METAMON: Finding Inconsistencies between Program Documentation and Behavior using Metamorphic LLM Queries

Hyeonseok Lee
School of Computing
KAIST
Daejeon, Republic of Korea
christmas@kaist.ac.kr

Gabin An
School of Computing
KAIST
Daejeon, Republic of Korea
gabin.an@kaist.ac.kr

Shin Yoo
School of Computing
KAIST
Daejeon, Republic of Korea
shin.yoo@kaist.ac.kr

***Abstract*—Code documentation can, if written precisely, help developers better understand the code they accompany. However, unlike code, code documentation cannot be automatically verified via execution, potentially leading to inconsistencies between documentation and the actual behavior. While such inconsistencies can harmful for developer's understanding of the code, checking and finding them remains a costly task due to the involvement of human engineers. This paper proposes METAMON, which uses an existing search-based test generation technique to capture the current program behavior in the form of test cases, and subsequently uses LLM-based code reasoning to identify the generated regression test oracles that are not consistent with the program specifications in the documentation. METAMON is supported in this task by metamorphic testing and self-consistency. An empirical evaluation against 9,482 pairs of code documentation and code snippets, generated using five open-source projects from Defects4J v2.0.1, shows that METAMON can classify the code-and-documentation inconsistencies with the precision of 0.72 and the recall of 0.48.**

***Index Terms*—Metamorphic Testing, Code Documentation, Large Language Model, Oracle Problem, Regression Test**

I. INTRODUCTION

Code documentation, such as comments or docstrings, is a human-readable description of the source code and its behavior. While languages like Java and Python provide documentation formats that can be processed by machines to produce structured documents (`PyDoc` and `JavaDoc`, respectively), primarily code documentation aims to aid the human understanding of the code that they accompany [1]. As such, it is critical that code documentation is consistent with the behavior of the code they accompany. If the documentation is out of sync with the program behavior, developers may understand the existing code incorrectly or imprecisely, potentially leading to buggy code changes.

Finding such inconsistencies between code documentation and program behavior is an important but challenging problem, due to the fact that documentation is written in natural languages whereas program behavior is described in programming langauges. Checking the semantic consistency across natural and programming languages has been primarily reserved only for human developers. However, the cost of manual inspection means that such checks cannot be performed frequently enough, resulting in inconsistencies in real-world projects [2]. Existing attempts to detect these inconsistencies either depend on rule-based approaches [3]–[6] or textual similarities [7]–[9] to make the task more tractable.

Recently, LLMs have shown significant capabilities to perform logical reasoning across the barrier of natural and programming languages. In addition to successfully synthesizing code from natural language specifications [10], LLMs can generate bug reproducing tests from bug reports written in natural language [11], or incorporate error messages from synthesized test cases to improve the code coverage iteratively [12]. These capabilities are directly relevant to the task of checking documentation inconsistencies, and LLMs have been evaluated for the task [13]. However, existing techniques based on LLMs tend to simply prompt LLMs to spot inconsistencies between code and documentation and falls short of actually checking the dynamic program behaviour. Given that LLMs can hallucinate [14], augmenting LLM-based inconsistency checking with a more concrete exploration of program behavior seems to be a missed opportunity.

This paper presents METAMON, an automated technique that finds inconsistencies between code documentation and program behavior via the use of a search-based test data generation technique and LLMs. To capture program behavior in more detail, instead of presenting the code verbatim in prompts, METAMON uses EvoSuite [15] to generate regression test cases. These test cases not only achieve high structural coverage (and thereby expose program behavior in more depth) but also capture the current program behavior in the form of assertions that record the program output. METAMON uses LLMs to judge whether these regression test oracles are correct or not with respect to the current documentation. To further enhance its performance, METAMON adopts a couple of prompt engineering strategies. First, METAMON uses metamorphic LLM queries: given a prompt P and an answer A, METAMON will form a subsequent query prompt P' and an expected answer A' by transforming (P, A) with Metamorphic Relations (MRs). Second, METAMON uses Chain-of-Thoughts [16] and self-consistency [17] to improve the accuracy of the responses from the LLM.

We evaluate METAMON using 9,482 pairs of Java methods and their documentation, taken from five open source projects in Defects4J [18] version 2.0.1: the dataset contains both consistent and inconsistent pairs, since we generate half of the pairs using the buggy versions. METAMON can classify the consistency between the code documentation and the program behavior captured in test cases with a precision of 0.72 and a recall of 0.48. An ablation study shows that both the metamorphic queries, and the advanced prompt engineering techniques, contribute positively to the performance.

The technical contributions of this paper are as follows:

- We present METAMON, an LLM-based technique that can check inconsistencies between documentation and program behavior. METAMON captures program behavior by generating regression test cases using EvoSuite, and uses metamorphic LLM queries and tailored prompt engineering techniques to enhance its performance.
- We perform an empirical evaluation of METAMON based on 9,482 pairs of documentation and source code extracted from five open-source projects in Defects4J v2.0.1. An ablation study shows that metamorphic LLM queries and self-consistency-based scoring all contribute to the final performance.
- We make the implementation of METAMON, along with the dataset of 9,482 pairs of code documentation and method/test code snippets, publicly available for replicability: https://figshare.com/s/dd17b119d40d4bf3176a

The rest of the paper is structured as follows: Section II provides an overview of the background and related research. Section III presents our approach, METAMON. Detailed experimental settings and research questions for evaluating METAMON are covered in Section IV, while Section V details the findings, and Section VI concludes.

