)

Our empirical evaluation considers three state-of-the-art LLMs from different families: GPT-5.1 [3] from Open AI, Llama 3.3 70B [4] from Meta, and DeepSeek-R1 [2]. These LLMs have shown great reasoning skills, adaptability to various tasks (Mixture-of-Experts) and good computational efficiency, making them promising candidates for our purposes.

C. Evaluation Metrics

In RQ1, we investigate whether augmenting dynamic specification inference with LLM-generated counterexamples improves the overall quality of the inferred specifications. To this end, we infer postcondition assertions for each method in our subject set using both the original SpecFuzzer tool and our extended SpecFuzzer implementation that incorporates LLM-based counterexample generation, referred to as SpecFuzzer+GPT. We compare the performance of SpecFuzzer and SpecFuzzer+GPT using standard performance metrics: precision, recall, and F1-score. These metrics are computed by comparing the inferred assertions against the ground truth specifications. While in previous studies these metrics were partially computed manually [27], in our study we fully automate this process by formally specifying the ground truth assertions. For subjects with numeric or boolean assertions, we use the Z3 SMT solver [11], available in [6]. For more complex cases involving data structure implementations, we use the Alloy Analyzer [1], the automated specification analysis tool for the Alloy specification language [20].

Given a set G of ground truth postcondition assertions for a subject method, we compute the precision of a set A of inferred assertions as follows:

$$\text{Precision} = \frac{|\{a \in A : G \models a\}|}{|A|}$$

That is, precision is the ratio of inferred assertions that are implied by the ground truth. To compute recall, we discard the set of invalid assertions $I \subseteq A$, the assertions in A that are not implied by the ground truth G, and compute recall as follows:

$$\text{Recall} = \frac{|\{g \in G : (A \setminus I) \models g\}|}{|G|}$$

Thus, recall is the ratio of ground truth assertions that are implied by the valid inferred assertions (i.e., those in $A \setminus I$). Finally, F1-score is computed as the harmonic mean of precision and recall.

In RQ2 and RQ3, we evaluate the quality and consistency of the LLM's outputs (see Figure 3 for further details on the labels below). For RQ2, we measure the accuracy of the `[[VERDICT]]` provided by the LLM. Using the correct assertions identified in RQ1 as ground truth, we report precision and recall. We consider a true positive an assertion where the `[[VERDICT]]` was `FAILED` and the dynamic analysis correctly discarded it; otherwise, it is a false positive. A true negative is an assertion where the `[[VERDICT]]` was `OK` and the dynamic analysis retained it; otherwise, it is a false negative.

In RQ3, we go one step further: for each case where the LLM's `[[VERDICT]]` was `FAILED` and it generated a `[[TEST]]`, we measure whether that counterexample indeed discards the judged invalid assertion. This allows us to assess the reasoning abilities of the LLMs and to determine whether some incorrect assertions were discarded collaterally by other tests rather than by the generated counterexamples.

D. Implementation

To run our experiments, we implemented the workflow in Figure 2 as a Python prototype on top of SpecFuzzer [27], adding an LLM-based counterexample generation stage after Specfuzzer's standard generate-filter-reduce pipeline. All scripts and data are in our replication package [5].

VII. Experimental Results

In this section we present the results obtained when running the GPT-5.1 LLM, and later in Section VIII-A we discuss the results with Llama 3.3 70B and DeepSeek-R1.

A. RQ1: Specification Inference Improvement

Table I summarizes the accuracy of the inferred postconditions, with respect to the ground-truth, when the LLM-generated test cases (counterexamples) are used or not in the dynamic specification inference process. For SpecFuzzer and SpecFuzzer+GPT, we report the number of ground truth postconditions (#GT) for each method, the number of tests used by each approach, and the performance in terms of precision, recall, and F1-score.

Precision: Notably, when incorporating the LLM-generated counterexamples, we observe a significant improvement in precision across most subjects, increasing from an average of 67.83% with SpecFuzzer to 74.17% with SpecFuzzer+GPT. More precisely, we observe that the increase in precision can be up to 54.52% for certain methods, such as `Polyupdate_sm`, with an average increase of ∼7% across all methods. In total, the LLM-generated counterexamples allow to increase precision for 48.84% of the methods (21 out of 43 methods), while remaining the same for the rest.

