From Implemented to Expected Behaviors: Leveraging Regression Oracles for Non-regression Fault Detection using LLMs

Stefano Ruberto
JRC European Commission
Ispra, Italy
stefano.ruberto@ec.europa.eu

Judith Perera
University of Auckland
Auckland, New Zealand
jper120@aucklanduni.ac.nz

Gunel Jahangirova
King's College London
London, United Kingdom
gunel.jahangirova@kcl.ac.uk

Valerio Terragni
University of Auckland
Auckland, New Zealand
v.terragni@auckland.ac.nz

***Abstract*—Automated test generation tools often produce assertions that reflect implemented behavior, limiting their usage to regression testing. In this paper, we propose LLMPROPHET, a black-box approach that applies Few-Shot Learning with LLMs, using automatically generated regression tests as context to identify non-regression faults without relying on source code. By employing iterative cross-validation and a leave-one-out strategy, LLMPROPHET identifies regression assertions that are misaligned with expected behaviors. We outline LLMPROPHET's workflow, feasibility, and preliminary findings, demonstrating its potential for LLM-driven fault detection.**

***Index Terms*—Regression Oracles, Non-Regression Faults, Few-Shot Learning, Large Language Models, Fault Detection**

I. INTRODUCTION

Recent advances in software testing have significantly improved test input generation, leaving the oracle problem as the primary obstacle to full test automation [1]. Existing state-of-the-art tools such as EVOSUITE [2] and RANDOOP [3] generate test case assertions that capture the implemented behavior, which is useful for detecting regression faults in future software versions. However, to uncover faults in the current software version, these assertions must align with the expected behavior—often known only to human developers.

The advancements in applying Large Language Models (LLMs) to software engineering tasks [4] have sparked interest in their potential for generating test cases [5], [6] and, more specifically, test oracles that capture expected behavior [7]. The study by Konstantinou et al. [7] evaluates the effectiveness of LLMs in two key areas: *oracle classification*, which involves determining whether a given assertion aligns with the expected behavior, and *oracle generation*, which focuses on creating new assertions that accurately represent the expected behavior. Their findings reveal that LLMs frequently produce oracles that reflect the program's implemented behavior rather than its expected behavior and are not successful in accurately identifying which oracles align with the expected behavior.

This paper presents LLMPROPHET, a novel approach to verify whether a regression assertion, which aligns with the implemented behavior, also aligns with the expected behavior. Unlike Konstantinou et al. [7], LLMPROPHET avoids prompting source code to LLMs directly, as this could introduce biases into the LLM's reasoning process. Instead, it uses regression tests generated by existing tools to train an LLM on the behavior of the class under test (CUT) through a Few-Shot Learning strategy. By employing iterative cross-validation and a leave-one-out technique – and by leveraging the LLM's extensive internal knowledge – LLMPROPHET identifies regression assertions that do not capture the expected behavior, thereby revealing non-regression faults. Our preliminary findings suggest that this approach is promising and deserves further investigation.

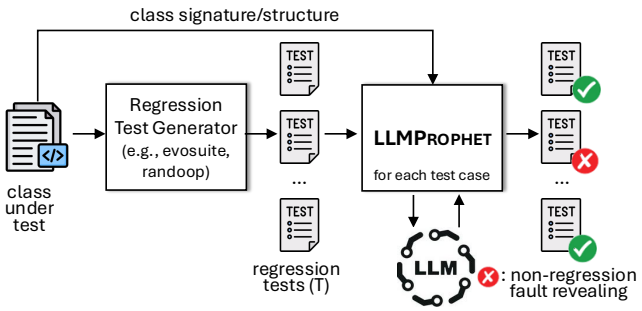

Fig. 1: Logical architecture of LLMPROPHET

II. LLMPROPHET

Figure 1 shows the logical architecture of our envisioned approach. LLMPROPHET takes as input a regression test suite generated by tools such as EVOSUITE [2], RANDOOP [3], or PYNGUIN [8] represented as $\mathcal{T} = \langle t_1, t_2, \ldots, t_n \rangle$, along with the class signatures of the CUT and any relevant classes (i.e., CUT direct dependencies). It then automatically evaluates each test case, marking it as non-regression fault-revealing or not.