II. BACKGROUND AND RELATED WORK

A. Code-and-Documentation Inconsistency

Code documentation refers to the descriptions of specifications and requirements of the program, or explanations of the source code, written with the purpose of improving code readability, facilitating easier maintenance, supporting collaboration among developers, and enhancing code reusability. However, it is common for documentation and source code to become out-of-sync over time, which can happen when only code is updated without the documentation or vice versa [2]. Analyzing the consistency between documentation and source code has traditionally been a challenging task. Some approaches have used rule-based methods to detect specific types of inconsistencies, such as issues related to locking [3], interrupt-related concurrency bugs [4], null value-related exceptions [5], and identifier renaming [6]. Other techniques have treated the inconsistency detection problem as a text similarity problem, employing machine learning models [7]–[9], which often requires a dedicated training step.

Recently, LLMs have shown remarkable capabilities in understanding both natural and programming languages, enabling them to be applied to tasks such as code generation from specifications [10], [19] or document generation from source code [19]–[21]. This suggests that checking the consistency between code and documentation is now becoming increasingly feasible. For example, a recent study shows that GPT-4 [22] can identify subtle inconsistencies between code and its documentation [13]. However, this study does not focus directly on the problem of inconsistency detection, but rather uses it as a means of assessing the code understanding capabilities of LLMs. In comparison, we propose a novel LLM-based approach that checks the consistency between program behavior (captured by regression tests) and their specifications captured in documentation, rather than directly comparing source code and documentation. We also introduce the concept of metamorphic prompting.

B. Metamorphic Testing

Metamorphic testing [23] is a testing technique that aims to reveal faults when there is no explicit test oracle. In metamorphic testing, the correctness of a program is not based on the expected output (from oracles): rather, it is based on the relationships between different inputs and their corresponding outputs, known as Metamorphic Relations (MRs). For example, in a program that calculates the square of a number, i.e., $f(x) = x^2$, the metamorphic relation could be that the square of the negative of a number is equal to the square of the number, i.e, $a = -b \rightarrow f(a) = f(b)$. Metamorphic testing has been successfully applied to machine learning models [24]–[26], which are essentially untestable [27].

In our work, we use the concept of metamorphic testing to assess the reliability of LLMs in identifying inconsistencies between program specifications and behavior. By examining the alignment of the LLM's responses with the expected MRs, we can assess the model's reliability and consistency in comprehending the underlying relationships between the program documentation and behavior. Note that this approach can also be seen as a form of LLM self-consistency, as the LLM should produce opposite outputs for inverted queries if it is truly consistent.

III. METHODOLOGY

In this paper, we introduce METAMON, a novel approach that uses LLM to automatically identify inconsistencies between program documentation and behavior. Given the program documentation for a method that meets certain quality criteria (Step A), instead of directly analyzing the method's source code, METAMON generates regression tests to capture the current semantics of the target method (Step B). To enhance the reliability of the evaluation, METAMON generates two types of prompts based on metamorphic relations: the original/transformed-version prompts that ask whether the captured program behavior in the original/transformed versions of the regression test aligns well with the program documentation (Step C). Subsequently, each type of prompt is queried to LLM multiple times, and the answers are recorded (Step D). The final set of responses from the LLM is aggregated to compute a *consistency* score, which numerically represents the extent to which the given specification in method documentation aligns with the captured method behavior (Step E). In the following sections, we provide more detailed explanations for each step of METAMON approach.

A. Selecting Method Documentation

The quantity and quality of documentation vary across projects and even down to the level of individual classes or methods, influenced by factors such as a method's complexity and its significance within the project. To ensure a fair evaluation of METAMON, we focus on documentation containing specifications that meet a set of minimum criteria.

Components in documentation essential for our analysis include descriptions of method input parameters and expected output values. This description is needed when constructing unit tests, which are divided into the test prefix/setup and the test oracle. The test prefix/setup initiates the method with appropriate inputs and drives the unit under test to an interesting state, while the test oracle specifies a condition that the resultant state should satisfy. These specifications are typically documented in Java using `@param` and `@return` tags. Thus, our selection process prioritizes methods whose specifications clearly delineate these aspects, ensuring METAMON is assessed against well-defined and actionable criteria.

B. Capturing Program behavior using Regression Tests

We employ automatically generated regression tests to produce a textual representation of the current program behavior. This textual representation serves as input for METAMON, enabling us to compare the program's captured behavior against its specifications. Regression test generation tools, such as EvoSuite [15], are typically used to generate tests that help ensure that future updates do not inadvertently disrupt the existing functionality of the program.

Fig. 1 illustrates an example of a regression test generated for the `ClassUtils.getPackageName` method in the Apache Commons Lang project, which has an incorrect oracle. This test demonstrates the consequences of an artificially introduced bug in the method's return statement. When `ClassUtils.getPackageName("line.separator")` is invoked with the current version of the program, it erroneously returns `"ine"`. This output is in conflict with the method's documented expected behavior, which is to accurately return `"line"` as the package name for the specified input.