This increase in precision comes from the quality of the counterexamples generated by the LLM, which effectively invalidate spurious postcondition assertions that would otherwise be retained by SpecFuzzer. These are test cases that, when incorporated into the test suite and executed, lead SpecFuzzer's dynamic analysis to discard a greater number of invalid assertions. In Table II, we show, for each subject method, the number of new test cases generated by GPT-5.1, the number of inferred assertions before and after incorporating the LLM-generated tests, and the percentage of discarded assertions.

Notably, the 1,009 new counterexample test cases generated by GPT-5.1 allow us to invalidate and eliminate a total of 1,877 invalid postcondition assertions inferred by SpecFuzzer. This represents a total reduction of 10.09% of the inferred assertions, which directly contributes to the observed increase in precision. On average, GPT generated 53 new test cases per method, and discarded 13% of the inferred assertions. Some subjects experienced a significant reduction, such as `eiffel.RingBuffer` and `cozy.Polyupdate`, where 44.06% and 64.42% of the assertions were discarded, respectively.

As an example of the counterexamples that LLMs can generate, consider the method `SimpleMethods.abs`, which computes the absolute value of an integer. Figure 4(a) shows the method code along with a postcondition inferred by SpecFuzzer, stating that the return value `result` is always non-negative. While this postcondition seems correct at first glance, and holds for most integer inputs, it fails for the specific input `Integer.MIN_VALUE` (-2147483648) due to integer overflow: in this case, this method's implementation returns `Integer.MIN_VALUE`. Figure 4(b) shows a counterexample test case generated by GPT-5.1 that demonstrates this violation. When this test case is executed, it produces a negative `result` value, thereby invalidating the inferred postcondition. This example illustrates how LLMs can effectively reason about subtle edge cases in program behavior, and generate test cases that expose flaws in the specifications inferred by dynamic analysis tools like SpecFuzzer.

```java
public int abs(final int x) {
  final int i = x >>> 31;
  int result = (x^(~i + 1)) + i;
  assert (result >= 0); // Inferred postcondition
  return result;
}
```

(a) `SimpleMethods.abs` method and a postcondition inferred by SpecFuzzer.

```java
@Test
public void testAbs_NegativeReturn() throws Throwable {
  examples.SimpleMethods sm = new examples.SimpleMethods();
  int x = Integer.MIN_VALUE; // -2147483648
  int result = sm.abs(x);
}
```

(b) Counterexample test case generated by GPT-5.1, invalidating the inferred postcondition.

Fig. 4: A target method with a postcondition inferred by SpecFuzzer and a counterexample generated by GPT-5.1.

Recall: Given our designed workflow, in every method we analyze, the recall will necessarily remain the same when using SpecFuzzer alone, compared to using SpecFuzzer+GPT. This is due to the fact that SpecFuzzer+GPT can only contribute additional test cases. Thus, this can only make the dynamic analysis stronger, discarding candidate assertions that were wrongly accepted by a less thorough initial test suite. This behavior is expected, as we do not rely on the LLM to decide whether an assertion has been invalidated, but on SpecFuzzer's dynamic analysis stage, which runs each test to check assertion validity.

Notice also that SpecFuzzer+GPT's recall is limited by SpecFuzzer's specification language and inference capabilities. Thus, any valid assertion that is not produced by SpecFuzzer, will not be captured by SpecFuzzer+GPT either. Extending SpecFuzzer's recall capabilities would require proposing new candidate assertions, which is not supported by our designed workflow.

Finally, considering both precision and recall, we observe an overall improvement in F1-score, as it positively affected by the increase in precision.

RQ1 Answer. Incorporating LLM-generated counterexamples significantly improves the precision of the inferred postconditions by effectively invalidating spurious assertions. The recall remains unchanged, confirming that no valid postconditions are discarded. The overall F1-score improves as a direct result of the increased precision.

