Round-Trip Mutation Testing: Translating Code to Natural Language Intent and back

Asma Hamidi[1], Cedric Richter[1], Ahmed Khanfir[2,1], Mike Papadakis[1]
{asma.hamidi, cedric.richter, michail.papadakis}@uni.lu
ahmed.khanfir@ensi-uma.tn
[1] *SnT, University of Luxembourg*, Luxembourg
[2] *RIADI, ENSI, University of Manouba*, Tunisia

***Abstract*—This paper presents Round-Trip Mutation Testing (RTM), a novel approach that generates mutants from LLM mistranslations between a program code and its intent. Leveraging the generative capability of LLMs from and to programming and natural language, and given an input program, our approach predicts its intent, that is used to generate programs, which when different from the original one, constitute the output mutants. The approach produces additionally mutants, stemming from artificially provoked mistranslations, by mutating the intent prior to the final programs (mutants) generation. Originating from the propagation of small changes in the intent to the code, our intuition is that these programs would present subtle semantic differences from the original one, simulating likely-to-occur faults that could result from specification misunderstandings, and enabling mutation testing. To evaluate RTM, we run it on 40 real buggy methods and evaluate its effectiveness and cost-efficiency in guiding testing towards detecting the bugs. Our results demonstrate the potential of round-trip mutation testing to produce syntactically more diverse mutants, potentially exposing faults that traditional mutation operators fail to reveal. More interestingly, RTM outperforms traditional pattern-based mutation in producing smaller and stronger test-suites, detecting on average over 4 and 1.7 times more faults when selecting only 4 and 30 tests respectively.**

***Index Terms*—Mutation Testing, Mutants, LLM, Round-Trip**

I. Introduction

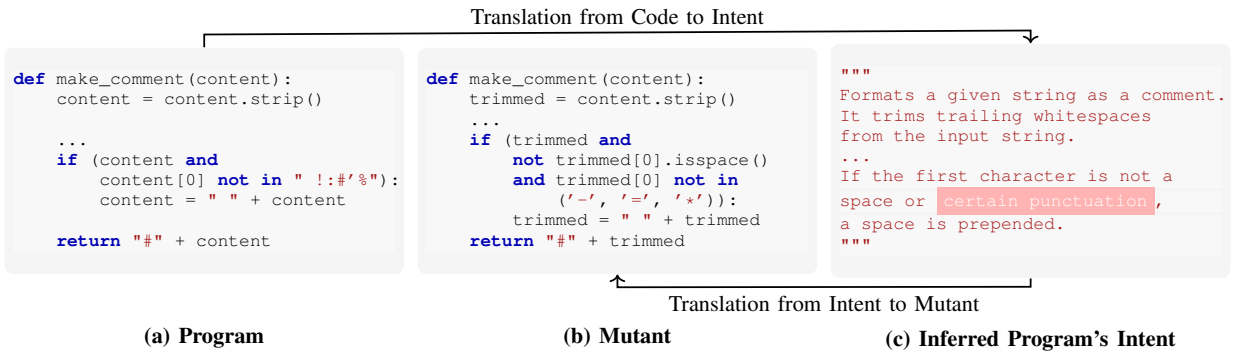
Several studies have proven that mutation testing is efficient in guiding testing towards higher fault-detection capabilities [1], [2]. It operates typically by generating different variants, so-called *mutants*, of the program under test by introducing syntactic alterations of its source code. Mutants are evaluated with respect to the current test suite, and a mutant is detected (or killed) if the test suite behaves differently on the mutant than on the original program. Undetected (or survived) mutants indicate potential untested corner-cases which the developer can cover by adding new targeted tests.

Throughout the last decades, mutation testing has been an important research topic in software engineering [2]–[6], leading to the proposition of numerous generation techniques [7]–[9] aimed primarily to reduce the cost of mutation analysis by minimizing redundancy while increasing coverage, ensuring diversity among generated mutants, and improving their realism and practical relevance, e.g. by simulating real-faults and resembling developer mistakes [10]–[13].