C. Prompt Engineering based on Metamorphic Relations

METAMON uses Metamorphic Relations (MRs) to improve the reliability of the LLM responses. These MRs are grounded in two core components: *the input transformation* and *the output relation* [23]. The MRs employed in METAMON are defined as follows:

$$R = \{(a_1, a_2, Exec(t, a_1), Exec(t, a_2)) \mid a_1 = \neg a_2 \\ \rightarrow Exec(t, a_1) = \neg Exec(t, a_2)\} \quad (1)$$

TABLE I: Oracle transformations based on MR

Transformation	Description
MR_T2F	Replacing `assertTrue` to `assertFalse`
MR_F2T	Replacing `assertFalse` to `assertTrue`
MR_N2NN	Replacing `assertNull` to `assertNotNull`
MR_NN2N	Replacing `assertNotNull` to `assertNull`
MR_E2NE	Replacing `assertEquals` to `assertNotEquals`
MR_NE2E	Replacing `assertNotEquals` to `assertEquals`
MR_S2NS	Replacing `assertSame` to `assertNotSame`
MR_NS2S	Replacing `assertNotSame` to `assertSame`

where a_1 and a_2 are assertion predicates that negate each other, the function *Exec* returns the execution result (true if pass, false if fail) for test input *t* against the given assertion. Note that the execution results for *t* with a_1 should be different from that of a_2 in order to satisfy the output relation. To generate a transformed test case, we apply a transformation as described in Table I to the source test case generated at Step B. Through *input transformation*, we negate the semantics of the original assertion. Subsequently, to verify whether *the output relation* is met, we compare test outcomes to ensure that a pass in one execution directly corresponds to a fail in its counterpart, and vice versa. Notably, we filter out original assertions that do not lead to clear metamorphic relations, such as assertions that expect thrown exceptions (e.g., `assertThrows`).

In the final stage of the prompt construction, we enhance the model's problem-solving capability by incorporating the Chain-of-Thought technique [28]. This approach explicitly outlines each reasoning step, thereby improving the model's analytical processes. It facilitates systematic evaluation of the model's reasoning and ensures clarity in its responses, significantly enhancing the reliability and interpretability of outcomes. The process involves several key steps:

- **Step 1:** Identifying Method Signature
- **Step 2:** Identifying Method Description
- **Step 3:** Evaluating Test Case
- **Step 4:** Asking Confirmatory Question
- **Step 5:** Labeling Oracle

We use the term *metamorphic prompt* to refer to a pair of an original prompt and its transformed prompt. The prompt example referenced by Fig. 1 is illustrated in Fig. 2. The texts with gray background are automatically filled in by METAMON for both the original and the transformed prompts, while the sections with blue and red backgrounds represent the content for the original and transformed prompts, respectively.

D. Querying LLM

Using the generated prompt, METAMON queries the LLM to respond to the reasoning steps (Step D). We present both the original version and the transformed-version prompt to the LLM, each version n times, adopting the self-consistency prompt engineering. For the output, we instruct the LLM to label the correctness of the given test case and assertion with respect to the given documentation using three distinct types: `<correct>`, `<undecidable>` and `<incorrect>`. Given that the test oracles (representing the program's behavior) in the original version and the transformed version are opposite

```
public static String getPackageName(String className) {
    if (StringUtils.isEmpty(className)) {
        return StringUtils.EMPTY;
    }

    while (className.charAt(0) == '[') {
        className = className.substring(1);
    }

    //
    // < ... Omitted ... >
    //

    if (i == -1) {
        return StringUtils.EMPTY;
    }
    // Bug! (should be substring(0, i);)
    return className.substring(1, i);
}
```

(a) In Lang-1f, **ClassUtils.Java**-(mutated line 306)

```
/**
 * Gets the package name from a String.
 *
 * The string passed in is assumed to be a class name - it
     is not checked.
 * If the class is unpackaged, return an empty string.
 *
 * @param className the className to get the package name
    for, may be null
 * @return the package name or an empty string
 */
```

(b) Javadoc

```
public void test()  throws Throwable  {
    String string0 = ClassUtils.getPackageName("line.
     separator");
    assertEquals("ine", string0);
}
```

(c) Regression Test Case Generated by EvoSuite

Fig. 1: An example of a buggy source code along with its corresponding Javadoc and EvoSuite-generated regression test case

to each other, we would expect the responses from an LLM to differ for each prompt if the LLM is consistent and can judge the consistency between the given document and the test case.

E. Scoring based on LLM Responses

We aggregate LLM responses and convert these responses into scores. The scoring methodology differs for original and transformed prompts, and is calculated as follows:

1) Original Prompts: For original prompts, the final label obtained from the LLM's responses is converted into a numerical form using the following function:

$$f_{orig}(r) = \begin{cases} +1, & \text{if } r = \texttt{<correct>} \\ 0, & \text{if } r = \texttt{<undecidable>} \\ -1, & \text{if } r = \texttt{<incorrect>} \end{cases}$$

Suppose that we query the original-version prompt n times and the answers are $\{r_1, \cdots, r_n\}$, we aggregate the answers by taking the sum of the numerical values, $\text{score}_{orig}(\{r_1, \cdots, r_n\}) = \sum_{i=1}^{n} f_{orig}(r)$, which ranges from $-n$ to n. This score represents the degree of alignment or misalignment between the program's actual behavior and its intended specification in documentation, as perceived by the LLM. A score of n indicates that all n responses from the LLM unanimously confirmed a consistency between the program behavior and specifications. Conversely, a score of $-n$ signifies that all responses identified an inconsistency.