B. RQ2: Invalid Postcondition Identification Effectiveness

Table III summarizes GPT-5.1's performance in identifying invalid postconditions. We observe that GPT-5.1 correctly judged as invalid 90% of the invalid postconditions (recall), although only 55% of those judged invalid were indeed invalid (precision). Improving precision is a challenge to address in future LLM-based counterexample generation approaches.

TABLE I: Specification inference performance of SpecFuzzer and SpecFuzzer+GPT, our SpecFuzzer extension augmented with LLM-based counterexample generation using the GPT-5.1 model.

Subject	#GT	#Tests		Precision(%)		Recall(%)		F1-Score	
		SpecFuzzer	SpecFuzzer+GPT	SpecFuzzer	SpecFuzzer+GPT	SpecFuzzer	SpecFuzzer+GPT	SpecFuzzer	SpecFuzzer+GPT
SimpleMethods_addElementToSet	1	313	315	100.00	100.00	100.00	100.00	100.0	100.0
SimpleMethods_abs	3	281	351	74.70	96.88	0.00	0.00	0.0	0.0
SimpleMethods_getMin	3	277	277	100.00	100.00	66.67	66.67	80.0	80.0
SimpleMethods_incrementAt	2	343	344	96.49	96.49	0.00	0.00	0.0	0.0
StackAr_makeEmpty	3	241	251	57.81	58.62	100.00	100.00	73.26	73.91
StackAr_topAndPop	5	264	289	69.86	72.30	40.00	40.00	50.87	51.51
StackAr_pop	4	259	266	70.89	71.63	50.00	50.00	58.64	58.89
StackAr_push	4	264	280	89.19	89.19	50.00	50.00	64.07	64.07
StackAr_top	4	259	261	79.66	79.66	50.00	50.00	61.44	61.44
QueueAr_enqueue	5	241	294	80.97	83.30	80.00	80.00	80.48	81.62
QueueAr_getFront	5	241	249	76.47	74.59	80.00	80.00	78.2	77.2
QueueAr_dequeue	5	241	260	65.93	65.93	20.00	20.00	30.69	30.69
QueueAr_dequeueAll	5	241	260	80.05	79.95	80.00	80.00	80.02	79.97
QueueAr_makeEmpty	5	264	328	85.01	86.15	60.00	60.00	70.35	70.74
ArithmeticUtils_subAndCheck	1	271	275	100.00	100.00	100.00	100.00	100.0	100.0
FastMathNew_floor	1	286	348	40.54	93.75	0.00	0.00	0.0	0.0
MathUtilsNew_copySignInt	4	499	501	100.00	100.00	0.00	0.00	0.0	0.0
BooleanUtils_toBoolean	1	499	501	100.00	100.00	100.00	100.00	100.0	100.0
BooleanUtils_compare	3	273	291	82.35	82.35	100.00	100.00	90.32	90.32
IntMath_mod	1	328	360	1.64	1.67	0.00	0.00	0.0	0.0
Angle_getTurn	4	499	501	100.00	100.00	25.00	25.00	40.0	40.0
MathUtil_clamp	3	499	523	100.00	100.00	33.33	33.33	50.0	50.0
Envelope_maxExtent	4	273	275	9.02	13.30	0.00	0.00	0.0	0.0
Composite_addChild	5	299	312	21.39	22.10	0.00	0.00	0.0	0.0
DLLN_remove	1	316	318	100.00	100.00	100.00	100.00	100.0	100.0
DLLN_insertRight	4	316	322	83.87	82.35	25.00	25.00	38.52	38.36
Map_count	4	307	309	85.19	85.19	100.00	100.00	92.0	92.0
Map_remove	5	307	309	61.70	63.04	20.00	20.00	30.21	30.37
Map_extend	4	307	329	63.79	67.27	25.00	25.00	35.92	36.45
RingBuffer_extend	4	287	326	38.00	71.51	25.00	25.00	30.16	37.05
RingBuffer_item	3	287	293	44.94	65.87	66.67	66.67	53.69	66.26
RingBuffer_remove	3	287	311	39.22	59.45	100.00	100.00	56.34	74.57
RingBuffer_count	3	287	375	58.28	91.40	0.00	0.00	0.0	0.0
RingBuffer_wipeOut	3	287	302	53.25	63.58	100.00	100.00	69.5	77.74
Polyupdate_a1	2	271	489	1.52	4.46	50.00	50.00	2.95	8.2
Polyupdate_sm	1	271	315	21.09	75.61	100.00	100.00	34.83	86.11
Structure_foo	1	265	265	50.00	50.00	100.00	100.00	66.67	66.67
Structure_setX	1	265	289	80.39	85.82	100.00	100.00	89.13	92.37
ListComp02_insert_r	3	259	261	94.12	96.77	100.00	100.00	96.97	98.36
ListComp02_insert_s	3	259	259	93.75	93.75	100.00	100.00	96.77	96.77
MaxBag_remove	4	256	258	41.67	41.67	0.00	0.00	0.0	0.0
MaxBag_getMax	3	256	282	100.00	100.00	66.67	66.67	80.0	80.0
MaxBag_add	3	256	260	23.81	23.81	33.33	33.33	27.78	27.78
AVG		297.70	320.56	67.83	74.17	54.57	54.57	51.39	53.94

