Leveraging E2E Test Context for LLM-Enhanced Test Data and Descriptions

Enhancing Automated Software Testing with Runtime Data Integration

Mattheo de Wit[1]

Supervisor(s): Andy Zaidman[1]**, Amir Deljouyi**[1]

[1]**EEMCS, Delft University of Technology, The Netherlands**

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 23, 2024

Name of the student: Mattheo de Wit
Final project course: CSE3000 Research Project
Thesis committee: Andy Zaidman, Amir Deljouyi, Asterios Katsifodimos

Abstract

Automated software testing plays a critical role in improving software quality and reducing manual testing expenses. However, generating understandable and meaningful unit tests remains challenging, especially with frameworks optimized for coverage like Search-Based Software Testing (SBST). Large Language Models (LLMs) have the capability to generate human-like text, while capture/replay techniques can provide realistic data scenarios through trace logs, contributing to meaningful test case generation. This study introduces *UTGen+*, an approach that enhances LLM-based SBST by integrating trace logs from end-to-end tests, aiming to further improve test case understandability. We conducted a comparative user study with 9 participants using UTGen+, original UTGen, and conventional SBST (EvoSuite), focusing on the effects of trace log inclusion on the naturalness and relevancy of comments, identifiers, and test data across several projects. The results indicated that while UTGen+ did not improve the naturalness and relevancy of comments and identifiers, it significantly enhanced the relevancy of test data. These findings suggest that incorporating contextual data can indeed benefit the generation of more relevant and understandable automated test cases.

1 Introduction

Unit testing is an essential component in software development, being critical for ensuring software quality and maintaining high performance [1]. Automating this process can play a significant role in improving software quality further and reducing manual testing efforts [2]. Search-Based Software Testing (SBST) has, among others, emerged as a powerful approach that automates the generation of test cases based on formulating testing tasks as optimization problems [3]. SBST aims to optimize specific criteria such as maximizing code coverage or fault detection.

Despite its strengths, SBST is known to have certain limitations, especially in terms of the understandability of the generated test cases. Tools like EvoSuite, which implement SBST, often achieve high test coverage, but lack in generating tests that are easily readable by developers and contain meaningful test values [4], often even generating random strings. This lack of readability reduces the practical utility of automated tests in real-world development, making them more difficult to maintain and use them effectively to find software bugs [5].

To address these limitations, two main approaches have been explored by researchers:

1) Capture/Replay Techniques: Tools that leverage capture/replay techniques, such as MicroTestCarver [6], utilize information from end-to-end (E2E) tests to generate unit tests that not only contain realistic test scenarios but also realistic test data. Although these approaches enhance understandability and relevancy, they often lag behind in coverage and cannot be used to further enhance understandability with for example explanatory comments.

```
@Test
public void testGetPropertyReturningNonEmptyString() {
    NamedEntity namedEntity = new NamedEntity();
    namedEntity.setProperty("Xt1Byl64')^>");
    String property = namedEntity.getProperty();
    assertEquals("Xt1Byl64')^>", property);
}
```

```
@Test
public void testGetPropertyReturningNonEmptyString() {
  // Given: A NamedEntity instance and a property value to be set
  NamedEntity namedEntity = new NamedEntity();
  namedEntity.setProperty("SampleEntityName");

  // When: The property is retrieved from the NamedEntity instance
  String property = namedEntity.getProperty();

  // Then: The retrieved property should match the value that was set
  assertEquals("SampleEntityName", property);
}
```

Listing 1: A test generated by EvoSuite (top) and the improved version by UTGen (bottom)

2) Integration of Large Language Models (LLMs): The recent developments in the area of large language models (LLMs) have opened up many possibilities for improving understandability in automated test generation [7]. UTGen, as an example, combines SBST with LLMs to achieve substantial improvements in test case understandability without compromising on coverage and pass rates [8].

Motivating example: Consider Listing 1, which compares a test case generated by EvoSuite with one enhanced by UTGen. The top example, produced by EvoSuite, uses a random string that lacks contextual meaning, making it difficult for developers to understand the intent of the test. In contrast, the bottom example from UTGen includes comments that clarify the test's purpose and attempts a meaningful test string, significantly improving its readability and relevance.

Given the potential value of Capture/Replay and LLM techniques, it highlights a significant opportunity for further enhancement. Currently, UTGen leverages general context data available from identifier and method names, but does not fully exploit specific, detailed execution data, resulting in suboptimal relevancy of test data [8]. As we will see in Listing 3, the `getProperty()` method does not actually refer to entity names, although this was implied by UTGen in Listing 1. Integrating the detailed trace logs from capture/replay techniques could potentially bridge this gap, providing richer context for the LLMs decision-making processes and resulting in test cases that are even more aligned with practical application scenarios.

This research's contribution includes the introduction of a novel approach that combines the capture/replay techniques from MTC with the natural language processing strengths of UTGen. By integrating detailed execution trace logs from E2E tests into UTGen, this approach, which we refer to as *UTGen+*, aims to leverage the best of both worlds, enhancing the readability and relevancy of the generated test cases, without compromising on test coverage.

