Intent-Based Mutation Testing: From Naturally Written Programming Intents to Mutants

Asma Hamidi
SnT, University of Luxembourg
Luxembourg
asma.hamidi@uni.lu

Ahmed Khanfir
Medtech, South Mediterranean University
Tunis, Tunisia
ahmed.khanfir@medtech.tn

Mike Papadakis
SnT, University of Luxembourg
Luxembourg
michail.papadakis@uni.lu

***Abstract*—This paper presents intent-based mutation testing, a testing approach that generates mutations by changing the programming intents that are implemented in the programs under test. In contrast to traditional mutation testing, which changes (mutates) the way programs are written, intent mutation changes (mutates) the behavior of the programs by producing mutations that implement (slightly) different intents than those implemented in the original program. The mutations of the programming intents represent possible corner cases and misunderstandings of the program behavior, i.e., program specifications, and thus can capture different classes of faults than traditional (syntax-based) mutation. Moreover, since programming intents can be implemented in different ways, intent-based mutation testing can generate diverse and complex mutations that are close to the original programming intents (specifications) and thus direct testing towards the intent variants of the program behavior/specifications. We implement intent-based mutation testing using Large Language Models (LLMs) that mutate programming intents and transform them into mutants. We evaluate intent-based mutation on 29 programs and show that it generates mutations that are syntactically complex, semantically diverse, and quite different (semantically) from the traditional ones. We also show that 55% of the intent-based mutations are not subsumed by traditional mutations. Overall, our analysis shows that intent-based mutation testing can be a powerful complement to traditional (syntax-based) mutation testing.**

I. Introduction

Mutation testing has long been recognized as one of the most powerful testing techniques [1], [2]. It generates program variants by altering the way programs are written, i.e., by making simple syntactic changes to the code under test. These variants are then used as targets for differential program analysis, that is, test writing (or test selection) with the aim to distinguish the behavior of the original program from that of the variants. When a test triggers a difference in the behavior of the mutant and the original programs, the mutant is considered as covered, called 'killed', otherwise is considered as not covered and called 'live'. The effectiveness of the test suites is then measured by the mutation score, the proportion of mutants killed over all considered mutants [2].

Traditional mutation testing operates at the program syntax level and thus is typically oriented toward errors that are syntactically small, i.e., the syntactic distance of the variants to the original program is rather small. For example, a typical mutation is to replace one operator such as '$>$' with another '$>=$' in a relational expression. This approach allows the introduction of subtle semantic deviations that make mutation effective at testing the behavioral boundaries of the programs under test. At the same time, this approach limits testing to the program logic that is actually implemented, making mutation testing less effective in revealing complex behavior-oriented (falling on the core of business logic) and omission faults [1].

To address these issues, we propose a novel approach to mutation testing, namely *intent-based mutation testing*. An intent is the programmer's objective for the code, described informally in natural language and offers a description of the task that is implemented. For example, if a programmer intends to create a function that calculates the factorial of a number, the intent could be formulated as "a function that takes an integer as an input and calculates its factorial". Intent-based mutation aims at testing programs by formulating alternative implementations of the same intents (intents implemented in the original program) as well as by formulating intent variant implementations corresponding to slightly altered intents (mutated intents). In other words, we generate mutated intents and contrast their respective implementations. We consider that intent-based mutation testing can target potential misunderstandings of the program's intents or specifications, which include faults that are hard to capture with traditional mutation.

By making minor adjustments to the initial intents of the program under test, it is possible to introduce mutations that reflect (small) misunderstandings of the actually implemented programming intents. These adjustments lead to an implementation that is being interpreted differently from what was originally done. Additionally, mutated intents can lead to mutations that include global transformations spanning across the entire intent implementation, e.g., an entire method. This approach contrasts with traditional mutation testing, where small and local changes are made to the programs syntax.

Intent-based mutation testing aims to find a different class of faults and should complement traditional mutation testing. When dealing with a programming intent, intention-based mutation can be seen as a process that mutates the intended behavior/specifications rather than the program. Previous work has investigated ways to create mutations considering the code context of mutation points [3]–[5], but these approaches are fundamentally limited to traditional syntactic mutations and therefore share the limitations of traditional mutation testing.

979-8-3315-3467-7/25 © 2025 IEEE

ICST Workshops 2025, Naples, Italy
Mutation 2025

Accepted for publication by IEEE. © 2025 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/ republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

We release intent-based mutation testing by using automatic programming tools, such as Large Language Models, to automatically formulate program implementations based on programming intents written in natural language. We generate intent variants by asking the tools to directly generate variant implementations and by mutating the natural language descriptions, which are then turned into actual programs and form our intent-based mutations.