RQ2 Answer. GPT-5.1 demonstrates a very high capability to correctly identify invalid postconditions, achieving high precision and recall in its judgments. When it labels a postcondition as invalid, the prediction is highly reliable.

C. RQ3: Counter-example Generation Effectiveness

RQ3 concerns the effectiveness of GPT-5.1 in generating counter-examples for assertions judged as invalid. We observe that the LLM judged at least one candidate postcondition as invalid in 40 out of the 43 analyzed subjects. For each of the 40 subjects, GPT-5.1 successfully generated at least one compilable counterexample, where on average, 87.2% of the generated tests compiled correctly. In 27 out of the 40 cases, the counterexamples managed to invalidate at least one of the targeted candidate postconditions, discarding on average 47.76% of the assertions judged invalid.

It is worth remarking that some of the counterexamples produced by GPT-5.1 were very useful for identifying weaknesses in our ground-truth, and consequently improving it. For instance, the counterexample shown in Figure 4 for method `abs`, was initially considered a false positive, since we believed that the assertion `result >= 0` was correct. However, GPT-5.1 realized that for that specific implementation, the assertion does not hold for `Integer.MIN_VALUE` (actually, it is the only value for which it does not hold). In this case, the ground-truth was revised to match this behavior.

TABLE II: Counterexample test cases generated by GPT-5.1 and their impact on the invariants inferred by SpecFuzzer in terms of discarded assertions.

Subject	#M	SpecFuzzer	SpecFuzzer+GPT (one iter.)		
		#Invs	#New Tests	#Invs	Red. ⇓
SimpleMethods	4	213	79	190	10.8%
StackAr	5	1702	87	1675	1.59%
QueueAr	5	4221	156	4062	3.77%
ArithmeticUtils	1	4	4	4	0%
FastMath	1	50	62	18	64%
MathUtils	1	18	2	17	5.56%
BooleanUtils	2	60	20	59	1.67%
IntMath	1	323	32	318	1.55%
Angle	1	4	2	3	25%
MathUtil	1	19	24	18	5.26%
Envelope	1	694	2	646	6.92%
Composite	1	7623	13	7378	3.21%
DLLN	2	162	8	160	1.23%
Map	3	141	26	137	2.84%
RingBuffer	5	2281	172	1276	44.06%
Polyupdate	2	430	262	153	64.42%
Structure	2	156	24	145	7.05%
ListComp02	2	68	2	65	4.41%
MaxBag	3	436	32	404	7.34%
TOTAL	43	18,605	1,009	16,728	10.09%

TABLE III: GPT-5.1 accuracy in identifying invalid postconditions.