We hypothesise that leveraging the traces from carved E2E tests and using them as contextual input for the prompts of UTGen will result more relevant and more readable test data and descriptions. The following questions can be asked to verify this hypothesis:

RQ$_1$ *How does integrating contextual data from the trace logs of end-to-end tests into the LLM-enhanced SBST approach in UTGen affect the understandability and relevancy of generated* ***comments*** *in test cases?*

RQ$_2$ *How does integrating contextual data from the trace logs of end-to-end tests into the LLM-enhanced SBST approach in UTGen affect the understandability and relevancy of generated* ***identifiers*** *in test cases?*

RQ$_3$ *How does integrating contextual data from the trace logs of end-to-end tests into the LLM-enhanced SBST approach in UTGen affect the understandability and relevancy of generated* ***test data*** *in test cases?*

In this research paper, we will present an implementation of *UTGen+*, combining capture/replay techniques with LLMs to enhance the readability and relevancy of SBST-generated test case. The effectiveness of this approach will be assessed through a user study that tests its impact on test case quality. This evaluation aims to verify UTGen+'s potential to improve automated software testing significantly.

2 Background

2.1 Search-based software testing

Automated test generation tools have been designed to minimize the effort required in testing [2]. Search-Based testing frameworks generally generate test suites from Java source code utilizing search-based optimization functions, guided by fitness functions [3]. EvoSuite [9] for example, uses an evolutionary search approach to evolve and optimize entire tests suites, and mutation based assertion generation to generate minimal but representative test cases.

Although generally resulting in a high coverage when desired, research has highlighted several challenges with tests generated in this manner, notably their generally lower understandability of test data and assertions, compared to tests written by humans [4].

2.2 LLM-based Unit Test generation

Unit test generation using LLMs typically involves either pre-training or fine-tuning LLMs specifically for the task or using advanced prompt engineering techniques to leverage pre-trained models without additional training [7]:

- *Pre-training/Fine-tuning:* According to Wang et al., early studies often used an approach where LLMs are first pre-trained on a considerable amount of code and testing examples, followed by fine-tuning on specific datasets related to unit tests. This approach attempts to help LLMs learn the relationship between code functionalities and their corresponding tests.
- *Prompt Engineering:* More recent methods often employ more advanced prompt engineering techniques, using zero-shot or few-shot learning, where LLMs generate test cases based on prompts constructed from code snippets.

As summarized by Wang et al., unit tests generated by just LLMs often lack in coverage, sometimes only reaching 2% of coverage on a complex dataset such as SF110 [10] [7].

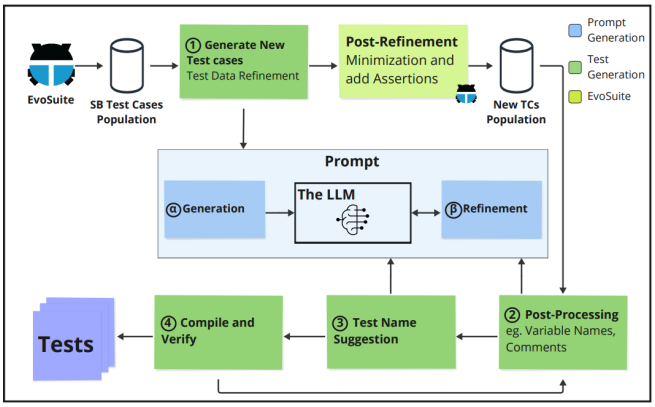

Figure 1: Architecture of UTGen

UTGen

UTGen [8] takes a different, more hybrid approach, incorporating LLMs into an existing SBST framework (EvoSuite) in several stages, as also shown in Figure 1 (taken from their original paper):

1. Refinement of Test Data: After initial test cases are generated using EvoSuite, LLMs are used to refine the test data, making the random strings more realistic.
2. Post-processing of Test Cases: LLMs update identifier names and add descriptive comments to make the tests more understandable.
3. Naming Tests: Automatically generate meaningful names for tests based on their content and purpose.

Additionally, UTGen addresses common issues with LLM outputs, such as hallucinations or irrelevant content, by employing methods like CodeBLEU to ensure that the enhancements preserve or improve the original intent and functionality of the test cases. Although it relies on a specific model like CodeLLama:7b-instruct, UTGen's design allows easy substitution of LLMs to adapt to different needs and capabilities.

The study further shows that UTGen's LLM-enhanced test cases not only improve understandability compared to EvoSuite, but also assist more effectively in bug detection and fixing, showing higher numbers of bugs addressed and the reduced time required to resolve them.

2.3 Capture/Replay Test Generation

Capture/Replay refers to the process of *recording* method calls, so that they can be *replayed* later, with the purpose of only testing a specific small set of functions [11]. This technique is also known as carving.

MicroTestCarver (MTC) [6] implements this idea to be able to generate realistic unit tests. As a basis, it takes an E2E test (either manual or scripted), and while instrumenting this test, method calls and input data are being recorded using a modified version of BTrace [12]. Afterwards, this data is parsed and is used to generate unit tests using a template based approach. The collected data is stored in a trace log, distinguishing between NodeMethods (NMs), which are central to the tracing process, and LeafMethods (LMs), which do not invoke further methods of interest. Trace logs and serialized objects are then parsed into a classmap, a hashmap that

```java
@Test
public void testGetPropertyReturningNonEmptyString() {
    NamedEntity namedEntity = new NamedEntity();
    namedEntity.setProperty("Xt1Byl64')^>");
    String property = namedEntity.getProperty();
    assertEquals("Xt1Byl64')^>", property);
}
```

Listing 2: Original EvoSuite Test

links classes to their methods, facilitating structured access to method calls for test generation.

MTC generates unit tests by integrating data from trace logs with source code analysis, using patterns like Arrange-Act-Assert [13] to structure tests for clarity. Test names are generated to reflect their function and input, enhancing understandability.