2) Transformed Prompts: The scoring system for transformed prompts is deliberately inverted:

$$f_{tran}(r) = \begin{cases} +1, & \text{if } r = \texttt{<incorrect>} \\ 0, & \text{if } r = \texttt{<undecidable>} \\ -1, & \text{if } r = \texttt{<correct>} \end{cases}$$

Similarly, the answers from n queries are aggregated as: $\text{score}_{tran}(\{r'_1, \cdots, r'_n\}) = \sum_{i=1}^{n} f_{tran}(r')$, which also ranges from $-n$ to n, representing the degree of alignment or misalignment between the program's behavior and its specification. For instance, a score of $-n$ indicates that the LLM consistently found a consistency between the behavior and specification of the *inversed* program across all n responses. This, in turn, indicates an inconsistency between the behavior and specification of the original program, aligning with f_{orig}.

We then aggregate scores from both the original and transformed prompts by taking the sum of both scores. When querying both types of prompts n times, the final score will range from $[-2n, 2n]$, where $2n$ represents a scenario where all n responses for the original prompt are `<correct>`, and all n responses for the transformed prompt are `<incorrect>` (or vice versa). We then normalize this score to a score ranging in $[-1, +1]$ by dividing it by $2n$.

TABLE II: Details of METAMON dataset

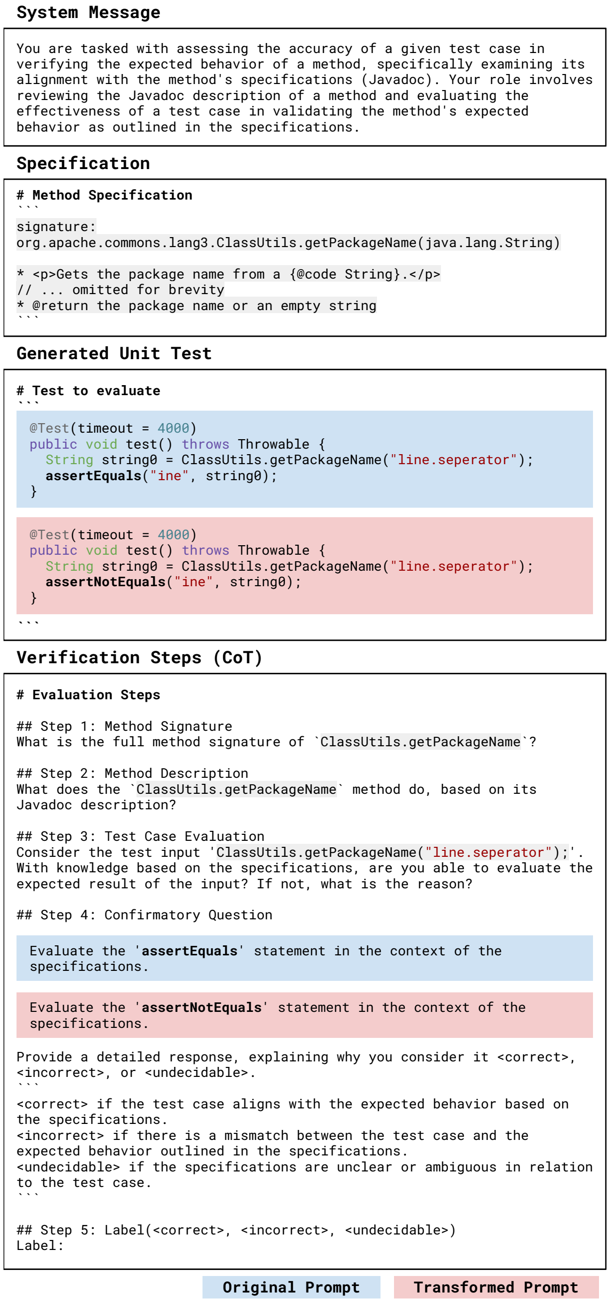
Projects	# Mutants	# Test		
		w/ incorrect oracle	w/ correct oracle	Total
Chart	11,589	2,684	2,684	5,368
Closure	343	93	93	186
Lang	4,723	594	594	1,188
Math	11,168	740	740	1,480
Time	1,983	630	630	1,260
Total	29,806	4,741	4,741	9,482

IV. EVALUATION SETUP

A. Dataset

We evaluate our approach on a carefully constructed dataset comprising $9,482$ pairs of tests and documentation. This dataset, shown in Table II, is evenly balanced, containing $4,800$ tests with incorrect oracles and an equal number of tests with correct oracles. We construct this dataset from five open-source projects included in Defects4J v2.0.1 as follows:

1) ***Documentation Quality Assessment:*** For each project, we examine the documentation quality of each method to confirm it contains descriptions for both parameters and return



System Message

~~~~
You are tasked with assessing the accuracy of a given test case in
verifying the expected behavior of a method, specifically examining its
alignment with the method's specifications (Javadoc). Your role involves
reviewing the Javadoc description of a method and evaluating the
effectiveness of a test case in validating the method's expected
behavior as outlined in the specifications.
~~~~

Specification

~~~~
# Method Specification
```
signature:
org.apache.commons.lang3.ClassUtils.getPackageName(java.lang.String)

* <p>Gets the package name from a {@code String}.</p>
// ... omitted for brevity
* @return the package name or an empty string
```
~~~~

Generated Unit Test

~~~~
# Test to evaluate
```
  @Test(timeout = 4000)
  public void test() throws Throwable {
    String string0 = ClassUtils.getPackageName("line.seperator");
    assertEquals("ine", string0);
  }

  @Test(timeout = 4000)
  public void test() throws Throwable {
    String string0 = ClassUtils.getPackageName("line.seperator");
    assertNotEquals("ine", string0);
  }
```
~~~~