Subject	Precision	Recall
Angle_getTurn	100	100
BooleanUtils_toBoolean	100	100
Composite_addChild	27.27	100
Envelope_maxExtent	100	100
FastMathNew_floor	64	100
IntMath_mod	8.33	33.33
ListComp02_insert_r	100	100
Map_extend	27.27	100
MathUtil_clamp	12.5	100
MaxBag_getMax	81.25	92.86
Polyupdate_a1	85.39	95
Polyupdate_sm	91.3	91.3
QueueAr_dequeueAll	25	100
QueueAr_enqueue	62.16	85.19
QueueAr_getFront	80	100
QueueAr_makeEmpty	33.33	66.67
RingBuffer_count	94.92	96.55
RingBuffer_extend	75.76	100
RingBuffer_item	50	57.14
RingBuffer_remove	22.22	100
RingBuffer_wipeOut	55.56	62.5
SimpleMethods_abs	33.33	100
StackAr_makeEmpty	20	50
StackAr_pop	33.33	100
StackAr_push	25	100
StackAr_topAndPop	57.14	100
Structure_setX	23.08	100
AVG	55.12	90.02

RQ3 Answer. GPT-5.1 successfully generates compilable counterexample test cases for a large majority of the subjects. These counterexamples are effective at invalidating targeted, spurious postcondition assertions. In some cases, inaccuracies in the ground-truth are revealed.

VIII. Discussion

A. Generalization to other LLMs

To assess whether our findings generalize beyond the primary model under evaluation, we replicated our experiments using two additional LLMs: Llama 3.3 70B and DeepSeek-R1. Notice that the same requests were made for each model to mitigate any coincidental result. These models were selected because they belong to different model families, are open-weights, and are representative of models with state-of-the-art capabilities in problem solving and large-scale code generation.

Table IV summarizes the number of LLM-generated counterexamples, the number of invalid postconditions that were discarded (and the corresponding percentages), as well as the precision, recall and F1-score obtained after adding the corresponding tests to the suites used by SpecFuzzer's dynamic analysis. First, we observe that Llama 3.3 70B generated 430 compilable counterexamples for the subjects, while GPT-5.1 generated 982 compilable tests (128% more). Llama's counterexamples discarded a total of 1,048 invalid postconditions, a 5.63% reduction of the inferred specifications, in contrast to the 1,877 discarded by GPT-5.1 (79% more than Llama). Llama allowed us to obtain a precision of 71.13%, a recall of 51.47%, and an F1-score of 50.31%, lower than the metrics obtained with GPT-5.1. These observations, given that the same number and same requests were made with these models, suggest that GPT-5.1 is more cost-effective than Llama 3.3 70B, to generate counterexamples for invalid postconditions.

Second, in the case of the DeepSeek-R1 model, it generated 3,684 compilable counterexamples (275% more than GPT-5.1) that discarded a total of 2,173 invalid postconditions (almost 16% more than GPT-5.1). DeepSeek-R1 was more effective than GPT-5.1 in generating compilable counterexamples, but it obtains a precision of 75.62%, a recall of 51.47%, and an F1-score of 52.04%, a slightly lower performance than that obtained by GPT-5.1, suggesting that GPT-5.1 was the most cost-effective LLM for the task under study.

Our results show that open-weights LLMs obtain a performance comparable to the commercial one, being effective at identifying incorrect postconditions and generating valid counterexamples.

B. Towards 'static' LLM-based specification inference

Our experiments provided concrete evidence that modern LLMs have strong reasoning skills that allow them to statically determine whether a given postcondition is valid or not, for a given method under analysis. Then, it is rather straightforward to devise a (static) mechanism to determine the validity or invalidity of a candidate postcondition, by directly using the LLM `[[VERDICT]]` response. This may be considered an instance of the LLM-as-a-Judge strategy [39].

To further explore this possibility, we replicated the experiments with GPT-5.1, but restricted the number of tests generated by SpecFuzzer during the dynamic analysis to just 50 tests, approximately 15% of the number of tests used in

TABLE IV: Comparison between different LLMs

Model	#Tests	#Removed Assertions	Red. (% ⇓)	Precision(%)	Recall(%)	F1-score
GPT-5.1	982	1,877	10.09%	74.17%	54.57%	53.94%
Llama 3.3 70B	430	1,048	5.63%	71.13%	54.57%	50.31%
Deepseek-R1	3,684	2,173	11.68%	75.62%	54.57%	52.04%

RQ1 (see Table II). This reduces SpecFuzzer's capabilities to discard invalid postconditions, so the LLM has more invalid assertions to evaluate and discard. Our experiments show that, after analyzing 15,491 assertions, GPT-5.1 managed to eliminate 5,140 invalid postconditions. Overall, with this simple LLM-as-a-Judge approach [39], GPT-5.1 obtained a precision of 64.21%, a recall of 83.08, and an F1-score of 72.44%. These results are very promising and consistent with our previous observations, encouraging researchers to explore the direct use of LLMs for candidate specification assessment in future specification inference tools.