3 Implementation

3.1 Overview

This research integrates the two described tools: UTGen and MTC, attempting to enhance automated software testing by including contextual runtime data in *UTGen+*. Our approach involves extracting relevant parts from trace logs generated by the MTC-parser and incorporating these details into the prompts used by UTGen. The enriched prompts aim to improve the understandability and relevancy of the generated test suites. Figure 2 shows the general workflow of UTGen+, combined with how it fits into the original UTGen implementation. This implementation extends UTGen's original prompt generation, visible in Figure 1.

Since the additions of UTGen+ will be used in both the Test Data Refinement and Post-Processing stages of UTGen, the same implementation is used for both stages, only differentiating in which prompt template to use when conducting an LLM.

Consider exemplary Listing 2, a test generated by EvoSuite on the `getProperty()` method. The following subsection describes how this test will be improved in the Test Data Refinement stage, using UTGen+.

3.2 Detailed implementation

Phase 0: Trace Log Generation

Our implementation of UTGen+ does not include the trace agent used by the MTC, nor any other trace agent to reduce development time. For this research, we have utilized the traces already generated in the MTC project, and refer to their work for a comprehensive explanation of this process [6].

Listing 3 exemplifies the general structure of these trace logs by showing the traces for the `NamedEntity` class, referred to in the previous section. These traces have been modified for formatting purposes.

Phase 1: Parsing traces

The initial phase of our implementation involves parsing the trace logs into a structured hashmap (referred to as "Classmap"), similar to the methodology employed in MTC elaborated in Chapter 2. This structure is pivotal for organizing the extracted data in a way that is accessible for further processing.

```
public java.lang.String com.example.NamedEntity$getProperty:{
Args: []

Fields: [{
    name: entityName,
    type: java.lang.String,
    object: "John Doe"
    ....
}, {
    name: eyeColor,
    type: java.lang.String,
    object: "Green"
    ....
}]
Return: {
    name: null,
    type: java.lang.String,
    object: "Green"
    ....
}:

public java.lang.String com.example.NamedEntity$toString:{
    ....
}
```

Listing 3: Example traces for the NamedEntity class

In UTGen+, we have implemented the generation of the classmap before the evolutionary search of UTGen, to be notified in time when parsing issues arise. This classmap is generated only once to reduce processing time, and passed through several methods of UTGen in a wrapping TraceParser object before being used in the different generation stages.

Phase 2: Method matching

During the preparation phase of the LLM prompts for a given test case in UTGen, our process iterates over the method statements in this test. For every method invoked in the test, we cross-reference occurrences within the classmap and compile a list of all recorded usages of those methods. In the provided examples, a match will be found between the `namedEntity.getProperty()` of the EvoSuite Test in Listing 2 and the `getProperty` methodName in the classmap of Listing 4.

In an attempt to reduce prompt size and complexity, only one example per invoked method will continue to the next phases. The current implementation conducts a pseudo-random selection for this process if there are multiple recorded usages. More enhanced example selection, such as selection based on argument matching, is left for later research.

Here we also have to note external library functions and other leaf methods are not included in the classmap at this stage, meaning those methods in the test case will never be

```
[{
    "Key": "com.example.NamedEntity",
    "Value": [{
            "methodName": getProperty,
            "fields": [...],
            "return": {
                "type": "java.lang.String",
                "object": "Green",
                ....
            }
        },{
            "methodName": "toString",
            ...
        }]
    }
},
// Other classes...
]
```

Listing 4: JSON representation of the corresponding classmap

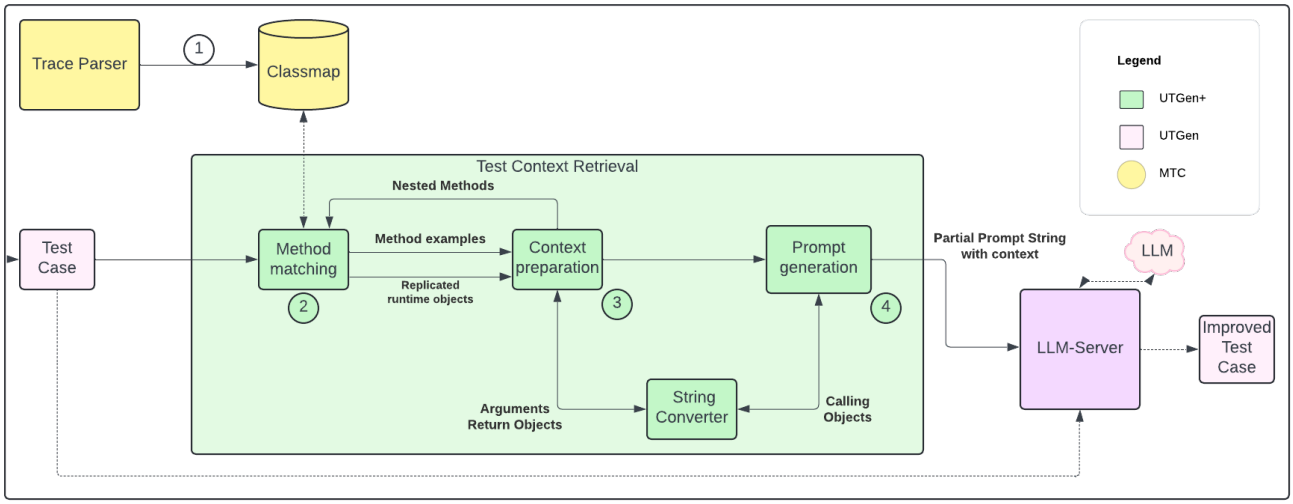

Figure 2: Architecture of UTGen+

exemplified by UTGen+. Although a well-trained LLM is presumed to possess some intrinsic knowledge of common libraries, the specific usage in the E2E-test could potentially be critical for understanding the interactions within the test. We recommend further research on this topic, regarding the effects of this and explore possible inclusion by further modifying the tracing agent.

