Effective mutation test generation for End-to-End test programs using large language models

Anonymous Author(s)

***Abstract*—**

***Keywords*—** Mutation Testing, LLM, Software Engineering

I. Introduction

The quality of web applications is a major concern in the software software life cycle. It is often costly and difficult to guarantee. It crucial for sensitive industries such as healthcare and banking. banking. To test a web application, we use several types of tests, which can be grouped into three levels of a pyramid. From the bottom of the pyramid pyramid, unit tests, service tests and End-to-End (E2E) tests. E2E tests are generally produced using frameworks such as Selenium or Cypress. [1]. End-to-End testing is a testing approach that aims to test the behavior of a web application as a whole from the end-user's perspective. As a result, the program under test (PUT) is a black box from the testing point of view. Among other things, this characteristic makes it difficult to assess the quality of E2E tests during both test creation and maintenance. Ricca et al. [2] have identified several challenges to the quality of E2E tests, including the problem of fragility. In general, mutation tests are used to assess test fragility [3]. Mutation testing is a testing technique that consists of introducing defects into the LUP source code and then evaluating the ability of the tests to capture these mutations [4]. This method is used in industry for unit testing, due to its white-box testing characteristic. However, a number of studies are beginning to focus on the application of mutation testing to E2E testing. Yandrapally et al. [5] proposes the MaewU framework, which evaluates user interface (UI) test suites. It introduces 16 mutation operations based on 250 bug reports.

Leotta et al. [6] subsequently proposed the MUTTA tool, which automates the mutation testing process for E2E tests. This work has shown that mutation testing is indeed the most relevant approach for assessing the quality of E2E tests, but several challenges remain. These include: (i) the high cost (execution time and number of mutants) of mutation tests, (ii) the identification and filtering of equivalent mutants, and (iii) the efficiency of mutation tests. In this work, we apply mutation test generation to address the following problem: *How do you generate efficient mutation tests for end-to-end testing by meeting the challenges of cost, identification and reduction of equivalent mutants?*

We propose an approach based on language models and PUT bug logs to generate closer-to-existing, necessary and diversified mutation tests. Our study is designed to answer the following research questions:

- **RQ1:** What is the most adapted model for generating mutation tests? *[Aurel: mutants introduits / mutants générés ?]*
- **RQ2:** Can LLM be used to identify equivalent mutants? If so, which model is best adapted to this task? *[Aurel: mutants équivalents / mutants générés ?]*
- **RQ3:** Is the cost (execution time and number of mutants) of the mutation tests generated by our approach reduced compared with the state-of-the-art approach?
- **RQ4:** Can LLM generate more efficient mutation tests than the state-of-the-art approach? *[Aurel: voir le rapport entre la répartition des mutants équivalents par rapport au nombre de mutants générés dans les 2 approches. Efficacité = mutants tués / mutants générés, temps d'exécution, nombre de mutants équivalents / mutants générés.]*

II. Background

A. Mutant Generation

A **mutant** is a small change in the source code of a PUT. The change is made to simulate a real-life defect in the code [7], [8]. The aim of mutation testing is to verify that a given test is capable of detecting the mutant by failing. This is known as killing the mutant. Mutants can be divided into two broad categories:

- **Equivalent Mutants:** Mutants that do not affect the behavior of the PUT. These mutants are not useful for testing.
- **Non-Equivalent Mutants:** Mutants that affect the behavior of the PUT. These mutants are useful for testing.

Mutant generation is a crucial step in the mutation testing process. It is important to note that mutant generation can be either manual or automatic. Manual approaches obviously require human intervention. Whereas automatic approaches use tools such as PIT [6], [9], Major [10], Jumble [11] and Javalanche [12]. On the other hand, its tools can generate equivalent mutants, which can lead to incorrect results when evaluating mutation tests and therefore low efficiency. In a recent work, Leotta et al. [6] used PIT to generate mutants, as the tool provides the high number of mutators needed for E2E mutation testing.