IX. Threats to Validity

a) External validity: A potential threat to the validity of this study concerns the specification inference tools and LLMs used for the analysis. To mitigate this risk, we relied on SpecFuzzer, one of the state-of-the-art tools for dynamic specification inference, that has been more effective than related tools like Daikon [13], GAssert [33] and EvoSpex [29]. In addition, we evaluated multiple LLMs from different model families to ensure that our findings are consistent and generalizable.

b) Internal validity: The performance of our empirical evaluation depends on the underlying tooling and LLMs. Consequently, a potential threat to internal validity arises from variations in model capability as well as from the nondeterministic nature of LLM outputs. To mitigate these threats, we evaluated multiple models from different families, reducing the risk that our findings are specific to a specific model. Additionally, to limit nondeterminism, we configured all models with a low temperature setting (0.1), aiming to make the generation process as deterministic as possible.

c) Construct validity: A potential threat to validity is data leakage, i.e., whether the LLMs have already seen our evaluation set during training. However, the risk is low for two reasons. First, the candidate specifications are generated at runtime by SpecFuzzer using fuzzing, so the specific assertions the LLMs must assess are unlikely to appear in any training corpus. While ground truth assertions exist, we use constraint solving to check semantic consistency since the syntactic form may differ substantially from any seen example. Second, the LLMs are instructed to generate novel test cases that invalidate candidate postconditions, requiring reasoning and creating tests likely not existing for the project under analysis.

X. Conclusions

In this paper, we empirically evaluated whether the test generation capabilities of LLMs can improve the specification inference process. We specifically instructed three state-of-the-art LLMs, namely GPT-5.1, Llama 3.3 70B, and DeepSeek-R1, to determine whether a candidate assertion is a valid postcondition for the method under test, and to generate a counterexample in the form of a JUnit test if it is judged invalid. These counterexamples can then be used to augment the test suites employed by SpecFuzzer during dynamic analysis, helping it to discard invalid postconditions.

Our empirical evaluation on a diverse benchmark of Java methods showed that incorporating LLM-generated counterexamples leads to a substantial improvement in specification quality. In particular, GPT-5.1 led to an average increase of approximately 7 percentage points in precision, while recall remained effectively unchanged. This result indicates that GPT-5.1 can successfully eliminate invalid assertions without discarding valid properties from the ground truth. Consequently, the overall quality of inferred specifications improved, as reflected by higher F1-scores across most subjects. We also showed that this observation generalizes to other LLMs. Llama 3.3 70B discarded 5.63% of the assertions deemed invalid, as it generated fewer compilable counterexamples than GPT-5.1. DeepSeek-R1 generated the highest number of compilable counterexamples (275% more than GPT-5.1), discarding 11.68% of invalid assertions (a 16% improvement over GPT-5.1), although at a higher computational cost.

We also observed that LLMs can effectively identify invalid postconditions, achieving 90% recall. However, precision remains at only 55%, meaning that nearly half of the assertions flagged as invalid are actually valid. This leads to wasted resources, as the LLMs attempt to generate counterexamples for correct postconditions. Thus, while LLMs show promise in detecting actual issues, further improvements are needed to reduce false positives and make the process more efficient.

Overall, this work demonstrates that LLM-based counterexample generation is a practical and effective mechanism for improving dynamic specification inference.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This work has been partially supported by Luxembourg's Ministry of Economy through RDI Law project *"Innovations for 21st Century Assessment Authoring"*, by the Luxembourg National Research Fund (FNR) PEARL program (grant agreement 16544475), by Argentina's ANPCyT through grant 2021-4862, by China's State Administration of Foreign Experts Affairs through project *"Trustworthy Evolution of LLM-generated Models"*, and by EU's Marie Sklodowska-Curie grant No. 101008233 (MISSION).