Internally, LLMPROPHET leverages an LLM by querying it n times – once for each test case in $\mathcal{T}$. For each test case t_i, the LLM evaluates whether the assertion in the test aligns with the expected behavior or merely aligns with the current (potentially faulty) implementation. In essence, while the regression test case under analysis pass on the current implementation (by design), LLMPROPHET aims to determine whether it would also pass on a correct implementation of the CUT.

During each query, the remaining $n-1$ test cases are provided as contextual examples for the LLM to use as reference

(representing possible input/output pairs). More formally, the approach follows a **cross-validation process**: it selects $n-1$ test cases as context for the model, applies Few-Shot Learning with an LLM to approximate the behavior of the class under test, and evaluates the remaining case t_i using a leave-one-out strategy.

This process is repeated iteratively for all test cases, ensuring that each test is cross-validated. Note that a test case may contain multiple assertions, in which case LLMPROPHET evaluates each assertion individually. Since the entire process is **fully automated**, LLMPROPHET provides an efficient and novel way to identify potential non-regression faults in the current version of the program, relying on automatically generated regression tests.

A potential issue is that if the CUT is faulty, it is likely that multiple tests would contain regression oracles that predicate on incorrect program behaviors. This will result in the context being polluted with noisy input/output pairs that do not align with the expected (correct) behavior of the CUT. However, our preliminary results reveales that this is not a critical issue. In fact, we observed that LLMPROPHET, when specifically prompted, recognizes and ignores such noisy test cases. From our preliminary analysis we conjecture that this is because an LLM deduces the expected behavior of the CUT not only from the given examples, but (especially) also from contextual information (such as class, method, and parameter names, as well as JAVADOC/comments) and its vast internal knowledge. This will be further discussed in the next sections.

III. RUNNING EXAMPLE

We now explain how LLMPROPHET works through a running example. Listing 2 presents the prompt to determine whether a regression test exposes non-regression faults.

The example involves a JAVA flight booking application. The `FlightBooking` class is initialized with a customer and implements a method, `findBestFlight`, which is responsible for finding the best flight. The input parameter for this method is an instance of the `PreferenceDate` class. This class specifies a preferred flight date and a range, defined in days, within which the application is allowed to propose alternative dates with both backward and forward flexibility. The method under test, `findBestFlight`, is expected to return a date within the specified range based on an arbitrarily complex internal logic.

Test case under analysis: Let us assume that LLMPROPHET has invoked a regression test suite generator (e.g., EVOSUITE or RANDOOP) that generated six test cases that predicate on the return value of `findBestFlight`. For instance, consider the following automatically generated JUNIT test case:

```
Customer customer06 = new Customer(24680,"Diana","Clark");
FlightBooking booking = new FlightBooking(customer06);
Date date01 = DateFormat.parse("20/05/2022");
PreferenceDate preferenceDate01 = new ↩
     PreferenceDate(date01,4);
Date date02 = booking.findBestFlight(preferenceDate01);
assertEquals(DateFormat.parse("15/05/2022"),date02);
```

Listing 1: regression test that predicates on an incorrect return value of `findBestFlight`.

This test case initializes a flight booking instance for a customer, sets a preferred flight date (`"20/05/2022"`) with a flexibility of four days, and checks if the returned date is `"15/05/2022"`. However, the assertion was automatically generated by executing the code and assuming the values returned by `findBestFlight` are correct (because it is a regression test). While the assertion passes (i.e., it returns `true`) on the current implementation, the test case triggers a fault because `"15/05/2022"` lies outside the range specified by `PreferenceDate` (i.e, from `"16/05/2022"` to `"24/05/2022"` inclusive). As such, in a correct implementation of `findBestFlight`, the test should fail.

GPT-4O, GPT-4O-MINI, GPT-O1, GPT-O1-MINI, and LLAMA 3.3 70B can all correctly identify that this test reveals a non-regression fault using only the prompt shown in Listing 2.