While existing approaches operate at source code level, the emergence of Large Language Models (LLMs) led to the introduction of Intent-based mutation [14], which applies the mutation to the underlying program intent. Not starting from the existing code and not relying on predefined mutation operators, enable the generation of completely new implementations, forming syntactically different and diverse mutants. To this end, intent-based mutation testing mutates the program intent, e.g. represented by the code documentation, and asks an LLM to generate code that corresponds to the mutated intents. By mutating the program's described semantics (intent), the approach is more likely to produce semantically dissimilar mutants from the original program that simulate faults arising from a developer's misunderstanding of the intent of the program. In practice, developer-provided descriptions (i.e. as method docstrings) are often scarce, limiting the applicability of intent-based mutation testing. In fact, only 34.2% of the methods in our evaluated real-world projects contain a docstring, as illustrated in Figure 2.

To address this limitation, we introduce *round-trip mutation testing* (RTM), a novel approach that mutates a program by performing a round-trip translation between code and a natural language description of its intent. Our approach relies on the observation that, while large language models (LLMs) are effective in generating code and code-intent, they may introduce noise and imperfections in the translation steps, thereby, deviating the semantics of the output programs from the original one. As a result, we obtain mutants representing possible misunderstandings of the program's behavior. Unlike prior works that use round-trip translation for LLM evaluation [15] and program repair [16], our approach leverages this process explicitly to generate mutants. Figure 1 illustrates an overview of round-trip mutation testing on an example program, where the subfigures *(a)*, *(c)* and *(b)* are respectively: the original code, its inferred intent and the resulting mutant. As can be seen, the translation from *(a)* to *(c)* is relatively accurate but misses the type of punctuation to check for, leading to the generation of code (mutant: from *(c)* to *(b)*) that is semantically close to the original implementation, but checks for different punctuation as in the original code.

We implemented four variants of our approach –mutating or not the program intent and asking for a precise or broad intent–



(a) Program **(b) Mutant** **(c) Inferred Program's Intent**

Fig. 1. **Round-trip mutation via Large Language Models.**

and evaluated its effectiveness and cost-efficiency in finding 40 faults from the BugsInPy dataset [17], i.e. guiding the selection of fault-revealing tests. To do so, we generate mutants on the buggy version of the program and select tests to kill them – simulating an actual testing scenario, where the bug is neither detected nor fixed. The results show that our approach can generate valid mutants that are syntactically more diverse than pattern-based mutation testing. More importantly, we observe that round-trip mutation outperforms traditional mutation testing, in selecting small test suites of higher fault detection capabilities. For example, when we restrict the selection to 4 tests, RTM yields test suites achieving on average ≈4.3 times higher fault-detection rates than the test suites selected by traditional mutants. When more tests are selected, this difference remains noticeable but gradually decreases, i.e., to ≈1.7 times with 30 tests until it becomes small after the selection of 34 tests, where RTM achieves its maximum avg fault detection of 44.6%, which is slightly higher than the 44% of traditional mutants. This endorses the potential of RTM in identifying fault-revealing tests earlier, making it suitable for the efficient construction of small test suites with large fault detection capabilities, while leaving room for improving its effectiveness in larger test suite scenarios.

In general, our contributions can be summarized as follows:

- We propose round-trip mutation testing, a novel approach that produces mutants from LLM mistranslations between natural and programming language, with a generalization of intent-based mutation to real-world scenarios, i.e. enabling it on non-documented and description-free code.
- We show that round-trip mutation can generate valid mutants that are syntactically distinct from traditional mutants.
- We provide empirical insights on the potential advantage of guiding test selection by round-trip mutation over pattern-based mutation, in achieving small test suites of higher fault detection capability.

II. Background

Mutation testing [2] is a test-adequacy criterion that evaluates test suite effectiveness based on its capacity to detect mutants. These mutants are typically obtained by modifying the target program code, using predefined and tuned mutation operators [2]–[4] which introduce small syntactic changes, e.g. replacing an arithmetic operator + with a −. A mutant is said to be *killed* by a test suite if its execution result on