We evaluate intent-based mutation on 29 HumanEval+ programs and show that it generates syntactically complex mutations that are semantically different from traditional mutations and diverse. Perhaps more importantly, we also show that more than 23% of the intent-based subsuming mutations cannot be detected by any of the traditional mutation-killing tests. Overall, our analysis corroborates the finding that intent-based mutations are strong and introduce faults that are not captured by traditional mutation testing techniques.

This paper makes the following contributions:

- We present intent-based mutation testing, outline its role in testing, discuss its difference from traditional mutation testing, and detail how it can be implemented. We also outline future research directions towards semantic and specification-driven testing approaches.
- We show the ability of intent-based mutation testing to generate complex and semantically diverse mutations, which cannot be detected by traditional mutation-based tests, i.e., tests generated to kill traditional mutations.
- We show that intent-based mutation has the potential to expand the fault detection abilities of mutation testing by revealing 23% more faults than those revealed by traditional mutation.

II. Background and related work

A. Mutation Testing

Mutation testing typically operates by introducing a few changes to the program code, thereby creating many different versions of it (named *mutants*) [2]. Those changes are usually obtained by applying predefined patterns called *mutation operators* [2] on the target code, which can, for example, invert relational operators (e.g., replacing $\geq$ with $<$) or arithmetic operators (e.g., replacing $+$ with a $-$), etc. Mutants can be used to assess the strength of test suites, by measuring their ability to trigger different behaviors from the original program. If a test suite fails when executed on a mutant, it is said to be *killed*, else, it is said to be *live* or *survived*. As some mutants cannot be killed, i.e., if they are functionally equivalent to the original program, they are said to be *equivalent*, otherwise, they are said to be *killable* [6]. By computing the ratio of killed mutants by a test suite (among all the generated ones), we can measure the test suite adequacy. This ratio is called the *mutation score*. The live mutants can serve as testing objectives and guide developers in writing effective tests [6]. Mutation testing techniques generate redundant mutants that can be *duplicated* [7] or *equivalent* [6], [8] to the original program.

Much research have focused on improving the efficiency of mutation testing, aiming at increasing the coverage, diversity and real-fault coupling while reducing the redundancy among the generated mutants. This has resulted in the proposal of several pattern-based mutation techniques [9]–[11] that rely on syntactic mutation operators. These operators have been designed mainly on the basis of the target programming language grammar and have been empirically tuned through multiple studies [2], [6], [12]–[14] to increase their effectiveness.

With the advancement of machine learning, recent research has focused on generating mutants based on real faults. For instance, Tufano et al. [5] and Zao et al. [15] proposed neural machine translation techniques to inject faults, trained on real bug fixes. Patra et al. [16] proposed also a learning approach, that adapts then applies pre-learned fault patterns on the target project. Khanfir et al. [17] proposed the usage of bug reports together with inverted automated-program-repair operators to inject faults. Their results are promising, however, may be of limited usability, depending on the availability of good bug reports or diverse and untangled fix commits [18].

Degiovanni et al. [19], [20] proposed μBERT a context-aware mutation testing technique which does not rely on historical bugs or the language grammar but rather on LLM, i.e. CodeBERT [21], knowledge of developer code. This approach mutates the target program by replacing its tokens one at once with inaccurate CodeBERT predictions, producing several likely-to-occur mutants. Empirical comparative studies [22], [23] with other learning and pattern-based approaches, give evidence of its high efficiency and cost efficiency in generating mutants that couple and reveal real faults, which makes it a suitable comparison baseline for our approach.

Unlike those approaches, our approach does not apply changes to the program code, but to its intent, written in natural language, instead. Hence, it does not depend on any prior particular knowledge, i.e. historical real bugs or programming language grammar. In fact, it relies solely on the natural language comprehension and code generation capability of LLMs.

B. Large Language Models

Large Language Models (LLMs) [24]–[26], such as GPT (Generative Pre-trained Transformer), have revolutionized the field of natural language processing (NLP) with their ability to understand and generate human-like text. One of the remarkable capabilities of these models is their efficiency in generating code from natural language descriptions. GPT models, in particular, can interpret a user's intent expressed in plain language and translate it into executable code across various programming languages. This is achieved through extensive training on diverse datasets, including code repositories and documentation, enabling the model to learn syntax, semantics, and common programming patterns. As a result, developers can leverage GPT to automate coding tasks.