Phase 3: Context preparation

For each selected recorded usage, we extract the fields of the calling object, their arguments, and the observed return values as documented in the trace logs. The results are represented as a string, visible in the last line of Listing 5.

Given that these arguments and return values can be complex, non-primitive or nested objects, we have created a custom utility class to convert their runtime Objects generated by the parser into a presentable String format using Java reflection.

To enrich the context further and enhance the LLM's understanding of how specific return values are derived from given arguments, we retrieve the nested method calls up to one level of depth. This decision attempts to balance the need for contextual depth with the focus on the task at hand [14]. Too many levels of nesting could potentially clutter the model's understanding with data that are irrelevant for the method-under-test. Further research is recommended for determining an optimal depth value.

In Listing 5, an example of this is added in lines 6 and 7, although these cannot be derived from earlier shown Listings for simplicity purposes.

Phase 4: Prompt preparation

In order to include the context in the prompts, we need to convert it to a presentable string for an LLM [15]. To achieve this, we have attempted first provide a string representation of the calling object, and then presenting the method in question as if was called on this object, as exemplified by Listing 5. Afterwards, some additional explanatory notes for the LLM

```
1| Now follows a runtime example of the getProperty() method:
2| Suppose this 1st example object:
3| NamedEntity namedentity{name: "John Doe", eyeColor: "Green", ...}

4| Example usage on this object:
5| namedentity.getProperty() //Returned: "Green"

6| Inside this method getProperty(), at least the following nested methods
   were called during runtime:
7| NamedEntity{name "John Doe", eyeColor: "Green", ... }
     .getEyeColor() //Returned "Green"
```

Listing 5: Related method context as included in a prompt

are added, and a similar representation for nested methods is appended if applicable.

In our implementation we make use of the OpenAI's GPT-4o API[1] (See also subsection 4.1), in contrast to UTGen's original implementation with CodeLLama:7b-instruct [8] which consistently produced non-compilable or non-passing test cases in the early development stages. Further research regarding this observation and an optimized LLM configuration are recommended.

Specifically, we are using the assistant role (System-User-Assistant model) to provide the actual context [16]. After using the same system instruction as UTgen, the User asks for runtime examples of the methods used by the test case. As an answer from the assistant, we provide the example from Listing 5. Afterwards, the User provides the test case and asks for improvements. Similarly to UTGen however, this implementation is pretty flexible and can easily be exchanged with different models [8].

By presenting the contextual data in this manner, we expect the LLM to understand the `getProperty()` method is actually a method to access the eyeColor property of the `NamedEntity`; something it would not be able to know without additional context. This is however just an illustration of how traces can benefit environments without much context available. In most cases the method names are expected to be more descriptive by itself, therefore already reducing the need

[1] https://platform.openai.com/docs/models/gpt-4o

```java
@Test
public void testGetPropertyReturningNonEmptyString() {
        NamedEntity namedEntity = new NamedEntity();
        namedEntity.setProperty("Blue");
        String property = namedEntity.getProperty();
        assertEquals("Blue", property);
}
```

Listing 6: Improved test data after LLM conduction

for additional context. If the `getProperty()` method would have been renamed to the more descriptive `getEyeColor()` we expect original UTGen to also generate a relevant String, as opposed to the `"sampleEntityName"` in Listing 1.

Phase 5: Response parsing
Since the output format of the LLM responses is not expected to change, compared to UTGen, we have not modified the parsing of responses and data extraction. From this stage, regular UTGen processes continue until a new LLM call for test generation will be made. Following the examples previously provided, we can expect an answer similar to the test case shown in Listing 6 in the Test Data refinement stage.

4 Experimental setup

To assess the effect of incorporating contextual runtime data from trace logs into the prompts with UTGen+, as compared to the traditional UTGen approach, we structured our experimental setup around the following research questions:

RQ$_1$ *How does integrating contextual data from the trace logs of end-to-end tests into the LLM-enhanced SBST approach in UTGen affect the understandability and relevancy of generated* ***comments*** *in test cases?*

RQ$_2$ *How does integrating contextual data from the trace logs of end-to-end tests into the LLM-enhanced SBST approach in UTGen affect the understandability and relevancy of generated* ***identifiers*** *in test cases?*

RQ$_3$ *How does integrating contextual data from the trace logs of end-to-end tests into the LLM-enhanced SBST approach in UTGen affect the understandability and relevancy of generated* ***test data*** *in test cases?*

This chapter outlines the dataset used, the methodology of generating and selecting tests, and the design of our comparative study.

4.1 Dataset and generation

As part of our experimental setup, we utilized the same projects specified in the MicroTestCarver (MTC) paper, with the exception of the Alfio project, which was excluded due to unresolved compilation issues. These projects were originally selected to be active and mature Java applications from different domains and of varying sizes [6]. The projects ultimately included in our test generation phase were LAB-insurance[2], Spring-Testing[3], and Pet-Clinic[4]. This selection allowed us to directly reuse the trace logs from the MTC replication package, which also contains the corresponding E2E tests.

[2]https://github.com/asc-lab/micronaut-microservices-poc
[3]https://github.com/hamvocke/spring-testing
[4]https://github.com/spring-projects/spring-petclinic

Application	Version	#Test Files	#Tests	#With Traces	Model
LAB-Insurance	1.0.0	11	83	37	*GPT-4o*
Petclinic	2.7.3	10	49	31	*GPT-4o*
Spring-Testing	0.0.1	2	31	8	*GPT-4o*

Table 1: Tests generated for evaluation

For our test generation, we used both the original UTGen and the enhanced UTGen+, using GPT-4o as our chosen model due to its sufficiently large context window and proven high coding capabilities [17]. The same seed was used across all test generations for GPT-4o and EvoSuite, to maintain reproducibility, and it was also verified all GPT responses had the same system fingerprint [18].