Class Description:
As an expert Java tester, you are given the following abstract Java classes:

```
public class PreferenceDate {
   public Date date;
   public int flexibilityDays;
}

public class FlightBooking {
   public void FlightBooking(Customer customer);
   public Date findBestFlight(PreferenceDate date);
}
```

Example Test Cases:

1. Learn from this example:

```
Customer customer01 = new Customer(34567, "John", "Smith");
FlightBooking booking = new FlightBooking(customer01);
Date date01 = DateFormat.parse("10/12/2024");
PreferenceDate preferenceDate01 = new ↩
     PreferenceDate(date01, 3);
Date date02 = findBestFlight(preferenceDate01);
assertEquals(DateFormat.parse("11/12/2024"), date02); // ↩
     <outcome>true</outcome>
```

2. Learn from this example:

```
Customer customer01 = new Customer(12345, "Alice", ↩
     "Brown");
FlightBooking booking = new FlightBooking(customer01);
Date date01 = DateFormat.parse("16/08/2022");
PreferenceDate preferenceDate01 = new ↩
     PreferenceDate(date01, 3);
Date date02 = booking.findBestFlight(preferenceDate01);
assertEquals(DateFormat.parse("15/08/2022"), date02); // ↩
     <outcome>true</outcome>
```

3. Learn from this example:

```
Customer customer03 = new Customer(98765, "Eve", "Adams");
FlightBooking booking = new FlightBooking(customer03);
Date date01 = DateFormat.parse("05/09/2023");
PreferenceDate preferenceDate01 = new ↩
     PreferenceDate(date01, 3);
Date date02 = booking.findBestFlight(preferenceDate01);
assertEquals(DateFormat.parse("10/09/2023"), date02); // ↩
     <outcome>true</outcome>
```

4. Learn from this example:

```
Customer customer04 = new Customer(67890, "Bob", "Smith");
FlightBooking booking = new FlightBooking(customer04);
Date date01 = DateFormat.parse("01/01/2024");
PreferenceDate preferenceDate01 = new ↩
     PreferenceDate(date01, 5);
Date date02 = booking.findBestFlight(preferenceDate01);
assertEquals(DateFormat.parse("01/01/2024"), date02); // ↩
     <outcome>true</outcome>
```

5. Learn from this example:

```
Customer customer05 = new Customer(13579, "Charlie", ↩
     "Johnson");
FlightBooking booking = new FlightBooking(customer05);
Date date01 = DateFormat.parse("10/10/2023");
```

```
    PreferenceDate preferenceDate01 = new ↩
        PreferenceDate(date01, 2);
    Date date02 = booking.findBestFlight(preferenceDate01);
    assertEquals(DateFormat.parse("12/10/2023"), date02); // ↩
        <outcome>true</outcome>

Instructions:
Some of the above assertions may be incorrect as they
capture the implemented behavior, not the expected
behavior. Exclude the incorrect examples from your
reasoning.
Now, consider this new test case:

    Customer customer06 = new Customer(24680, "Diana", ↩
        "Clark");
    FlightBooking booking = new FlightBooking(customer06);
    Date date01 = DateFormat.parse("20/05/2022");
    PreferenceDate preferenceDate01 = new ↩
        PreferenceDate(date01, 4);
    Date date02 = booking.findBestFlight(preferenceDate01);

Evaluate the following assertion step by step, assuming a
correct implementation of the classes:

    assertEquals(DateFormat.parse("15/05/2022"),date02);

Question:
Is this a plausible assertion? In other words, should the
outcome of this assertion be true or false?

Reason step by step and then answer with a single tag:
<outcome>true</outcome>, or
<outcome>false</outcome>
```

Listing 2. Example of LLMPROPHET's prompt to detect non-regression faults from automatically generated regression tests

The prompt consists of four main components: 1) *Class Description:* the signatures of the class under test and relevant classes (without the implementation) 2) *Example Test Cases:* a set of test cases used by the LLM to infer the expected logic behind the behavior of the method under test; 3) *Instructions:* the description of the task, and 4) *Question*: the specific query that the LLM is expected to answer.

1) Class Description: In our approach, the LLM derives its expectations about the behavior of the CUT entirely from its internal knowledge and the provided class signatures. We intentionally exclude the actual class implementations from the input. This decision ensures that the LLM infers the correct expected behavior, rather than mimicking the current (potentially faulty) implementation. By not providing the source code, we avoid introducing bias that could interfere with the LLM's ability to assess correctness. Instead of virtually executing the code, the LLM uses class names, parameter names, and other textual descriptions as cues to retrieve its understanding of the expected behavior. In this way, the LLM functions as a black-box reasoning engine, identifying faults by detecting deviations from these inferred expectations.