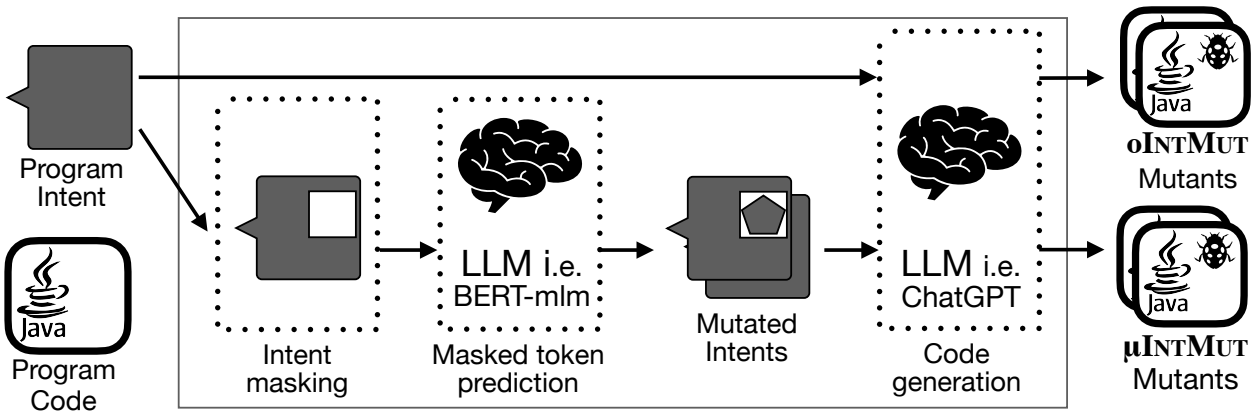

Fig. 1: Intent-based mutation testing workflow.

III. Approach

Our approach uses BERT [27] to mutate the natural intent, by replacing its tokens with the inaccurate predictions of the model. We invoke its Masked Language Modeling (MLM) pipeline to predict replacements of masked tokens from the input intent, based on the context of the remaining intent text. As the model has been trained on a large corpus, and is able to write human-like text, even when inaccurate, we expect its predictions to result into meaningful mutated intents.

We leverage the capability of Large Language Models (LLMs) to link natural language descriptions (task descriptions) and code to generate mutations based on code intents. The mutations are the result of the mutated intents and/or LLM mistakes in generating code for a given intent. A high-level overview of the functioning of our approach is described in Figure 1.

This approach operates in 4 distinct steps as follows:

1) It masks tokens from the given input intent, creating several masked versions of it – one for each token.
2) Then, it passes those intents to a language model, i.e., BERT, to predict a value for each masked token.
3) Next, it invokes an LLM, i.e. GPT, to generate an implementation for each of those generated intents.
4) In addition, our approach generates mutations, by generating code directly (step 3) using the original intent but asking for multiple, i.e., 10, alternative implementations.

A. Intent masking

In this step, we mask the intent description tokens, one at a time, producing one masked intent per token. This means that every masked version contains the original intent with one missing token, replaced by the placeholder `<mask>`.

This way, we can obtain several mutations from the same intent with small syntactical differences (one token difference). As we aim at introducing behavioural mutations, we exclude tokens that are irrelevant to the text context, i.e. the punctuation characters, and mask only alphanumeric tokens.

For example, for the sentence *function that takes an integer*, μIntMut produces the following masked sequences:

- **<mask>** that takes an integer
- function **<mask>** takes an integer
- function that **<mask>** an integer
- function that takes **<mask>** integer
- function that takes an **<mask>**

B. BERT-MLM prediction

μIntMut invokes BERT [27], a pre-trained language model, to predict replacements for the masked tokens. To do so, it tokenizes every masked version into a tokens vector then crops it to a subset one that fits the maximum size allowed by the model (512). Next, our approach feeds these vectors to BERT-MLM to predict the most probable replacements of the masked token. Our intuition is that the larger the text portion accompanying the mask placeholder, the better BERT would be able to capture the text context, and consequently, the more meaningful its predictions would be. This step ends with the generation of one mutated intent per masked token.

C. LLM code generation

We employ GPT-3.5- with 0.8 temperature, to generate code from intents written in natural language. We run it with the intents from the previous step and generate a code implementation per mutated intent. As those codes have been generated via different intents, we expect them to be different, and thus useful as mutants.

The proposed approach produces also mutants by direct invocation of the LLM with the original code intent. This means that for a given code intent, it asks the LLM to generate different implementations of it, i.e. ten alternative implementations for a given intent.

This way we can generate mutants that represent mistakes of the intent done by the LLM. We thus, have two approaches:

1) Mutated intents (μIntMut): Generates mutants by mutating the original intent and then generating their corresponding implementations.

*2) Original intents (o*IntMut*):* Generates mutants by generating alternative implementations of the original intent.

IV. Experimental study

A. Research questions

Intent-based mutation testing alters the original programming intents with the intention of producing similar but different implementations. To this end, we investigate the extent to which this is possible when using our framework. We thus check whether the intents lead to mutations that are syntactically and semantically different from the original program. An additional aspect we consider is the differences among the mutations themselves, as it is not useful to produce variants that are not diverse. Therefore, we ask:

RQ1 (Diversity): *Does intent-based mutation testing generate valid mutations that are syntactically and semantically diverse?*

We answer this question by measuring the syntactic and semantic similarity of the intent-based mutations with the original programs. We also check the subsumption relationships among these mutations to study the semantic overlaps between those mutations, as well as the number of mutations that are subsuming, i.e., a typical mutation testing diversity metric [22], [28].