Table 1 presents the number of tests generated for each project, along with the subset of tests that utilized trace logs when generated with UTGen+. For the LAB-insurance project, classes were pre-selected on actually being present in the trace data, to reduce generation time.

The full generated tests along with an overview of the classes are available online [19].

4.2 Test selection

The selection of tests for our comparative study was conducted manually, based on the following criteria:

- *Trace Log Utilization:* Only tests where trace logs were actually employed in UTGen+'s post-processing prompts were considered.
- *Comparative Fairness:* Tests that failed at any stage in one implementation but succeeded in the other were excluded to maintain fairness in the comparison.
- *Test Data Completeness* Tests with complete test data were highly preferred in the selection. A noted bug in UTGen led to the removal of an undocumented amount of improved test data, resulting in the remaining tests appearing highly similar or missing test data, and potentially skewing the results.

We established these criteria to ensure that the selected tests were representative of realistic usage scenarios and conducive to a fair comparison between UTGen and UTGen+.

4.3 Study design

To address the research questions, a comparative user study was conducted with nine participants. This study was designed to evaluate and compare the naturalness and relevancy of test cases generated by UTGen+, UTGen, and *EvoSuite* across two distinct rounds. Naturalness in our study refers to the degree at which the test cases are readable and easily comprehensible, highlighting how intuitively the test behavior can be understood by developers [5] [8]. Relevancy, on the other hand, assesses whether the test elements are meaningful in the software's operational context [6].

Participants
The study involved nine participants, comprising six undergraduate students (all with more than one year of unit testing experience) and three graduates (BSc holders/MSc students). Each participant had prior experience with unit testing and had completed a formal course on software testing.

This background ensured that all participants had a sufficient understanding of the key concepts necessary to provide informed feedback on the test cases.

Survey

The nine participants were each prompted a total of 7 questions across 2 rounds. An overview of the questions can be found in our replication package [19].

Round 1: Assessment of Naturalness During the first round (questions 1 - 4), participants were presented with test cases generated by the three different implementations in a random order: UTGen+ (with trace logs), UTGen, and *EvoSuite*. They were asked to assess the naturalness/readability of each test case. This assessment was conducted using a Likert scale for three separate elements: comments, identifiers, and test data. Additionally, participants were required to provide qualitative feedback to justify their ratings, offering deeper insights into their perceptions of each implementation's output.

Round 2: Assessment of Relevancy The second round (questions 5- 7) focused on relevancy, with a similar setup to the first round. However, in this round, participants were provided with an extensive background explanation of the test cases' environment, all of which related to the "Pet Clinic" project. Again, they rated the relevancy of comments, identifiers, and test data using a Likert scale, providing qualitative justifications for their ratings.

Additionally, we implemented several strategies to minimize bias in the participants' responses:

- *Separation of Criteria:* By distinctively separating the naturalness/readability and relevancy assessments into two rounds, the study minimized the risk of participants conflating these criteria.
- *Additional Code Highlighting:* To reduce bias and prevent misunderstandings regarding which elements the questions referred to, the test code provided to participants was enhanced with additional code highlighting. This highlighting was based on the Monokai theme[5], with some modifications to comments and identifiers to further ensure clear visibility.
- *Randomization of Test Cases:* The order of test cases and the methods by which they were generated were randomized in the questionnaire to prevent any order effects or biases associated with particular tools.
- *Anonymity of Tools:* Participants were not informed of the generation methods of the given test cases, to prevent any preconceived biases toward specific tools affecting the participants' ratings.

Using this methodology we evaluated the potential improvements offered by UTGen+ compared to UTGen through a user study, and have prepared a solid foundation for analyzing the data and researching our hypothesis.

5 Results

This chapter presents the results of the comparative user study conducted to evaluate the naturalness and relevancy of test cases generated by UTGen+, UTGen, and EvoSuite. Violin graphs are used to illustrate the distribution and density of scores for naturalness and relevancy as rated by the nine participants. These graphs provide a visual interpretation of the data, highlighting the median scores and the variability of responses for each testing tool [6].

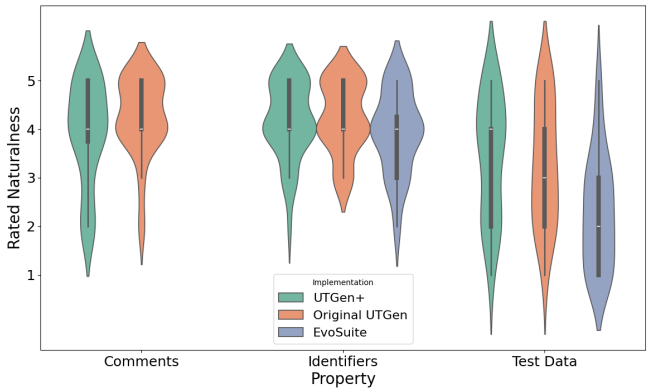

Figure 3: Naturalness Results

5.1 Naturalness

The violin graph displaying the aggregated results for naturalness (Figure 3) provides a quantitative comparison of the performance between UTGen+, UTGen, and EvoSuite, as rated by the participants.