REFERENCES

[1] Alloy analyzer. https://alloytools.org/, 2026.

[2] Deepseek-r1. https://github.com/deepseek-ai/DeepSeek-R1, 2026.

[3] Gpt-5.1. https://openai.com/index/gpt-5-1/, 2026.

[4] Llama 3.3. https://www.llama.com/docs/model-cards-and-prompt-formats/llama3_3/, 2026.

[5] Replication package of our study. https://zenodo.org/records/18899070, 2026.

[6] Z3. https://github.com/Z3Prover/z3, 2026.

[7] Pablo Abad, Nazareno Aguirre, Valeria S. Bengolea, Daniel Alfredo Ciolek, Marcelo F. Frias, Juan P. Galeotti, Tom Maibaum, Mariano M. Moscato, Nicolás Rosner, and Ignacio Vissani. Improving test generation under rich contracts by tight bounds and incremental SAT solving. In *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013*, pages 21–30. IEEE Computer Society, 2013.

[8] Nadia Alshahwan, Jubin Chheda, Anastasia Finegenova, Mark Harman, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. Automated unit test improvement using Large Language Models at Meta. In *ACM International Conference on the Foundations of Software Engineering (FSE 2024)*, July 2024.

[9] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. Chatunitest: A framework for llm-based test generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, FSE 2024, page 572–576, New York, NY, USA, 2024. Association for Computing Machinery.

[10] Marcelo d'Amorim, Carlos Pacheco, Tao Xie, Darko Marinov, and Michael D. Ernst. An empirical comparison of automated generation and classification techniques for object-oriented unit testing. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), 18-22 September 2006, Tokyo, Japan*, pages 59–68. IEEE Computer Society, 2006.

[11] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[12] Madeline Endres, Sarah Fakhoury, Saikat Chakraborty, and Shuvendu K. Lahiri. Can large language models transform natural language intent into formal method postconditions? *Proc. ACM Softw. Eng.*, 1(FSE):1889–1912, 2024.

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

[14] Christopher Foster, Abhishek Gulati, Mark Harman, Inna Harper, Ke Mao, Jillian Ritchey, Hervé Robert, and Shubho Sengupta. Mutation-guided llm-based test generation at meta. In *2025 ACM Conference on Foundations of Software Engineering (FSE 2025)*. ACM, 2025. Also available as arXiv preprint arXiv:2501.12862.

[15] Aayush Garg, Renzo Degiovanni, Facundo Molina, Maxime Cordy, Nazareno Aguirre, Mike Papadakis, and Yves Le Traon. Enabling efficient assertion inference. In *34th IEEE International Symposium on Software Reliability Engineering, ISSRE 2023, Florence, Italy, October 9-12, 2023*, pages 623–634. IEEE, 2023.

[16] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 2002.

[17] Soneya Binta Hossain and Matthew Dwyer. Togll: Correct and strong test oracle generation with llms, 2024.

[18] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review. *ACM Trans. Softw. Eng. Methodol.*, September 2024. Just Accepted.

[19] Chen Huo and James Clause. Improving oracle quality by detecting brittle assertions and unused inputs in tests. In Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey, editors, *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 621–631. ACM, 2014.

[20] Daniel Jackson. Alloy: a language and tool for exploring software designs. *Commun. ACM*, 62(9):66–76, 2019.

[21] Gunel Jahangirova, David Clark, Mark Harman, and Paolo Tonella. Test oracle assessment and improvement. In Andreas Zeller and Abhik Roychoudhury, editors, *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, pages 247–258. ACM, 2016.

[22] Michael Konstantinou, Renzo Degiovanni, and Mike Papadakis. Do llms generate test oracles that capture the actual or the expected program behaviour?, 2024.

[23] Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Sci. Comput. Program.*, 55(1-3):185–208, 2005.