Verification Steps (CoT)

~~~~
# Evaluation Steps

## Step 1: Method Signature
What is the full method signature of `ClassUtils.getPackageName`?

## Step 2: Method Description
What does the `ClassUtils.getPackageName` method do, based on its
Javadoc description?

## Step 3: Test Case Evaluation
Consider the test input 'ClassUtils.getPackageName("line.seperator");'.
With knowledge based on the specifications, are you able to evaluate the
expected result of the input? If not, what is the reason?

## Step 4: Confirmatory Question

  Evaluate the 'assertEquals' statement in the context of the
  specifications.

  Evaluate the 'assertNotEquals' statement in the context of the
  specifications.

Provide a detailed response, explaining why you consider it <correct>,
<incorrect>, or <undecidable>.
```
<correct> if the test case aligns with the expected behavior based on
the specifications.
<incorrect> if there is a mismatch between the test case and the
expected behavior outlined in the specifications.
<undecidable> if the specifications are unclear or ambiguous in relation
to the test case.
```

## Step 5: Label(<correct>, <incorrect>, <undecidable>)
Label:
~~~~

Fig. 2: An example of metamorphic prompt

conditions within the latest fixed version for each project (e.g., Chart-1f, Closure-1f). Only methods with documentation satisfying these criteria move to the next step. As illustrated in Table II, despite its large size, many methods from the Closure project were filtered out during this phase.

2) ***Mutant Injection:*** This step involves artificially introducing modifications to the program semantics. We create a set of methods injected with first-order mutants with Major [29], a mutation testing tool designed for Java programs, on methods that have passed the documentation quality assessment.

3) ***Regression Test Generation:*** Generating regression tests for every mutated method would incur a significant computational cost. Instead, through a random selection process, we choose no more than 10 mutants per method, prioritizing mutants that modify distinct lines of code to ensure diversity. However, if a method produces fewer than 10 mutants, we accept the set as is, without sampling. After selecting the mutants out of about 30,000 mutants from five projects, we employ EvoSuite to generate regression tests targeting the chosen mutated methods.

4) ***Oracle Identification:*** Automatically generated tests are executed against the latest fixed version of the program, identifying the outcomes as failing or passing tests. A failing test indicates a test with an incorrect oracle that captures the behavior modified by the mutants injected into the program. Conversely, a passing test refers to a test with a correct oracle that fails to capture the behavior modified by the mutants injected into the program. Since some mutants may be difficult to kill, the number of failing tests is significantly smaller than that of the passing ones. To ensure a balanced representation, we adjust the ratio by randomly selecting passing tests to equal the number of failing tests, facilitating a fairer comparison.

B. Experimental Settings

As a LLM model, we use `GPT-3.5-Turbo-0613` provided by OpenAI with a default parameter setting of temperature 0.7. We use EvoSuite version 1.0.7 and Major version 1.3.4 for test case and mutant generation, respectively.

C. Research Questions

We ask the following research questions in this paper.

1) ***RQ1. What is the effectiveness of* METAMON*?:*** To answer this question, we apply METAMON (with $n = 5$) to the 9,482 regression test cases shown in Table II. Our analysis focuses on assessing how well the normalized score computed from each metamorphic prompt corresponds with the ground truth. We evaluate how effectively METAMON identifies misalignment between regression oracles and documentations by examining the precision and recall against different thresholds.

2) ***RQ2. How does each component affect the performance of* METAMON*?:*** As described in Section III, METAMON employs techniques such as metamorphic relations and self-consistency to enhance the reliability of the LLM. To examine the impact of these techniques on METAMON, we conduct an ablation study. This study also evaluates the essential role of the `<undecidable>` label, which is used in cases where making a determination based solely on the provided prompt is difficult. We investigate the model's performance using only two labels, `<correct>` and `<incorrect>`, to assess the utility of the `<undecidable>` label.

3) ***RQ3. In what circumstance does* METAMON *fail to identify inconsistencies?:*** In RQ3, we conduct a qualitative analysis to identify the environments in which METAMON fails to detect inconsistencies accurately. Specifically, we examine cases where METAMON reports consistencies with high confidence, even though the test-specification pairs were, in

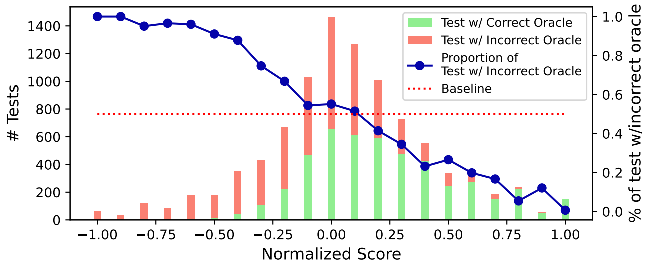


Fig. 3: Incorrect oracle detection of METAMON

TABLE III: Precision, Recall, and F1 at different thresholds

Score	Pre.	Rec.	F1	Score	Pre.	Rec.	F1
≤ -0.1	0.722	0.480	0.576	≤ -0.6	0.967	0.099	0.180
≤ -0.2	0.808	0.361	0.499	≤ -0.7	0.971	0.064	0.120
≤ -0.3	0.873	0.267	0.409	≤ -0.8	0.973	0.046	0.087
≤ -0.4	0.926	0.199	0.328	≤ -0.9	1.000	0.021	0.042
≤ -0.5	0.952	0.134	0.235	≤ -1.0	1.000	0.014	0.027

fact, inconsistent. This examination aims to uncover the causes behind these misjudgments.