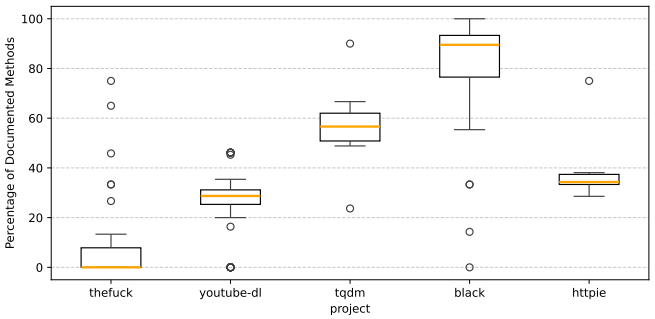


Fig. 2. Distribution of documented methods across the 5 projects

it differs from that on the original program, otherwise, it is said to be *survived* [5], [18]. Test suite thoroughness can then be approximated via its ability to detect (*kill*) mutants, where a higher mutant-killing ratio (*mutation score*) indicates a higher fault detection capability [19]. Survived mutants can be used as test-writing or generation target (writing tests to kill mutants), thereby, improving test suite thoroughness [20]. Mutation testing introduces also *duplicate* and semantically *equivalent* mutants to the original program. These form a major concern in mutation, falsifying mutation scores and causing unnecessary computational (i.e. test execution) and manual (i.e. analysis) overhead. Hence, the goal for effective mutation testing is to generate distinct *killable* mutants that can be used to construct adequate test suites.

III. Approach

A. Round-Trip Mutation

Given two domains $\mathbb{PL}$ (programs) and $\mathbb{NL}$ (natural language descriptions), round-trip mutation generates mutants by translating from $\mathbb{PL}$ to $\mathbb{NL}$ and back. To this end, round-trip mutation employs two models $M : \mathbb{PL} \rightarrow \mathbb{NL}$ and $M^{-1} : \mathbb{NL} \rightarrow \mathbb{PL}$ that ensure the transformation between the two domains. Given a program $P \in \mathbb{PL}$, a round-trip mutant P_m is thus defined as:

$$P_m = M^{-1}(M(P)), \tag{1}$$

where P_m is syntactically different from P. As can be seen in Figure 1, round-trip mutation testing with an *imperfect* translator, i.e. an LLM, can already yield valid mutants of the original program.

In addition to mutants resulting from M^{-1} and M imperfect translations, our approach generates mutants by propagating

You are given a Python function `function_name`:
```python
function_code
```
in a few sentences, generate a high-level description that explains the overall purpose or intent of the function, focusing on the general goal rather than implementation details.

(a) Broad

You are given a Python function `function_name`:
```python
function_code
```
in a few sentences, generate a precise description of the functionality of the code, focusing on implementation details and specific behavior.

(b) Precise

Fig. 3. Excerpts from the prompt template used to generate function intents at different precision levels from a Python method implementation.

intent-mutations into code. To do so, it applies a transformation $\mu : \mathbb{NL} \rightarrow \mathbb{NL}$ on the output of M (the program intent) before proceeding with the M^{-1} transformation (back to code).

More generally, for a given input program P, our approach generates mutants P_μ as follows:

$$P_\mu = M^{-1}(\mu(M(P))) \tag{2}$$

In the simplest case, μ is the identity function and thus $P_\mu = P_m$ (output of Equation 1). In our RTM first implementation, we use the same LLM for all transformations (M, M^{-1} and μ) and follow Hamidi et al. [14] approach in performing intent-mutation (μ), asking the LLM to propose word replacements that modify the intent's semantics while preserving grammatical correctness and semantic coherence.

B. Prompt Design

Accounting for the prompt design impact on our approach results, in particular, the M transformation from programming to a natural language, we employ two types of prompts (as shown in Figure 3):

Broad intent generation (Figure 3a):. which consists of instructing the LLM to generate a broad (high-level) description of the purpose of the program. This often excludes algorithmic details, making the description ambiguous and incomplete. As a result, the backward transformation M^{-1} has to fill-in the missing details, leading to implementations that are widely different than the original program.

Precise intent generation (Figure 3b):. which consists of further instructing the LLM to generate a more precise description of the program intent. This description includes important details about the implementation, e.g. steps to perform the computation, algorithms used, or handling of edge cases. As a result, the backward model can produce implementations that are significantly closer to the original program with only small implementation differences. Being more precise and preserving of the original program intent, we expect these descriptions to form a better starting point for intent-based mutations, enabling the generation of mutants that originate mainly from the propagation of intent-modifications.