Comments: The analysis of naturalness for comments reveals minimal distinction between UTGen+ and Original UTGen. Both implementations have the same median value, although UTGen+ displays greater variability as indicated by broader fluctuations in the data distribution. This suggests some inconsistency in the naturalness of comments generated by UTGen+, resulting in a visibly lower mean rating (Table 2).

Identifiers: Similarly, for identifiers, the data show little variance between the two versions of UTGen, with both implementations marginally surpassing EvoSuite in terms of naturalness.

Test Data: In terms of test data, UTGen+ demonstrates a marginal improvement in naturalness compared to Original UTGen. However, the distributions of the scores are nearly identical, indicating no to little improvement by UTGen+. Notably, both versions of UTGen were evaluated as being more natural than EvoSuite.

These observations suggest that, although there are slight variations in performance across different categories of test case components, the overall differences between UTGen+ and UTGen are minimal, with both generally outperforming EvoSuite in terms of naturalness.

Implementation	EvoSuite		Original UTGen		UTGen+	
	Mean_Rating	Median_Rating	Mean_Rating	Median_Rating	Mean_Rating	Median_Rating
Property						
Comments	-	-	4.25	4.00	3.92	4.00
Identifiers	3.92	4.00	4.22	4.00	4.17	4.00
Test Data	2.31	2.00	3.25	3.00	3.42	4.00

Table 2: Mean and Median Naturalness Ratings by Implementation and Property

[5]https://monokai.pro/

[6]https://www.labxchange.org/library/items/lb:LabXchange:46f64d7a:html:1

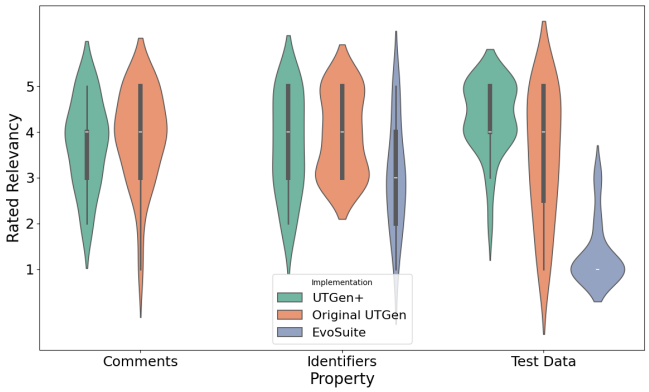

Figure 4: Relevancy results

Implementation	EvoSuite		Original UTGen		UTGen+	
	Mean_Rating	Median_Rating	Mean_Rating	Median_Rating	Mean_Rating	Median_Rating
Property						
Comments	-	-	3.89	4.00	3.74	4.00
Identifiers	3.04	3.00	3.93	4.00	3.70	4.00
Test Data	1.33	1.00	3.44	4.00	4.30	4.00

Table 3: Mean and Median Relevancy Ratings by Implementation and Property

5.2 Relevancy

The violin graph displaying the aggregated results for relevancy (Figure 4) allows us to compare how UTGen+, UTGen, and EvoSuite are perceived by users in terms of generating relevant test cases.

Comments: The user evaluations indicate that Original UTGen has a slightly better average performance in the relevancy of comments compared to UTGen+. However, UTGen shows greater variability in these scores, indicating less consistency in the results.

Identifiers: For identifiers, participants rated UTGen+ and Original UTGen similarly in terms of their effectiveness in generating relevant identifiers, with minimal differences between them. Both implementations outperform EvoSuite, similar to the pattern observed in the naturalness results.

Test Data: Regarding test data, UTGen+ is rated noticeably higher in relevancy compared to Original UTGen. Although both versions have the same median score, UTGen+ consistently achieves higher relevancy scores, with most of its scores around or above the median and having a higher mean. In contrast, scores for Original UTGen are more widely distributed, with the lower quartile at a rating of 2.5. Both UTGen versions visibly surpass EvoSuite in relevancy, which aligns with expectations considering EvoSuite's tendency to generate random or empty strings as test data [9]

This indicates that while UTGen+ and Original UTGen generally perform well in terms of relevancy, UTGen+ shows noticable improvements in the consistency and quality of test data generated, highlighting its effectiveness in integrating contextual runtime data.

6 Discussion

6.1 Results interpretation

RQ$_1$ *How does integrating contextual data from the trace logs of end-to-end tests into the LLM-enhanced SBST approach in UTGen affect the understandability and relevancy of generated* ***comments*** *in test cases?*

We observed a slight decrease in the mean rating for comments, from 4.25 to 3.92, a reduction of 7.8%. Answers to the qualitative question motivating the ratings suggest that this decrease may be attributed to differences in formatting by the LLM. Although variations in formatting were anticipated, they did not normalize as expected, possibly due to the limited sample size. Future research should investigate whether standardizing formatting would mitigate these effects. It's hypothesized that these factors could have disproportionately influenced the ratings, but further studies are needed to confirm this.

For relevancy, the decrease was even smaller, with the mean rating moving from 3.89 to 3.74 (-3.9%). Despite the small change, the distribution of scores shown in Figure 4 was very similar across the two versions. Participants generally perceived little difference in the necessity of comments, suggesting that slight variations in comment content were perceived as "unnecessary" rather than less relevant.

While the slight decrease in the naturalness and relevancy of comments generated by UTGen+ compared to UTGen can largely be attributed to minor formatting differences, further research is needed to determine how standardization or specific participant instructions regarding formatting perception could neutralize this effect.