V. RESULTS

A. *RQ1. Effectiveness of* METAMON

To answer RQ1, we present the results of METAMON (with $n = 5$) evaluated on the five projects listed in Table II. Fig. 3 shows the distribution of the number of metamorphic prompts across the normalized score, with the red indicating those based on tests with incorrect oracles, and the green for correct ones. The majority of evaluations result in a score of 0.0, where the METAMON either responded <undecidable> for all queries, or produced both positive and negative scores that canceled each other out. This emphasizes the difficulty in assessing the correctness of oracles from developer-written documentation that might lack sufficient detail for clear evaluation. Note that we used the documentation as they were written by developers. The blue line depicts the proportion of metamorphic prompts constructed from tests with incorrect oracles (denoted as red in the bar plot) relative to the entire set of metamorphic prompts corresponding to each score. The red dotted line represents a baseline of 0.5, reflecting the balanced nature of our dataset, which includes an equal number of tests with correct and incorrect oracles. A point close to 100% at a score of -1.0 signifies that the test-specification pairs steadily identified by METAMON to be inconsistent were indeed pairs with inconsistencies. Conversely, The point near 0% at a score of 1.0 means that the pairs confidently classified by the METAMON as aligned are very rarely associated with an incorrect oracle.

To further evaluate the performance of METAMON in detecting the test oracles inconsistent with documentation, we analyze the precision and recall at different scoring thresholds, as detailed in Table III. The oracle in a test is classified as *incorrect* if the normalized score is equal to or lower than a specified threshold. For example, when the threshold is set to -0.1, all instances with negative scores are classified as incorrect. At this threshold, the precision is 0.722 while the recall is 0.480. As the threshold is lowered, the precision increases but the recall drops. These results indicate that choosing an appropriate threshold based on user requirements can balance utility and performance.

TABLE IV: Spearman's correlation coefficient (ρ) and the p-value between scores and the ratio of incorrect oracles

	w/o <undecidable>		w/ <undecidable>	
	ρ	p-value	ρ	p-value
Metamorphic Prompt	-0.977	3.81e-13	**-0.992**	1.75e-18
Original Prompt	-0.700	1.65e-02	-0.355	2.84e-01
Transformed Prompt	-0.955	4.99e-06	-0.991	3.76e-09

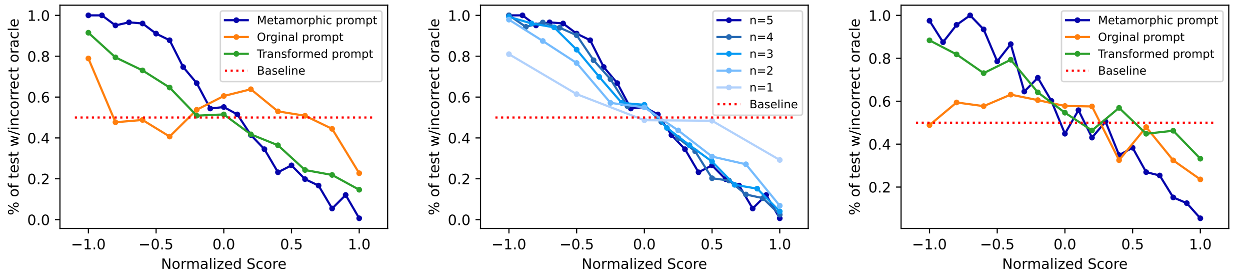
Answer to RQ1: METAMON effectively identified misalignment between documentation and test, demonstrating a high precision of 0.722 and a recall of 0.480 in detecting inconsistencies. When applying stricter thresholds, precision can be set to nearly 100%.

B. *RQ2: Ablation Study*

Fig. 4a shows the ratio of tests with incorrect oracles per their scores when METAMON is applied using only the original (orange) and the transformed (green) prompts. The blue line and red-dotted line follow the same representation as in RQ1. If metamorphic relations are not employed, i.e., assessing inconsistencies with just original prompts, the capability of METAMON to detect incorrect oracles shows a marked decrease. Specifically, when scores range between -1.0 and 0.0, the ratio drops below the baseline, leading to outcomes that are essentially indistinguishable from random alignment judgments. However, when incorporating the outcomes from the transformed prompts, as observed in RQ1, it is almost always the case that prompts receiving low scores from METAMON are strongly associated with incorrect oracles.

Fig. 4b illustrates the impact of the number of queries to the LLM on the performance of METAMON. When the number of queries, denoted as n in Section III-E, for both the original and transformed prompts increases, we observe a higher ratio of incorrect oracles within the score ranges from -1.0 to 0.0, and a lower ratio of incorrect oracles within the range of [0,1]. Additionally, the observed gain in performance grows smaller as n increases, suggesting that METAMON may be converging with respect to the number of queries.

We also explore the need for the <undecidable> label. Fig. 4c shows the results obtained using METAMON but without the <undecidable> label, i.e., the LLM is forced to label each assertion as <correct> or <incorrect>. It shows a weaker correlation between the scores and the ratio of oracles classified as incorrect at each score. This observation is supported by the Spearman's correlation coefficient values presented in Table IV. Spearman's correlation coefficient measures the strength and direction of a monotonic relationship between two variables. As can be seen in Table IV, including



(a) metamorphic (b) self-consistency (c) without `<undecidable>` label

Fig. 4: An impact of metamorphic relations, self-consistency, and labels

TABLE V: Analysis of false alarms in METAMON

	Chart	Lang	Math	Time	Total
Lack of Specification	2	0	0	0	2
Need for Contextual information	1	0	2	0	3
LLM Underperformance	0	8	7	1	16

`<undecidable>` produces a stronger negative correlation across metamorphic prompts as evidenced by Spearman's correlation coefficient values approaching -1. These findings suggest that including the `<undecidable>` label can improve the effectiveness of our approach, especially in situations where the program specifications in the documentation may not clearly define the program's semantics.