TABLE I
SUMMARY OF BUGGY PROGRAMS USED IN THE STUDY

Project	#Buggy version	#Buggy Method
youtube-dl	14	15
tqdm	3	4
black	7	14
httpie	2	2
thefuck	4	5

IV. EVALUATION

Our evaluation study aims at answering these questions:

RQ1 (Mutants validity) *Can round-trip mutation testing generate valid mutants?*

RQ2 (Mutants Diversity) *How diverse are round-trip mutations compared to traditional pattern-based mutations?*

RQ3 (Test Guidance) *How effective and cost-efficient are round-trip mutants in guiding testing towards higher fault detection with fewer tests?*

A. Experimental setup

We perform our evaluation on 40 real buggy methods from 30 programs collected from 5 different open-source Python projects, available in the BugsInPy [17] dataset. Each fault is represented by a faulty version of the program and a bug fix that resolves the fault. We focus on faults that are fixed by changing at least one line of code in a given method. We report the number of faults selected per project in Table I.

Mutation Generation. To answer our research questions, we evaluate and compare four variants of round-trip mutation testing: (1) Broad, (2) Precise, (3) μBroad, and (4) μPrecise. The configurations Broad and Precise apply round-trip mutation with the respective prompts directly *without* intent mutation (as in Equation 1). μBroad and μPrecise map the method to a natural language description, mutate the description, and then map the mutated description back to a mutant. Mutants are generated on the *buggy* version of the code and we generate 10 mutants per method using GPT-4o-mini as LLM. In addition, we reimplemented[1] the mutation operators defined in MutPy [21] and use it as a representative baseline for traditional mutation testing, which we note as traditional mutants (trad. mut.).

Test Augmentation. We augment the developer test suite with LLM-generated tests. We use GPT-4o-mini to generate tests from the fixed version of each buggy method. In the process, we iteratively refine the generated tests to generate valid tests with high coverage. We ensure that each test suite has at least one fault-revealing test. Overall, with the generated tests, we achieve coverage score of 89.9% to 94.4% per method under test. Figure 4 shows the total number of tests and the ratio of bug-revealing tests per buggy method.

Mutation Testing Simulation. In our experiments, we simulate a developer using mutation testing to write fault-revealing

[1]We originally planned to use MutPy with BugsInPy, but found that MutPy could not be applied due to dependency issues.

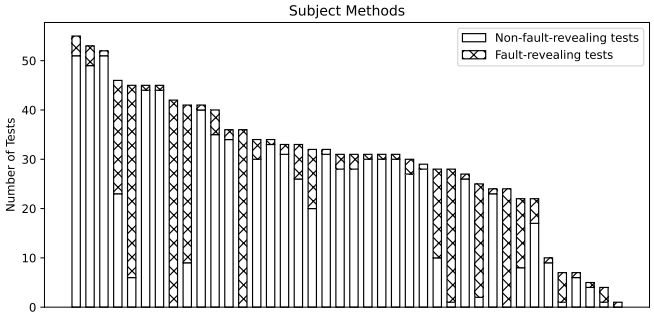


Fig. 4. Number of tests and fault-revealing tests.

tests. Hence, we generate mutants for the faulty version of the method under test (similar to [22]). A mutant is killed if there is a measurable difference between the test outcome of the mutated version and the original (faulty) version of the project. Meaning, their executions end with at least one different test result (pass/fail), or at least one assertion receiving a different value, e.g. the returned value from the method under test. All other mutants are surviving, and the developer would write tests to kill them. In our simulation, we start with an empty test suite (ts) and incrementally add tests from the full test suite (TS). To this end, we randomly pick a mutant that is not yet killed by the current test suite (ts). Then, we randomly select a killing test (one at a time) from the full test suite (TS) to be added to our current tests (ts). If a mutant cannot be killed, we move on without adding a test and randomly select the next mutant. In this process, a fault is detected as soon as we add a fault-revealing test. Since a fault-revealing test might be selected by accident, we repeat our simulation 100 times and report the average fault detection rate of the resulting test suite at different testing budgets.

V. Results

A. RQ1: Can round-trip mutation generate valid mutants?

To answer **RQ1**, we start by executing the mutants against the full test suite. We say that a mutant of a Python method is valid iff (1) the mutant passes syntactic validation of the Python interpreter and (2) the mutant is runnable, i.e. it can be tested without raising any runtime errors.