In this example, the input includes the method signatures of the `FlightBooking` and `PreferenceDate` classes. Although absent from this example, adding JAVADOC comments could further improve the LLM's performance by providing additional semantic clues.

2) Example Test Cases: The second component of the prompt consists of the remaining $n-1$ tests (five in our example). These tests are used in a few-shot learning scenario to help the LLM learn the expected behavior of the CUT by providing input/output pairs. For example, the first test case has input `PreferenceDate("10/12/2024", 3)` and output `"11/12/2024"`. Note that all tests pass because they capture the implemented behavior, as indicated by the comment `<outcome>true</outcome>`.

3) Instructions: The third component of the prompt describes the task by providing the test case under analysis. As mentioned earlier, LLMPROPHET assumes that errors may exist in the regression oracles of the example test cases, such as the third example being incorrect (`"10/09/2023"` falls outside the acceptable range). Therefore, the prompt informs the LLM that some examples might be erroneous and should be ignored. This approach introduces flexibility and helps prevent the LLM from overfitting to the examples, a well-known issue in AI, particularly with small datasets. Since the input does not include the implementation and provides only sparse examples, we expect the evaluation to also rely on the LLM's vast memory-derived knowledge, adapted from similar cases, and the semantic clues given by the class signatures.

4) Question: The final component asks if the assertion of the input test case (Listing 1) should pass or fail in a correct implementation of the CUT. The prompt requires the outcome of the assertion evaluation as an `<outcome>true/false</outcome>` tag to facilitate the automated parsing of the results. Since LLMs often perform better with *Chain-of-Thought* reasoning, we explicitly request this with the keyword "`step by step`". This results in a detailed output, but the final evaluation can be easily extracted by parsing the `<outcome>` tag.

The prompt can be easily parameterized to evaluate different test cases and classes in various scenarios while preserving its structure. The principles outlined here are general, and we expect this prompt to be applicable to different software and LLMs. We have tested this prompt on the previously mentioned five LLMs, confirming the validity of the design principles, as it always correctly detects the test cases from Listing 1 as non-regression fault-revealing (i.e., answering `<outcome>false</outcome>`). This is despite the fact that Example 3 is incorrect.

IV. LLMPROPHET'S SENSITIVITY TO THE EXAMPLES

We conducted a small-scale experiment to assess LLMPROPHET's tolerance to variations in both the quantity and correctness of input examples. We generated two sets of ten test cases for the `FlightBooking` class: one with "faulty assertions" (e.g., the one in Listing 1) that reflect incorrect behavior, and another with "correct assertions" (e.g., the first test case in Listing 2) aligned with the expected behavior.

Next, we created samples by combining different numbers of faulty assertions (0, 1, 5, 10) and examples (0, 1, 5, 10), as shown in Table I, where zero examples constitutes a zero-shot scenario. For instance, in our running example (Listing 2), we provided five test cases as context for the model, one of which contained a faulty assertion. We randomly selected correct and faulty cases from our two sets of test examples and used the same prompt format as in Listing 2. Each combination was evaluated twice using always the same test case under analysis of our running example, once

TABLE I: Results with GPT-4o using the non-regression fault-revealing assertion of our example and a correct assertion, while varying the number of provided and incorrect examples. ✓ and ✗ indicate a correct and incorrect classification, respectively.

assertion to analyse for the test in Listing 1	# incorrect examples	# examples 0	1	5	10
non-regression fault revealing `assertEquals("15/05/2022"),.);`	0	✓	✓	✓	✓
	1		✓	✓	✓
	5			✓	✓
	10				✓
correct `assertEquals("19/05/2022"),.);`	0	✓	✓	✓	✓
	1		✓	✓	✓
	5			✗	✓
	10				✓

with a faulty assertion (expecting `"15/05/2022"`) and once with a correct assertion (expecting `"19/05/2022"`). This allows us to evaluate both scenarios in which LLMPROPHET should produce `<outcome>false</outcome>` and `<outcome>true</outcome>`. We used GPT-4o with temperature 0.0 and top-p 1.0. We released all the prompts used in this experiment [9].