Having answered this question and showed that the intent mutations are actually valid and diverse, we contrast them with the mutations generated by other (syntactic-based) approaches to check whether they actually reflect different types of faults than the syntactic methods. Hence, we ask:

RQ2 (Overlap with syntactic mutations): *Are intent-based mutations semantically different from those mutants generated by syntactic-based mutations?*

To answer this question, we compare the semantic overlaps between the intent-based mutations with those produced by syntactic-based methods. We further strengthen our comparison; we also investigate the extent to which intent-based mutations subsume (and are subsumed by) syntactic mutations.

The above investigations aim at checking whether intent-based mutations are actually different from the syntactic ones, which leaves out the question of which approach is most effective and to what extent. Thus, we ask:

RQ3 (Effectiveness): *How effective is intent-based mutation testing in comparison to syntactic-based one?*

To answer this question, we form mutation-based test suites, with respect to the compared techniques, and check their ability to kill a reference set of subsuming mutants, i.e., the subsuming mutants of all the compared techniques together.

B. Experimental setup

HumanEval [29] is a benchmark created to evaluate the models' ability to generate code. It contains 164 python programming problems written by humans, each paired with a solution and test cases. *HumanEval-x* [30] is an extension of HumanEval that contains the same 164 problems but includes four additional programming languages: C++, Java, JavaScript, and Go. *HumanEval+* [31] is another extension of HumenEval that uses mutation testing to augment the programs' test suites.

We use the Java entries from HumanEval-x, specifically, we use the problem descriptions in natural language (English) reflecting the intentions to generate the mutants following the intent-based mutation we introduce above. To ensure the thoroughness of our analysis, we augment the Java entries using test cases from HumanEval+. We extract test data and expected results from the Python test code and translate them to Java by adapting data types and dropping incompatible test data. We construct Java test cases, compile, and execute them against the ground truth solution provided by HumanEval-x, keeping only the problems for which the entire test suite pass.

Unfortunately, our test subjects are small and result in very few mutations for many cases. We thus, need to ensure a reasonable number of mutations for each problem we use, and set a minimum threshold of five killable mutants. This means that we select from the dataset the problems for which each approach produces at least five killable mutants, resulting in a total of 29 problems. Table I records the summary statistics of the number of tests and the length of the description in terms of characters number. It includes the average, median, maximum, and minimum values for each.

TABLE I: Descriptive statistics of the problems we consider

Statistic	Mean	Median	Max	Min
Number of Tests	659.2	865	1025	69
Length of Description	629.55	547	1462	291

C. Experimental procedure

To address **RQ1**, we generate mutants using μIntMut on the dataset problems. We start by mutating the problem description by tokenizing it, masking one token at a time, and using BERT to predict a replacement for it. We prompt GPT-3.5-turbo using each of the mutated descriptions. Additionally, we generate mutants by directly prompting GPT-3.5-turbo to produce different implementations based on the same description. Finally, we mutate the original programs in the dataset using μBert.

The goal is to investigate whether the obtained implementations translate into mutants. To achieve this, we study the syntactic validity and examine how syntactically different they are from the original program by computing the syntactic distance between them. This step includes the computation of several metrics: BLEU score, the number of distinct tokens, cosine similarity, and Jaccard similarity to compute the distance, which is the difference 1 - similarity.

Furthermore, we address semantic diversity based on test execution and subsumption relations between mutants. We run the mutants obtained from μIntMut against the test cases and identify the minimal set of mutants that subsume all the others. To quantify this, we calculate the percentage of subsuming mutants relative to the total number of killed mutants. A higher percentage reflects broader differences in mutants behavior, suggesting a more diverse set.

To answer **RQ2**, we begin by comparing the different approaches semantically using the results of test failure (assertions violation) by identifying the mutants killed by identical sets of tests and illustrate the overlap. We then merge all the

killable mutants generated from the three approaches into one large set of mutants. From this set, we identify the minimal subset of subsuming mutants and compute the contribution of each approach to this set. This analysis shows the relative importance of the mutants of one approach against the others.

To answer **RQ3** and determine whether μINTMUT is more effective than the remaining approaches, we gather all the mutants from all approaches and identify their subsuming subset, which is then used to conduct the comparison. The process entails randomly selecting the minimal test sets that kill all mutants generated by μINTMUT, *o*INTMUT and μBERT one at a time, then using these test sets to kill the mutants of the merged subsuming subset. Since the tests are selected randomly, we repeat this process 100 times to reduce the randomness impact on our results.

Test selection algorithm: we start with an empty set and each time, we add a randomly chosen test and check if the mutation score increases (more mutants are killed). If not, we drop the test as it is redundant with respect to the already selected tests. We continue this process until all killable mutants from an approach are killed.