RQ$_2$ *How does integrating contextual data from the trace logs of end-to-end tests into the LLM-enhanced SBST approach in UTGen affect the understandability and* ***identifiers*** *in test cases?*

The naturalness ratings for identifiers showed a very minimal decrease, from an average of 4.22 to 4.17 (-1.2%). This slight decrease might be influenced by the low complexity of the provided test cases, which already included descriptive method names providing sufficient context for generating appropriate identifiers. While both UTGen versions were deemed relatively natural, especially compared to EvoSuite, the small difference suggests a need for further research into how test case complexity influences the generation of identifiers.

The decrease in relevancy ratings for identifiers, from 3.93 to 3.70 (-5.9%), was not expected and requires additional research. The qualitative answers show UTGen's identifiers to be slightly more specific (e.g., 'patientVisit' vs. 'visit', 'testEntity' vs. 'namedEntity'), which was rated as more relevant by the participants. We hypothesize this missing specificity in UTGen+ could be caused by the fact test data are enhanced before identifiers, giving the two implementations a practically different environment when updating identifiers. Further research should explore the role of test data relevancy in the identifier generation process, and investigate the impact on relevancy of the order of generation of test data with respect to identifiers.

Given these insights, the minimal differences in naturalness and relevancy ratings between UTGen and UTGen+ for identifiers suggest that the existing descriptive quality of provided method names may limit the observable benefits of integrating additional contextual data from trace logs, and actually show an unexpected slight decrease in relevancy.

RQ$_3$ *How does integrating contextual data from the trace logs of end-to-end tests into the LLM-enhanced SBST approach in UTGen affect the understandability and relevancy of generated **test data** in test cases?*

An improvement in the naturalness of test data was noted, with mean ratings increasing slightly from 3.25 to 3.42 (+5.2%). The notable lower rating as compared to identifiers and comments can be explained by one of the questions failing to improve test data in both UTGen versions, resulting in a random EvoSuite string being included. The participants observed UTGen+ generating more natural test data and sticking less to the EvoSuite format. The fact that they are on average similarly rated can be attributed to a provided test on retrieving a productID; UTGen+ included a realistic product ID ("PRD_001"), whereas UTGen generated ("testProductID123"), with the participants indicating the latter as more natural. This goes to show that sometimes realism and relevancy can go at the cost of naturalness.

The relevancy of test data saw a significant increase, from a mean rating of 3.44 in UTGen to 4.30 in UTGen+ (+25.0%). This substantial improvement highlights UTGen+'s potential in generating contextually relevant test data. Participants noted the test data's format and suitability as more aligned with the real-world usage of the "Pet Clinic" project, provided to them in the question introduction, suggesting that the inclusion of trace logs effectively enhances relevancy of test data. While these findings are promising, further studies should confirm if trace logs consistently contribute to increased relevancy across diverse testing scenarios.

Given these insights, we observe slight improvements in the naturalness of test cases, but also have to note realistic data sometimes come at the cost of naturalness. We can observe a significant increase in the relevancy of the generated test data, showing the potential of UTGen+ when it comes to generating contextually relevant test data in a broader context.

6.2 Threats to validity

Our study's validity may be impacted by several factors:

- *Sample Size and Test Selection Bias:* The small sample size and selective nature of the test cases, chosen based on the criteria of subsection 4.1, might not represent broader testing environments with a bigger variety. Expanding the diversity of test cases could help validate the generalizability of our findings.
- *Formatting Influences:* Participants' feedback suggests that variations in formatting by the LLM between UTGen+ and UTGen might have influenced the ratings, though such effects were expected to normalize This observation shows potential for further methodological refinement, such as standardizing formatting across test cases to isolate the impact of content changes more effectively.
- *Complexity of Test Cases:* The minimal impact observed in enhancing identifier relevancy may relate to the relatively straightforward nature of the test cases used, which may not have fully challenged the capabilities of the UTGen+ system. Future studies should consider generating a range of test cases with a higher variance in complexity to better evaluate UTGen+'s performance across diverse software testing scenarios.
- *Project Selection:* The selected projects are available as open-source projects on GitHub, meaning they may or may not have been used in GPT-4o's training dataset, therefore the model perhaps has intrinsic knowledge of the projects making it easier to generate seemingly relevant data. In the comparison we compare UTGen+ and UTGen in the same environment and focus only on the observed differences, in an attempt to reduce this possible bias
- *Particiant Bias:* 8 out of 9 participants of the user evaluation are known to study or have studied at the same university. Being taught the same practices, they might have a certain bias towards specific formatting or best-practices, which could have potentially impacted the results. A broader sample set of participants is recommended in related researches to reduce these possible side effects.

7 Conclusions and Future Work

This study has focused on enhancing the understandability and relevancy of automated software test cases by integrating trace logs from end-to-end tests with a LLM-enhanced SBST tool called UTGen. The enhanced version, which we present as *UTGen+*, attempts to leverage detailed execution trace logs by providing them as contextual input in the prompts of UTGen, with the purpose of improving the naturalness and relevancy of test elements such as comments, identifiers, and test data.

Our findings reveal that while UTGen+ showed a marginal decrease in the naturalness of comments and identifiers, it drastically enhanced the relevancy and naturalness of test data. More specific, the relevancy of test data generated by UTGen+ was notably higher compared to that produced by Original UTGen. This suggests that the inclusion of contextual data from trace logs can enrich the content of generated test cases, making them more applicable and valuable in real-world testing scenarios. This partially confirms our initial hypothesis that more contextually informed test generation could bridge existing gaps in automated test case development.

Despite these encouraging results, we identified several limitations. The study's scope was restritedc to a select number of software projects and utilized only a single LLM configuration. Additionally, the selected test cases were relatively simple, which might not have fully tested the capabilities of UTGen+, and the limited sample size and similarity of participant backgrounds, suggest that the findings should be interpreted cautiously regarding their generalizibility.