Table I shows promising results: the LLM always correctly identifies the faulty assertion by outputting `<outcome>false</outcome>`. However, it fails once on the correct assertion when the prompt includes five test cases, all of which are faulty (outputting false instead of true). This suggests the model can be biased toward incorrect behavior when all examples in the prompt are wrong. Surprisingly, the model recovers when presented with ten faulty examples and zero correct ones, suggesting that, in most cases, our prompt effectively instructs the LLM to ignore erroneous input/output pairs. We conjecture that under these conditions, the LLM relies on its internal knowledge to infer the correct behavior.

V. Discussion

Our preliminary results demonstrate the LLM's ability to retrieve an appropriate model from its **internal knowledge** to infer expected software behaviors. Indeed, the class signatures in the `FlightBooking` example provide minimal information about the expected method behaviors. However, the LLM's exposure to similar software systems and tests during training allows it to interpolate or adapt to the current scenario. General background knowledge, even non-technical, can also be crucial. For example, knowing the traditional process for booking a flight helps to correctly infer the expected behavior of the CUT.

Since the training set of LLMs can be extremely vast, we expect most software behaviors to be well covered. However, there may be exceptions where LLMPROPHET will be less effective, such as when the functionality under test is too specific or unusual, or when variable names and textual information are insufficient or unrelated to the CUT's behavior. In such complex cases, we conjecture that, unlike the simple `FlightBooking` example, the test cases will play a more significant role in helping the LLM infer the CUT's semantics.

VI. Conclusion and Future Work

This paper presented a novel approach for automated testing that uses automatically generated regression tests to detect non-regression faults. To the best of our knowledge, *this is a novel way of leveraging the reasoning power and knowledge base of LLMs for fault detection*. This new idea paper sparks interesting future work.

Prototype tool implementation: We plan to develop a prototype tool that fully automates every step of LLMPROPHET: test generation, prompt creation, and result reporting. Since our approach is language-independent, it would be interesting to evaluate LLMPROPHET's effectiveness on different programming languages. For instance, JAVA and PYTHON have well-established regression test generation tools (i.e., EVOSUITE [2], RANDOOP [3], PYNGUIN [8]).

Comprehensive evaluation: We need a comprehensive evaluation involving several CUTs of varying complexity to understand which semantic sources of information (test cases, class signatures, comments, or the LLM's internal knowledge) contribute most to LLMPROPHET's effectiveness. Although test cases had minimal impact in our simple example, they might become critical for more complex CUTs with less straightforward input/output relationships. An ablation study would help shed light on this. Moreover, a direct comparison with the approach of Konstantinou et al. [7] is necessary to confirm that a white-box approach that provides implementation details to the LLM can bias it toward implemented rather than expected behaviors.

Sustainability considerations: While recent LLMs achieve remarkable performance, they are also CPU- and energy-intensive. To reduce computational and energy consumption, we can train specialized, smaller LLMs. We have already run experiments on smaller, faster models like LLaMA 3.3 70B, and further optimizations are likely possible. Additionally, test generators like EVOSUITE can produce large test suites, so approaches for selecting a representative subset of tests could lower the overall cost of LLMPROPHET.

References

[1] V. Terragni, G. Jahangirova, P. Tonella, and M. Pezzè, "Evolutionary improvement of assertion oracles," in *ESEC/FSE*, 2020, p. 1178–1189.

[2] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *ESEC/FSE*, 2011, pp. 416–419.

[3] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for java," in *OOPSLA*, 2007, p. 815–816.

[4] V. Terragni, A. Vella, P. Roop, and K. Blincoe, "The future of ai-driven software engineering," *ACM Trans. Softw. Eng. Methodol.*, Jan. 2025.

[5] W. C. Ouedraogo, K. Kabore, H. Tian, Y. Song, A. Koyuncu, J. Klein, D. Lo, and T. F. Bissyande, "Llms and prompting for unit test generation: A large-scale evaluation," in *ASE*, 2024, pp. 2464–2465.

[6] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," *IEEE Transactions on Software Engineering*, 2023.

[7] M. Konstantinou, R. Degiovanni, and M. Papadakis, "Do llms generate test oracles that capture the actual or the expected program behaviour?" in *arXiv 2410.21136*, 2024.

[8] S. Lukasczyk and G. Fraser, "Pynguin: Automated unit test generation for python," in *ICSE*, 2022, pp. 168–172.

[9] S. Ruberto, J. Perera, G. Jahangirova, and V. Terragni, "LLMPROPHET experimental data," 2025, https://doi.org/10.5281/zenodo.14889625.