For comparison, we use an objective score (subsuming mutation score), which corresponds to the number of subsuming mutants in the merged set killed by the tests selected according to one approach, divided by the total number of subsuming mutants, and we compare the three approaches based on it.

V. RESULTS

A. RQ1: Does intent-based mutation testing generate valid mutations that are syntactically and semantically diverse?

Syntactic validity: The mutant generation results reported in Table II show that the three compared approaches produce valid (compilable) mutants. In fact, the majority (over 80%) of the implementations obtained by intent-based mutation are valid (91% by μINTMUT and 81% by *o*INTMUT), which is about three times the generation validity ratio of μBERT (27.6%). This relatively high ratio encourages the approach's design to rely on instruction-based code generation LLMs, i.e. GPT-3.5-turbo, to generate mutants as most of the generation effort results in syntactically valid code.

Moreover, from the last column (total killed) of Table II, we can see that a large ratio of the generated implementations by the proposed approach behave differently from the original code (failing tests that pass on the original code). This confirms that the proposed approach can generate programs that behave differently from the original one and are thereby useful for mutation testing.

TABLE II: Number of generated, valid, killed and alive mutants for each approach

	Total Generated	Total Valid	Total Alive	Total Killed
μINTMUT	2357	2144	620	1524
*o*INTMUT	290	235	24	211
μBERT	3608	996	117	879

Syntactic distance: Table III records the mean syntactic distance of mutants produced by the approaches from the original code. We observe that the intent-based ones are significantly more distant than those obtained by μBERT. For instance, intent-based mutants are about 0.7 bleu distant from the original code, which is over 23 times the bleu distance of μBERT mutants that is 0.031. The same difference is observed when comparing the cosine and jaccard distances; 0.003 and 0.021 for μBERT and 0.2 and 0.3 for intent-based mutants. This highlights the fact that intent-based mutants are more complex than those produced by μBERT, introducing several changes to the original code.

TABLE III: Mean syntactic distance between the mutated and original code

Metric	1 - BLEU	Tokens Diff.	1 - cosine	1 - jaccard
μINTMUT	0.672	68.91	0.185	0.284
*o*INTMUT	0.724	63	0.218	0.274
μBERT	0.031	1.296	0.003	0.021

Semantic Distance: Table IV lists the average semantic distances between the mutants and the original code. This metric reflects the number of tests each mutant fails over the total number of tests.

The table shows that all approaches produce mutants that are semantically distant from the original code. From the first columns, we can see that μBERT *scores the highest average distance of 0.336 compared to the 0.094 and 0.153 scored by respectively* μINTMUT *and* *o*INTMUT. This can be explained by the fact that μBERT produces a higher ratio of killable/valid mutants, as illustrated in Table II). In fact, when considering only the killed mutants, we see that all approaches have relatively similar average distance from the original program, 0.4 and 0.46 for those generated by μBERT intent-based mutations. This validates our approach ability to generate mutants; code implementations that are behaving differently (semantically different) from the original code, and thus can serve for mutation testing.

So far, our results show that our approach is capable of producing syntactically valid, complex, and semantically different mutations from the original code. In the following part, we investigate how diverse these mutants are, that is, how different (semantically) each mutant is from the others.

TABLE IV: Mean semantic distance between the mutations of each approach and the original program

Metric	Valid Mutants	Killable Mutants
μINTMUT	0.153	0.467
*o*INTMUT	0.094	0.46
μBERT	0.336	0.4

Semantic diversity: We conduct a subsumption analysis [32] among the mutations generated by our approach. The plot of Figure 2 shows the percentage of subsuming and subsumed mutants generated by μINTMUT. We find that the subsuming mutants form on average 60.2% and 76.29% of the mutants killed generated by μINTMUT and *o*INTMUT, respectively.

These ratios are relatively high compared to the subsumed mutants of μBERT that are 42.54%. This endorses the diversity of the generated mutants by the proposed approach.

When Computing the minimal subsuming mutants set by removing those that break the same test sets, the average proportion of minimal subsuming mutants is 11% and 24.74% of killed mutants generated by μINTMUT and oINTMUT respectively. This indicates considerable ratio of subsuming mutants being mutually subsumed, with an average of 71% and 60.58% of all subsuming mutants for μINTMUT and oINTMUT respectively.