Future research should focus on expanding the representability of test cases used to further challenge and evaluate the capabilities of UTGen+. Exploring different configurations of LLMs and extending the approach to include more diverse datasets and software environments would help in verifying and broadening the applicability of our findings. Additionally, investigating the effects of including examples based on matching arguments rather than random choice and extending the statement types analyzed, it could further improve the performance of UTGen+ and provide deeper insights in the opportunities of integrating trace logs in LLM-enhanced SBST.

8 Responsible Research

While conducting this research, we attempted to comply to high ethical standards and responsible research practices to ensure scientific integrity and reproducibility of our results:

Reproducibility

All code developed during this project, including scripts for test generation and data analysis tools, is publicly available on GitHub [19]. This allows other researchers to actively reproduce our results, promoting transparency and collaborative improvement. In all tools with pseudo-randomness involved, the random seed and system fingerprint (if applicable) have been documented for reproducibility. All generated classes and test cases are openly available in the same GitHub repository, as well as the Jupyter Notebooks used in the data analysis. The process for selecting test cases is detailed in Section 4.2, to ensure that other researchers can understand and replicate our methodology. This includes criteria for test case inclusion and exclusion, ensuring a transparent manual selection process, although automated selection would be recommended for later research, being more reproducible.

User Evaluation

We conducted our user evaluation through the Qualtrics[7] platform, the recommended platform by TU Delft for data collection. Measures were in place to prevent multiple entries by the same user and maintain integrity during data collection. All participants were provided with an Informed Consent Form at the start of the evaluation, ensuring all participants were fully informed about the nature of the research, the use of the data collected and their voluntary participation, all in accordance to the Netherlands Code of Conduct for Research Integrity (2018)[8].

Potential misuse

Although our research, and the automated generation of test cases in general, attempts to improve software quality, there is a potential misuse where technologies like *UTGen+* might be used to superficially pass tests and claim software quality, without actual thorough assurance. This could lead to undetected bugs and software vulnerabilities. While our research contributes to advancing automated testing technologies, it is crucial for developers and practitioners to remain cautious and ensure that these tools are used to genuinely enhance software quality, not to circumvent exhaustive testing processes.

Bias and threats to validity

As extensively analyzed in the discussion chapter, our study acknowledges potential biases and threats to validity, including the similar backgrounds of user study participants and the limited complexity of the test cases used. These factors are critical to consider for interpreting the study results and guide the need for broader testing scenarios in future research.

[7] https://www.qualtrics.com/

[8] https://www.nwo.nl/en/netherlands-code-conduct-research-integrity

References

[1] G. Tassey. The economic impacts of inadequate infrastructure for software testing, 2002.

[2] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. Does automated unit test generation really help software testers? a controlled empirical study. *ACM Trans. Softw. Eng. Methodol.*, 24(4), sep 2015.

[3] Mark Harman and Phil McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *Software Engineering, IEEE Transactions on*, 36:226 – 247, 05 2010.

[4] M. Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. An industrial evaluation of unit test generation: Finding real faults in a financial application. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 263–272, 2017.

[5] Ermira Daka, José Campos, Gordon Fraser, Jonathan Dorn, and Westley Weimer. Modeling readability to improve unit tests. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 107–118, New York, NY, USA, 2015. Association for Computing Machinery.

[6] A. Deljouyi and A.E. Zaidman. Generating understandable unit tests through end-to-end test scenario carving. In Leon Moonen, Christian Newman, and Alessandra Gorla, editors, *Proceedings of the 23rd IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Proceedings - 2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation, SCAM 2023, pages 107–118. IEEE Computer Society - Conference Publishing Services, 2023.

[7] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. Software testing with large language models: Survey, landscape, and vision, 2024.

[8] A. Deljouyi. Understandable test generation through capture/replay and llms. Presented at ICSE24, Lisbon, Portugal. Paper not published yet.

[9] Gordon Fraser and Andrea 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*, ESEC/FSE '11, page 416–419, New York, NY, USA, 2011. Association for Computing Machinery.

[10] Mohammed Latif Siddiq, Joanna C. S. Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinicius Carvalho Lopes. Using large language models to generate junit tests: An empirical study, 2024.

[11] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler. *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2021. [Online]. Available: https://www.fuzzingbook.org.

[12] Btrace - a safe, dynamic tracing tool for the java platform. https://github.com/btraceio/btrace, December 2022. [Online]. Available: https://github.com/btraceio/btrace.

[13] V. Khorikov. *Unit Testing Principles, Practices, and Patterns*. Manning, 2019.

[14] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 02 2024.

[15] Hengzhi Pei, Jinman Zhao, Leonard Lausen, Sheng Zha, and George Karypis. Better context makes better code language models: A case study on function call argument completion. *Proceedings of the AAAI Conference on Artificial Intelligence*, 37(4):5230–5238, Jun. 2023.

[16] Faren Yan, Peng Yu, and Xin Chen. Ltner: Large language model tagging for named entity recognition with contextualized entity marking, 2024.

[17] Li Zhong, Zilong Wang, and Jingbo Shang. Debug like a human: A large language model debugger via verifying runtime execution step-by-step, 2024.

[18] OpenAI. https://github.com/openai/openai-cookbook/blob/main/examples/Reproducible_outputs_with_the_seed_parameter.ipynb. [Online]. How to make your completions outputs reproducible with the new seed parameter.

[19] M. de Wit, A. Deljouyi, and A. Zaidman. https://github.com/UTGenPlus/UTGenPlus/. [Online]. Replication package of this UTGen+ project.