> **Answer to RQ2:** The ablation study shows that metamorphic relations, self-consistency, and the `<undecidable>` label enhance the effectiveness of METAMON.

C. *RQ3: Qualitative Analysis*

In RQ3, we analyze instances where METAMON identified tests with incorrect oracles as correct, reflected by normalized scores of 0.8 or higher, denoting high confidence. Among the 9,482 pairs analyzed, 21 were identified as exhibiting this discrepancy, and we manually inspected the reasons behind these. We categorize the causes of misclassification into three primary reasons, *Lack of Specification Detail*, *Need for Contextual Information*, and *LLM Underperformance*, whose distributions across projects are shown in Table V.

```
/*
 * Returns a string that is equivalent to the input string,
 * but with special characters converted to JavaScript
 * escape sequences. < ... Omitted ...>
 */
```

```
public void test()  throws Throwable  {
  String string0 = ImageMapUtilities.javascriptEscape("&lt;
     ");
  assertEquals("\\'lt;", string0);
}
```

Fig. 5: An example of *Lack of Specification Detail*

Lack of Specification Detail: An example of this category is in Fig. 5. Although the specification mentions that the method is supposed to convert special characters to JavaScript escape sequences, it does not provide concrete examples of special characters. This lack of detailed specification makes it challenging to assess the correctness of the test oracles.

```
/*signature: org.apache.commons.math3.geometry.euclidean.
     threed.Plane.isSimilarTo

 * Check if the instance is similar to another plane.
 * <p>Planes are considered similar if they contain the
     same
 * points. This does not mean they are equal since they can
      have
 * opposite normals.</p>
 * @param plane plane to which the instance is compared
 * @return true if the planes are similar
*/
```

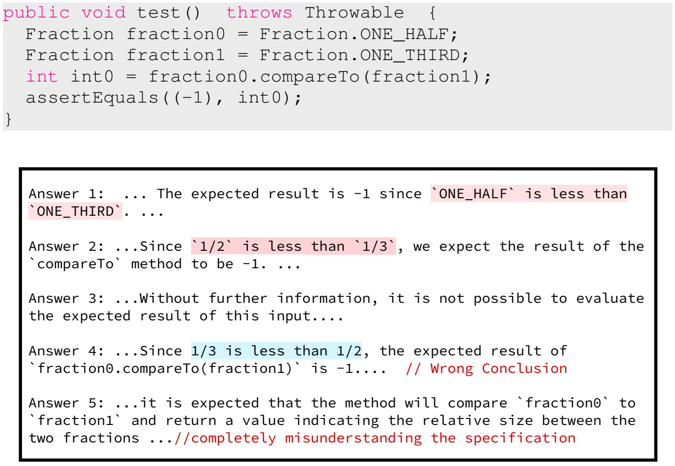
```
public void test3()  throws Throwable  {
  Vector3D vector3D0 = Vector3D.NaN;
  Plane plane0 = new Plane(vector3D0);
  boolean boolean0 = plane0.isSimilarTo(plane0);
  assertTrue(boolean0);
}
```

Fig. 6: An example of *Need for Contextual Information*

Need for Contextual Information: An example of this category is in Fig. 6 containing the specification of the `isSimilarTo` function from `Math`. As the properties that a Plan object acquires upon creation are not known, it becomes challenging to accurately evaluate the test outcomes.

LLM Underperformance: Fig. 7 shows a simple test case that the LLM fails to classify correctly due to its lack of mathematical reasoning capabilities. However, the reasoning abilities of LLMs have been improving over time. When we re-evaluated all prompts associated with this kind of case using GPT-4, the average normalized score improved to -0.156, reflecting the improved arithmetic capabilities, suggesting that such issues may further diminish as LLM performance continues to enhance.

> **Answer to RQ3:** The majority of false alarms were due to the limitations in the reasoning capabilities of the LLMs. Additionally, depending on projects, factors such as the lack of detailed specifications and the need for contextual information also contributed to these issues.



```
public void test()  throws Throwable  {
  Fraction fraction0 = Fraction.ONE_HALF;
  Fraction fraction1 = Fraction.ONE_THIRD;
  int int0 = fraction0.compareTo(fraction1);
  assertEquals((-1), int0);
}
```

```
Answer 1:  ... The expected result is -1 since `ONE_HALF` is less than
`ONE_THIRD`. ...

Answer 2: ...Since `1/2` is less than `1/3`, we expect the result of the
`compareTo` method to be -1. ...

Answer 3: ...Without further information, it is not possible to evaluate
the expected result of this input....

Answer 4: ...Since 1/3 is less than 1/2, the expected result of
`fraction0.compareTo(fraction1)` is -1....  // Wrong Conclusion

Answer 5: ...it is expected that the method will compare `fraction0` to
`fraction1` and return a value indicating the relative size between the
two fractions ...//completely misunderstanding the specification
```

Fig. 7: An example of *Underperformace of LLMs*

VI. Conclusion

This paper introduces Metamon, an LLM-based technique designed to automatically identify inconsistencies between a program's documentation and its actual behavior, as captured by regression test oracles. To address the issue of LLM hallucinations, Metamon captures program behavior via the regression test cases generated by EvoSuite and applies metamorphic prompts. Our experiments with 9,482 test-documentation pairs derived from Defects4J show that Metamon can effectively identify inconsistency with a precision of 0.72 and a recall of 0.48. We hope to expand upon these results by exploring its capabilities when used in conjunction with existing techniques such as fault localization and automatic program repair.