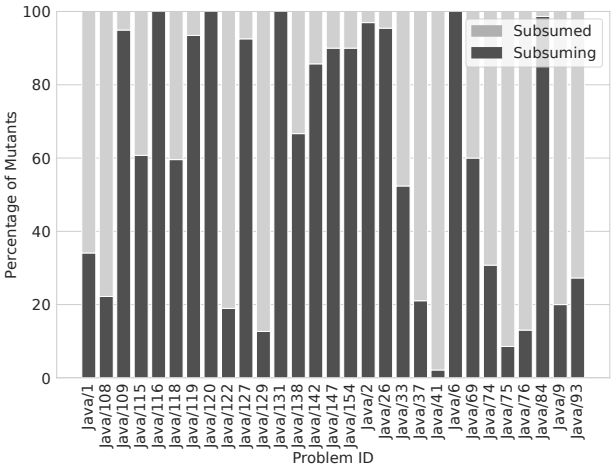

Fig. 2: Percentages of subsuming (dark grey) and subsumed (light grey) mutants produced by μINTMUT

B. RQ2: Are intent-based mutations semantically different from the syntactic-based ones?

Semantic Overlap: To study the diversity and overlap of mutants generated by each approach, we consider the differences of the killing tests between the mutant we study. This means that a mutation is unique if it is killed by a test set that is not killing any other mutant. While mutants that are killed by the same set of tests are considered an overlap. To illustrate the overlap, we begin by removing all the semantic duplicates in each approach before computing the overlap. Figure 3 illustrates a Venn diagram of the unique mutants generated by each approach.

From the diagram, we observe that oINTMUT has the lowest rate of semantically unique mutants, with 24.6% of its killable mutants being detected by a unique set of tests, compared to 64.4% and 83.3% of unique mutants in μINTMUT and μBERT respectively. This shows that intent-based mutation produces different types of faults compared to syntax-based mutation.

Subsumption: To study the complementarity and subsumption between the different approaches, we merge their generated mutants and compute their subsuming set. Then we compute the percentage of subsuming mutants provided by each approach within this set. We plot the obtained results for every task from our dataset in Figure 4. The boxplots show that the three approaches contribute to forming the subsuming set with

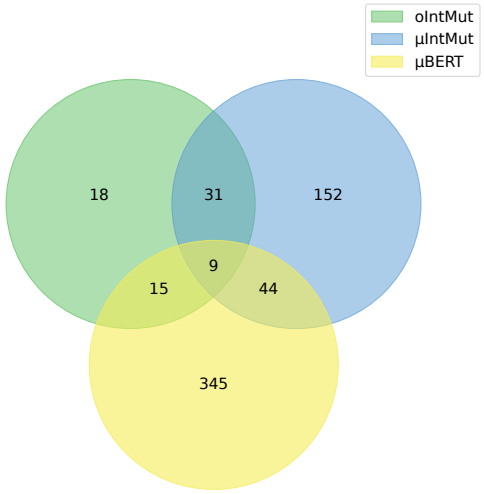

Fig. 3: Semantic overlap between the studied approaches. The majority of Intent-based mutations are not covered by traditional techniques, showing that it produces different types of faults compared to syntax-based mutation approaches.

ratios varying between 0% to 100% for μINTMUT and μBERT, and between 0% to 20% for oINTMUT. This indicates that for some tasks, one approach is subsuming the others while being totally subsumed for other tasks, however no approach subsumes always the other ones.

The boxplots depict also a large advantage for μINTMUT and μBERT over oINTMUT contributing on average by 53.3%, 39.5% and 7.2% of the subsuming mutants respectively. This difference can be explained by the limited number of mutants generated by oINTMUT, which produced 4 and 9 times less mutants than μBERT and μINTMUT, as indicated in Table II.

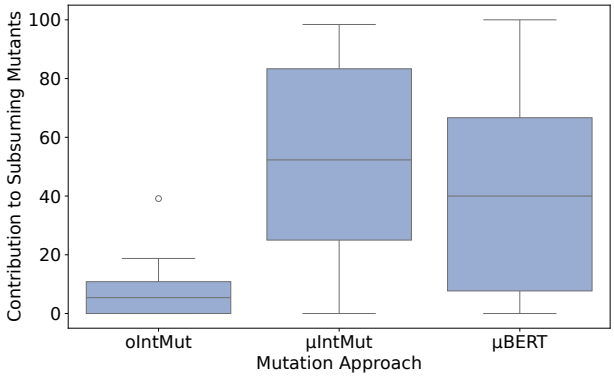

Fig. 4: Subsuming mutants among the different approaches

C. RQ3: How effective is intent-based mutation testing in comparison to syntactic-based one?

To have a common base of comparison between the approaches, we merge all mutants in one set and keep only the subsuming ones. Then we measure the subsuming mutation scores achieved by test suites designed to kill all the mutations