[24] Lisa (Ling) Liu, Bertrand Meyer, and Bernd Schoeller. Using contracts and boolean queries to improve the quality of automatic test generation. In Yuri Gurevich and Bertrand Meyer, editors, *Tests and Proofs - 1st International Conference, TAP 2007, Zurich, Switzerland, February 12-13, 2007. Revised Papers*, volume 4454 of *Lecture Notes in Computer Science*, pages 114–130. Springer, 2007.

[25] Francesco Logozzo and Thomas Ball. Modular and verified automatic program repair. In Gary T. Leavens and Matthew B. Dwyer, editors, *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*, pages 133–146. ACM, 2012.

[26] Facundo Molina, Nazareno Aguirre, and Alessandra Gorla. State field coverage: A metric for oracle quality. In *40th IEEE/ACM International Conference on Automated Software Engineering, ASE 2025, Seoul, Korea, Republic of, November 16-20, 2025*, pages 2707–2719. IEEE, 2025.

[27] Facundo Molina, Marcelo d'Amorim, and Nazareno Aguirre. Fuzzing class specifications. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 1008–1020. ACM, 2022.

[28] Facundo Molina, Alessandra Gorla, and Marcelo d'Amorim. Test oracle automation in the era of llms. *ACM Trans. Softw. Eng. Methodol.*, 34(5):150:1–150:24, 2025.

[29] Facundo Molina, Pablo Ponzio, Nazareno Aguirre, and Marcelo F. Frias. Evospex: An evolutionary algorithm for learning postconditions. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, pages 1223–1235. IEEE, 2021.

[30] Yu Pei, Carlo A. Furia, Martin Nordio, Yi Wei, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. *IEEE Trans. Software Eng.*, 40(5):427–449, 2014.

[31] Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. Code-aware prompting: A study of coverage-guided test generation in regression setting using llm. *Proc. ACM Softw. Eng.*, 1(FSE), July 2024.

[32] Seung Yeob Shin, Fabrizio Pastore, Domenico Bianculli, and Alexandra Baicoianu. Towards generating executable metamorphic relations using large language models. In Antonia Bertolino, João Pascoal Faria, Patricia Lago, and Laura Semini, editors, *Quality of Information and Communications Technology*, pages 126–141. Springer Nature Switzerland, 2024.

[33] Valerio Terragni, Gunel Jahangirova, Paolo Tonella, and Mauro Pezzè. Evolutionary improvement of assertion oracles. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 1178–1189, New York, NY, USA, 2020. Association for Computing Machinery.

[34] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. Software testing with large language models: Survey, landscape, and vision. *IEEE Trans. Softw. Eng.*, 50(4):911–936, February 2024.

[35] Zejun Wang, Kaibo Liu, Ge Li, and Zhi Jin. HITS: high-coverage llm-based unit test generation via method slicing. In Vladimir Filkov, Baishakhi Ray, and Minghui Zhou, editors, *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*, pages 1258–1268. ACM, 2024.

[36] Danning Xie, Byoungwoo Yoo, Nan Jiang, Mijung Kim, Lin Tan, Xiangyu Zhang, and Judy S. Lee. How effective are large language models in generating software specifications? In *2025 IEEE Interna-*

tional Conference on Software Analysis, Evolution and Reengineering (SANER), pages 1–12, 2025.

[37] Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. Evaluating and improving chatgpt for unit test generation. *Proc. ACM Softw. Eng.*, 1(FSE), July 2024.

[38] Yifan Zhang, Dave Towey, and Matthew Pike. Automated metamorphic-relation generation with chatgpt: An experience report. In Hossain Shahriar, Yuuichi Teranishi, Alfredo Cuzzocrea, Moushumi Sharmin, Dave Towey, A. K. M. Jahangir Alam Majumder, Hiroki Kashiwazaki, Ji-Jiang Yang, Michiharu Takemoto, Nazmus Sakib, Ryohei Banno, and Sheikh Iqbal Ahamed, editors, *47th IEEE Annual Computers, Software, and Applications Conference, COMPSAC 2023, Torino, Italy, June 26-30, 2023*, pages 1780–1785. IEEE, 2023.

[39] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging LLM-as-a-judge with MT-bench and Chatbot Arena. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, Red Hook, NY, USA, 2023. Curran Associates Inc.