References

[1] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, "Which documentation for software maintenance?" *Journal of the Brazilian Computer Society*, vol. 12, pp. 31–44, 2006.

[2] Z. M. Jiang and A. E. Hassan, "Examining the evolution of code comments in postgresql," in *Proceedings of the 2006 International Workshop on Mining Software Repositories*, ser. MSR '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 179–180. [Online]. Available: https://doi.org/10.1145/1137983.1138030

[3] L. Tan, D. Yuan, G. Krishna, and Y. Zhou, "/* icomment: Bugs or bad comments?*," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, 2007, pp. 145–158.

[4] L. Tan, Y. Zhou, and Y. Padioleau, "acomment: mining annotations from comments and code to detect interrupt related concurrency bugs," in *Proceedings of the 33rd international conference on software engineering*, 2011, pp. 11–20.

[5] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens, "@ tcomment: Testing javadoc comments to detect comment-code inconsistencies," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 2012, pp. 260–269.

[6] I. K. Ratol and M. P. Robillard, "Detecting fragile comments," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 112–122.

[7] A. Corazza, V. Maggio, and G. Scanniello, "Coherence of comments and method implementations: a dataset and an empirical investigation," *Software Quality Journal*, vol. 26, pp. 751–777, 2018.

[8] S. Panthaplackel, J. J. Li, M. Gligoric, and R. J. Mooney, "Deep just-in-time inconsistency detection between comments and source code," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 1, 2021, pp. 427–435.

[9] F. Rabbi and M. S. Siddik, "Detecting code comment inconsistency using siamese recurrent network," in *Proceedings of the 28th International Conference on Program Comprehension*, 2020, pp. 371–375.

[10] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago *et al.*, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[11] S. Kang, J. Yoon, and S. Yoo, "Large language models are few-shot testers: Exploring llm-based general bug reproduction," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2312–2323.

[12] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "Adaptive test generation using a large language model," *arXiv e-prints*, pp. arXiv–2302, 2023.

[13] Z. Li and D. Shin, "Mutation-based consistency testing for evaluating the code understanding capability of llms," *arXiv preprint arXiv:2401.05940*, 2024.

[14] L. Huang, W. Yu, W. Ma, W. Zhong, Z. Feng, H. Wang, Q. Chen, W. Peng, X. Feng, B. Qin *et al.*, "A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions," *arXiv preprint arXiv:2311.05232*, 2023.

[15] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 416–419. [Online]. Available: https://doi.org/10.1145/2025113.2025179

[16] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

[17] X. Wang, J. Wei, D. Schuurmans, Q. Le, E. Chi, S. Narang, A. Chowdhery, and D. Zhou, "Self-consistency improves chain of thought reasoning in language models," *arXiv preprint arXiv:2203.11171*, 2022.

[18] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 international symposium on software testing and analysis*, 2014, pp. 437–440.

[19] J. Shin, C. Tang, T. Mohati, M. Nayebi, S. Wang, and H. Hemmati, "Prompt engineering or fine tuning: An empirical assessment of large language models in automated software engineering tasks," *arXiv preprint arXiv:2310.10508*, 2023.

[20] W. Sun, C. Fang, Y. You, Y. Miao, Y. Liu, Y. Li, G. Deng, S. Huang, Y. Chen, Q. Zhang *et al.*, "Automatic code summarization via chatgpt: How far are we?" *arXiv preprint arXiv:2305.12865*, 2023.

[21] T. Ahmed, K. S. Pai, P. Devanbu, and E. T. Barr, "Improving few-shot prompts with relevant static analysis products," *arXiv preprint arXiv:2304.06815*, 2023.

[22] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[23] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés, "A survey on metamorphic testing," *IEEE Transactions on software engineering*, vol. 42, no. 9, pp. 805–824, 2016.

[24] L. Applis, A. Panichella, and A. van Deursen, "Assessing robustness of ml-based program analysis tools using metamorphic program transformations," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 1377–1381.

[25] C. Murphy, G. E. Kaiser, and L. Hu, "Properties of machine learning applications for use in metamorphic testing," 2008.

[26] X. Xie, J. W. Ho, C. Murphy, G. Kaiser, B. Xu, and T. Y. Chen, "Testing and validating machine learning classifiers by metamorphic testing," *Journal of Systems and Software*, vol. 84, no. 4, pp. 544–558, 2011.

[27] E. J. Weyuker, "On Testing Non-Testable Programs," *The Computer Journal*, vol. 25, no. 4, pp. 465–470, 11 1982.

[28] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," in *Proceedings of the 36th International Conference on Neural Information Processing Systems*, ser. NIPS '22. Red Hook, NY, USA: Curran Associates Inc., 2024.

[29] R. Just, F. Schweiggert, and G. M. Kapfhammer, "Major: An efficient and extensible tool for mutation analysis in a java compiler," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 2011, pp. 612–615.