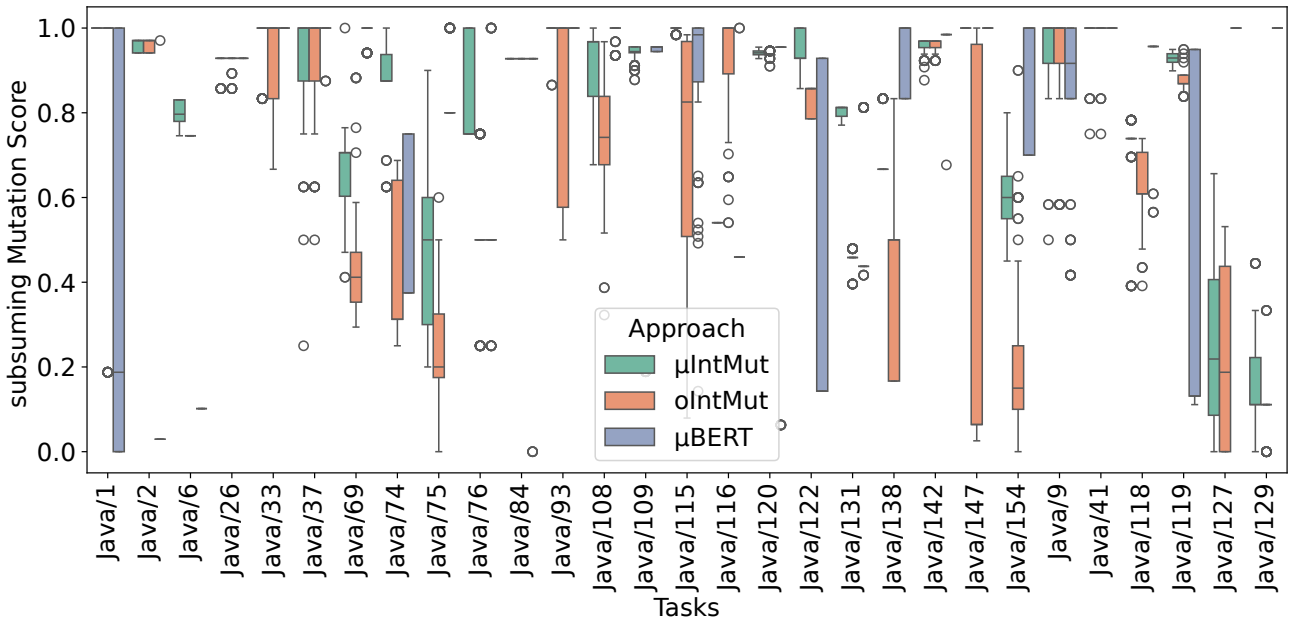

Fig. 5: Distribution of subsuming mutation scores, on the reference set of subsuming mutants, achieved by test suites selected based on each of the three approaches.

of one approach at a time. Figure 5 presents the objective comparison scores (subsuming mutation score) scored by each of the three approaches.

The comparison was conducted 100 times, resulting in 100 distinct scores per case. We illustrate the distributions using boxplots, from which we observe a variance across problems in the percentage of mutants killed by the tests selected from μINTMUT mutants, with some tasks even reaching 100%, and the majority exhibiting higher percentages, unlike oINTMUT and μBERT where we see higher variance and lower values.The score of μINTMUT is 0.817 indicating that tests selected based on μINTMUT kill 81.7% of the subsuming mutants produced by the three approaches (μINTMUT, μBERT and oINTMUT together), compared to 0.77 and 0.66 scored by μBERT and oINTMUT respectively. It is noted that since the reference set of subsuming mutants includes all the mutations produced by all the approaches it is expected that all approaches achieve high scores. The important thing though is that none of the approaches subsumes the others and all are far from being adequate. For instance, μBERT misses 23% of the faults that could be captured when considering all three approaches together. This indicates a large gap, especially by considering that μBERT is a strong approach, arguably as strong as (or stronger) traditional mutation testing [20].

Table V records the number of tasks where the scored subsuming mutation score by each approach is below a certain threshold, μINTMUT has the least number of cases below 50% with only 2, and 17 tasks above 90%. μBERT has the highest number above 90% in 19 cases while oINTMUT has the lowest. These results indicate that a large variance in the effectiveness of all approaches, with μINTMUT having the least low performing cases. μBERT on the other hand has the most cases with the highest scores.

TABLE V: Number of case with reference subsuming mutation score under various thresholds

Approach	50%	60%	70%	80%	90%
μINTMUT	2	4	6	10	12
oINTMUT	9	11	12	14	17
μBERT	8	9	9	9	10

VI. DISCUSSION

Table VI shows examples of mutants produced by μINTMUT. The problem is Java/37 from Humaneval-x. The replaced token and its position within the intent are in blue, green represents the original part, and red denotes the mutated one. For example, in Mutated Intent 1, the word **even** is replaced with **odd** and in Mutated Intent 2, a digit is altered from 6 to 2.

We notice that in mutant 1 the solution begins by adding the values at even indices to the result list, and the values at odd indices in another list, it sorts this list, and then adds its elements to the result list at odd indices. This behavior is the exact opposite of the original intent, which retains the values of odd indices and sorts the ones of even indices.