Results. Table II summarizes our results, reporting the percentages of mutants per approach that are (1) syntactically invalid, (2) surviving, (3) killable but incompetent (i.e. mutants that fail to run due to runtime errors) and (4) killable and runnable. As expected, Precise RTM produces the highest number of surviving mutants. The mutants are often closer to the original implementation than the other RTM variants which may result into equivalent mutants. Still, 65.5% are valid mutants that can be killed by the test suite. Both mutation and instructing the LLM to generate broader descriptions of the intent can help reduce the number of surviving mutants. Yet, the broader descriptions yield the highest number of valid mutants. Although traditional mutation produces a higher ratio of valid mutants, the percentage of killable mutants remains comparable of around 70%. Whereas traditional mutation produces more surviving mutants (17.9%; nearly 16% more suriving mutants than RTM), round-trip mutation produces relatively more incompetent mutants.

TABLE II
Round-Trip Mutants Vs Traditional Mutants

Approach	Invalid	Surviving	Killable	
			Incompetent	Runnable
Round-trip				
Broad	0.25%	0.25%	28.74%	70.75%
Precise	0.00%	2.00%	32.75%	65.50%
μBroad	0.50%	0.00%	33.00%	66.50%
μPrecise	0.25%	1.50%	47.75%	50.50%
trad. mut.	0.00%	17.93%	10.05%	72.01%

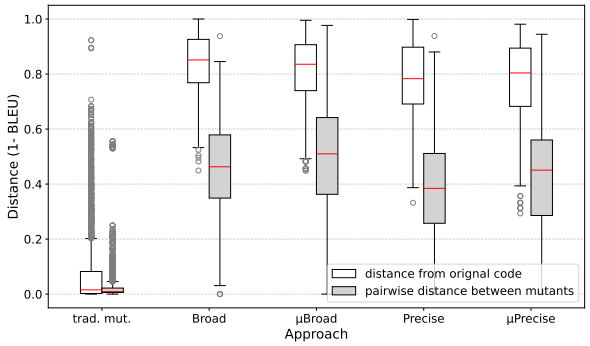


Fig. 5. Distribution of distances (1 - BLEU score) for code mutations. Boxplots show distances between each mutated method and its original implementation (white), as well as pairwise distances between mutants (grey).

B. RQ2: How syntactically diverse are round-trip mutants?

To answer this question, we investigate how syntactically close mutants are to the original code, as well as between mutants themselves. Prior to any computation, we remove docstrings to focus only on code. We then compute the BLEU distance (1 - BLEU score) of each mutant–original pair and of each mutant–mutant pair. We report the distributions achieved by each approach including the four RTM variants and the syntactic baseline.

Results. Figure 5 depicts the distributions of BLEU distances. Traditional mutants remain very close to both the original implementation and to each other, due to the fine-grained syntactic change. In contrast, RTM variants show a greater syntactic deviation and diversity. Mutants generated from Broad intents (which incorporate less information on the code) tend to produce more distant mutants compared to Precise variants, enabling the exploration of a wider spectrum of mutants beyond traditional mutation. Additionally, applying mutation to the intent increases diversity, resulting in mutants that are more varied among themselves.

C. RQ3: How effective is round-trip mutation testing for test guidance?

To answer **RQ3**, we simulate a mutation-guided test-selection scenario, and compare the fault detection ratios of the obtained test-suites by each approach, at each test-selection step. In addition, we include a random selection baseline, to contrast the advantage brought by mutation guidance.

Results. Figure 6 shows the result of our mutation testing simulation, reporting the percentage of faults that can be detected by selecting x tests (averaged over 100 runs). Although mutation testing achieves a significantly higher fault

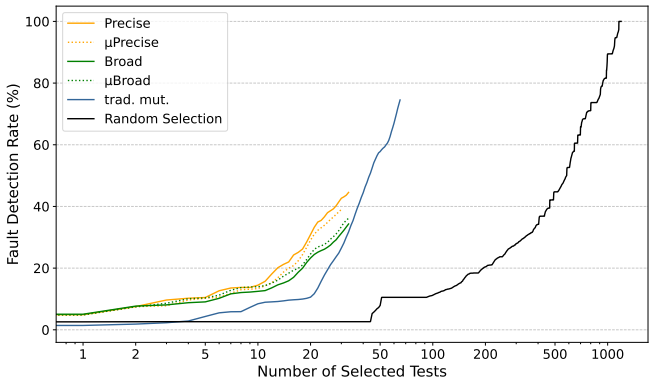


Fig. 6. Mutation testing simulation. The x-axis represents the number of tests selected by each approach (log scale) and the y-axis represents the fault detection rate of the selected tests across 100 simulations.