[Aurel: ML in Mutation Testing]
[Aurel: LLM in Mutation Testing]

III. Methodology

In this section, we present our methodology to generate mutation tests for E2E test programs. The methodology is illustrated in Figure 1. The methodology consists of the following steps:

- **Step 1:** *Generating mutant:* We generate mutants with adapted LLM models.
- **Step 2:** *Identifying equivalent mutants:* We use LLM or graph methodology to identify equivalent mutants.
- **Step 3:** *Sorting mutants:* In this step we sort then exclude equivalent mutants in the process. Exucluded equivalent mutants are stored in a database for future mutant generation. We are inspired by the Retrieval-augmented Generation (RAG) methodology *[Aurel: cite]* to enrich LLM use in first step.
- **Step 4:** *Introducing generated mutant in PUT:* We introduce the generated mutants in the PUT and check if PUT run successfully.
- **Step 5:** *Evaluating the generated mutants:* We evaluate the generated mutants by checking if they are killed by the E2E test suite.

IV. data

V. evaluation

VI. Conclusion

[Aurel: Actions : next steps + exemple de modeles, prompt, etc.]

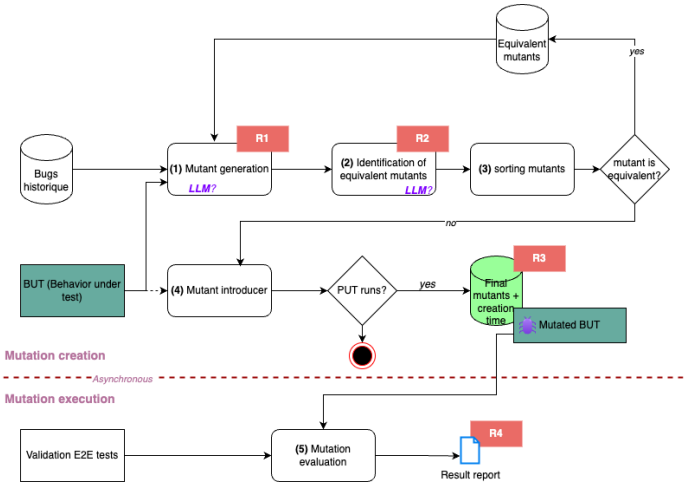

Figure 1: Methodology

REFERENCES

[1] M. Cerioli, M. Leotta, and F. Ricca, "What 5 million job advertisements tell us about testing: a preliminary empirical investigation," in *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, 2020, pp. 1586–1594.

[2] F. Ricca, M. Leotta, and A. Stocco, "Three open problems in the context of e2e web testing and a vision: Neonate," in *Advances in Computers*. Elsevier, 2019, vol. 113, pp. 89–133.

[3] S. Hamimoune and B. Falah, "Mutation testing techniques: A comparative study," in *2016 international conference on engineering & MIS (ICEMIS)*. IEEE, 2016, pp. 1–9.

[4] M. R. Woodward, "Mutation testing—its origin and evolution," *Information and Software Technology*, vol. 35, no. 3, pp. 163–169, 1993.

[5] R. Yandrapally and A. Mesbah, "Mutation analysis for assessing end-to-end web tests," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 183–194.

[6] M. Leotta, D. Paparella, and F. Ricca, "Mutta: a novel tool for e2e web mutation testing," *Software Quality Journal*, vol. 32, no. 1, pp. 5–26, 2024.

[7] A. J. Offutt and R. H. Untch, "Mutation 2000: Uniting the orthogonal," *Mutation testing for the new century*, pp. 34–44, 2001.

[8] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE transactions on software engineering*, vol. 37, no. 5, pp. 649–678, 2010.

[9] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "Pit: a practical mutation testing tool for java," in *Proceedings of the 25th international symposium on software testing and analysis*, 2016, pp. 449–452.

[10] R. Just, "The major mutation framework: Efficient and scalable mutation analysis for java," in *Proceedings of the 2014 international symposium on software testing and analysis*, 2014, pp. 433–436.

[11] S. A. Irvine, T. Pavlinic, L. Trigg, J. G. Cleary, S. Inglis, and M. Utting, "Jumble java byte code to measure the effectiveness of unit tests," in *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 2007, pp. 169–175.

[12] D. Schuler and A. Zeller, "Javalanche: Efficient mutation testing for java," in *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2009, pp. 297–298.