detection rate than random testing early on, all approaches saturate after around 66 selected tests. In these cases, all mutants are either killed or are not killable by the full test suite and hence no further tests are selected. Overall, RTM is more effective in selecting fault-revealing tests when the test budget is small (between 1 and 30 tests). In fact, it produces test suites achieving on average ≈4.3 and ≈1.7 times higher fault-detection rates than traditional mutants, when selecting only 4 and 30 tests, respectively. When selecting about 42 tests all approaches reach similar fault detection rates, with RTM achieving its maximum avg fault detection of ≈44.6%, which is slightly higher than the ≈44% of traditional mutants. Traditional mutation testing performs better for larger budgets (beyond 42 tests), reaching a maximum fault-detection of ≈74.5% after the selection of ≈60 tests. Perhaps surprisingly, while the precise intent produces less runnable mutants, it achieves a higher fault detection rate in comparison to all other round-trip mutation testing variants.

VI. Discussion

Based on our results, we see round-trip mutation testing as a promising approach for fault detection with a small test budget. However, we identified two current limitations of our approach which we aim to address in the future:

Impact of Intent Mutation. We initially expected that the mutation of the intent (inline with [14]) increases fault detection. Yet, our experiments show the contrary: the highest fault detection rate at a given test budget is achieved without applying intent mutation. We suspect that the problem stems from the fact that intent mutation increases the number of incompetent mutants (15% and 4.26% increase, respectively) in comparison to original generated intents. Therefore, we are currently investigating two strategies: (1) providing additional context during back translation and (2) using static or dynamic feedback to refine mutants. These strategies are motivated by the observation that functions in real projects are often tightly integrated and have complex dependencies that the code generation LLM might not be aware of.

Semantic Diversity of RTM Mutants. We further investigate why RTM selects fewer tests than traditional mutation testing in **RQ3**. To be effective in this scenario, mutants need to show diverse behavior, i.e. breaking different tests. In other words, if all or most mutants are killed by the same tests, the mutation analysis stops when very few tests are selected, leading consequently to an early plateau of test-suite fault detection. In fact, when investigating the percentage of semantically *unique* mutants (i.e. killed by a different set of tests; see Table III), we observe that more than 38% of traditional mutants are unique, compared to 20% of RTM mutants across all its variants.

TABLE III
Average Proportion of Semantically Unique Mutants Generated by Each Approach.

Approach	Runnable	All
Round-trip		
Broad	31.90%	21.10%
Precise	24.30%	16.30%
μBroad	33.30%	22.60%
μPrecise	40.90%	19.80%
trad. mut.	45.75%	38.44%

Threats to validity. Our results and conclusions might be affected by the nondeterministic nature of the employed LLMs. While it may produce different mutants, generating multiple implementations per program helps mitigate this threat and ensure the reproducibility of the overall results. Similar threats could emerge from the random selection of mutants and tests in our simulation. To mitigate this, we run it 100 times for every program and approach. In addition, relying primarily on automatically-generated tests can threaten the generalizability of our claims. We address this threat by ensuring high coverage, keeping only test suites containing fault-revealing tests and augmenting missing ones with developer-written tests from the dataset. A potential threat could arise from the used dataset (the selected faulty programs), and the focus on Python. Although we cannot ensure that the results generalize to different projects in other languages with different typing and runtime behaviors, we selected real programs from different open-source projects. Finally, while our syntax-based mutation baseline is a re-implementation, it uses the same operators as the standard mutation tool for Python, MutPy.

VII. Conclusion

In this paper, we introduce a novel LLM-based mutation testing approach, which generates mutants by applying a round-trip translation to the input program; from code to intent and back to code. By mutating the generated intent, the approach achieves intent-based mutation without relying on a pre-existing intent (e.g. developer-written documentation). The evaluation results on real Python bugs show that this approach produces valid mutants and can guide testing toward considerable fault detection rates with only few tests. Not involving any operators and not requiring any specific knowledge on the programming language at hand, we believe that the approach is easy to use in practice, particularly with the omnipresence of LLMs in all development environments and processes.

Acknowledgment

This work is supported by the Luxembourg National Research Fund (FNR) through the CORE project C23/IS/18182513/MiCE.

REFERENCES

[1] M. Papadakis and Y. Le Traon, "Metallaxis-fl: mutation-based fault localization," *Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 605–628, 2015.

[2] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman, "Chapter six - mutation testing advances: An analysis and survey," *Advances in Computers*, vol. 112, pp. 275–378, 2019.

[3] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge University Press, 2008.

[4] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," *ACM Trans. Softw. Eng. Methodol.*, vol. 5, no. 2, pp. 99–118, 1996.

[5] M. Marcozzi, S. Bardin, N. Kosmatov, M. Papadakis, V. Prevosto, and L. Correnson, "Time to clean your test objectives," in *International Conference on Software Engineering, ICSE*, 2018, pp. 456–467.

[6] M. Kintis, M. Papadakis, A. Papadopoulos, E. Valvis, N. Malevris, and Y. L. Traon, "How effective are mutation testing tools? an empirical analysis of java mutation testing tools with manual analysis and real faults," *Empir. Softw. Eng.*, vol. 23, no. 4, pp. 2426–2463, 2018.

[7] Y. Ma, J. Offutt, and Y. R. Kwon, "Mujava: an automated class mutation system," *Softw. Test. Verification Reliab.*, vol. 15, no. 2, pp. 97–133, 2005.

[8] T. Laurent, M. Papadakis, M. Kintis, C. Henard, Y. L. Traon, and A. Ventresque, "Assessing and improving the mutation testing practice of pit," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, March 2017, pp. 430–435.

[9] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "Pit: A practical mutation testing tool for java (demo)," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, p. 449–452.

[10] R. Degiovanni and M. Papadakis, "μbert: Mutation testing using pre-trained language models," in *15th IEEE International Conference on Software Testing, Verification and Validation Workshops ICST Workshops*, 2022, pp. 160–169.

[11] A. Khanfir, R. Degiovanni, M. Papadakis, and Y. L. Traon, "Efficient mutation testing via pre-trained language models," *arXiv:2301.03543*, 2023.

[12] J. Patra and M. Pradel, "Semantic bug seeding: A learning-based approach for creating realistic bugs," in *ESEC/FSE Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, p. 906–918.

[13] A. Khanfir, A. Koyuncu, M. Papadakis, M. Cordy, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Ibir: Bug report driven fault injection," *ACM Trans. Softw. Eng. Methodol.*, may 2022.

[14] A. Hamidi, A. Khanfir, and M. Papadakis, "Intent-based mutation testing: From naturally written programming intents to mutants," in *2025 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2025, pp. 347–357.

[15] M. Allamanis, S. Panthaplackel, and P. Yin, "Unsupervised evaluation of code llms with round-trip correctness," *arXiv preprint arXiv:2402.08699*, 2024.

[16] F. V. Ruiz, A. Grishina, M. Hort, and L. Moonen, "A novel approach for automatic program repair using round-trip translation with large language models," *arXiv preprint arXiv:2401.07994*, 2024.

[17] R. Widyasari, S. Q. Sim, C. Lok, H. Qi, J. Phan, Q. Tay, C. Tan, F. Wee, J. E. Tan, Y. Yieh *et al.*, "Bugsinpy: a database of existing bugs in python programs to enable controlled testing and debugging studies," in *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2020, pp. 1556–1560.

[18] M. Kintis, M. Papadakis, and N. Malevris, "Employing second-order mutation for isolating first-order equivalent mutants," *Softw. Test. Verification Reliab.*, vol. 25, no. 5-7, pp. 508–535, 2015.

[19] M. Ojdanic, A. Garg, A. Khanfir, R. Degiovanni, M. Papadakis, and Y. Le Traon, "Syntactic versus semantic similarity of artificial and real faults in mutation testing studies," *IEEE Transactions on Software Engineering*, vol. 49, no. 7, pp. 3922–3938, 2023.

[20] M. Ojdanic, A. Khanfir, A. Garg, R. Degiovanni, M. Papadakis, and Y. Le Traon, "On comparing mutation testing tools through learning-based mutant selection," in *2023 IEEE/ACM International Conference on Automation of Software Test (AST)*. IEEE, 2023, pp. 35–46.

[21] "Mutpy: Mutation testing tool for python 3.x code," https://github.com/mutpy/mutpy, 2019, accessed: 2026-03-12.

[22] T. T. Chekam, M. Papadakis, Y. L. Traon, and M. Harman, "An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption," in *International Conference on Software Engineering, ICSE*, 2017, pp. 597–608.