A. Threats to validity

The generalization of our results forms a threat to the validity of our work. A first concern may be attributed to the ability of Language Models to generatlize to other unseen data. Meaning, that performance may be different on other tasks and

TABLE VI: Mutated intents and related mutations.

```
This function takes a list l and returns
a list l' such that l' is identical
to l in the odd indices, while its values
at the even indices are equal to the values
 of the even indices of l, but sorted.

>>> sortEven(Arrays.asList(1, 2, 3))
[1, 2, 3]
>>> sortEven(Arrays.asList(5, 6, 3, 4))
[3, 6, 5, 4]
```

Resulting Implementation:

```
List<Integer> result = new ArrayList<>(l);
List<Integer> evenIndices = new ArrayList<>();
for (int i = 0; i < l.size(); i++) {
    if (i % 2 == 0) {
        evenIndices.add(l.get(i));
    }
}Collections.sort(evenIndices);
for (int i = 0; i < l.size(); i += 2) {
    result.set(i, evenIndices.remove(0));
}return result;
```

Mutated Intent 1

```
@@ even : 135 - 138 @@
+ values at the even indices are equal to the
- values at the odd indices are equal to the
```

Resulting Mutant 1:

```
List<Integer> result = new ArrayList<>();
List<Integer> evenValues = new ArrayList<>();
for (int i = 0; i < l.size(); i++) {
    if (i % 2 == 0) { result.add(l.get(i)); }
    else { evenValues.add(l.get(i));}
}Collections.sort(evenValues);
int evenIndex = 0;
for (int i = 0; i < l.size(); i++) {
    if (i % 2 != 0) {
        result.add(evenValues.get(evenIndex));
        evenIndex++;
    }
}return result;
```

Mutated Intent 2

```
@@ 6 : 305 - 305 @@
+     >>> sortEven(Arrays.asList(5, 6, 3, 4))
-     >>> sortEven(Arrays.asList(5, 2, 3, 4))
```

Resulting Mutant 2:

```
List<Integer> result = new ArrayList<>(l);
List<Integer> evenValues = new ArrayList<>();
for (int i = 0; i < l.size(); i++) {
    if (i % 2 == 0)
        evenValues.add(l.get(i));
}Collections.sort(evenValues);
int evenIndex = 0;
for (int i = 0; i < l.size(); i++) {
    if (i % 2 == 1) {
        result.set(i, evenValues.get(evenIndex));
        evenIndex++;
    }
}return result;
```

programs. To mitigate those threats we have conducted our experimental study on an independent dataset, specifically built to mitigate this threat in evaluating the performance of Large Language Models on code related tasks. The dataset counts tupples of human written code, description and tests that have not been included in the LLMs training sets. Nevertheless, we acknowledge that the obtained results may not generalise to other cases.

Other threats may arise from the non-deterministic nature of LLMs. This threat does not concern BERT, however concerns our employed code generation LLM. For instance, GPT-3.5-turbo tends to produce different answers for the same question. Although, this may reduce the reproducibility of our study, in a sense where we may obtain other mutants, we do not expect it to have an impact on our general results. Particularely, as we generate multiple mutants for different programs, we believe that the overall outcomes of the study will remain unchanged.

Some threats may arise from our semantic comparison of mutants based on their failing tests. We assume that the tests provided by HumanEval+ are complete and exhaustive, allowing us to capture behavioural differences between different programs. Although this may not be always the case, we believe that these test suites are sufficiently strong for our study. Moreover, we use the same setup, running the same test cases for all mutants generated by the compared approaches, giving a same base of comparison for all approaches.

VII. Conclusion and Future Work

We presented μIntMut, a mutation testing approach that generates mutants based on the program's intent. We proposed two ways to achieve this, by mutating the intent, and by generating different implementations of the intents. Our results revealed that μIntMut produces a set of complex and semantically diverse mutants, which are semantically unique when compared to μBert, a syntax-based approach, reflecting our approach's ability to capture different types of faults than those generated by traditional approaches.

APPENDIX
SUBJECT USED

Java/1 : Input to this function is a string containing multiple groups of nested parentheses. Your goal is to separate those group into separate strings and return the list of those. Separate groups are balanced (each open brace is properly closed) and not nested within each other. Ignore any spaces in the input string.
>>> separateParenGroups(””( ) (( )) (( )( ))””) [””()””, ””(())””, ””(()())””]

Java/2: Given a positive floating point number, it can be decomposed into an integer part (largest integer smaller than the given number) and decimals (leftover part always smaller than 1).
Return the decimal part of the number.
>>> truncateNumber(3.5) 0.5

Java/6 Input to this function is a string represented multiple groups for nested parentheses separated by spaces.For each of the group, output the deepest level of nesting of parentheses.
E.g. (()()) has maximum two levels of nesting while ((())) has three.
>>> parseNestedParens(””(()()) ((())) () ((())()())””) [2, 3, 1, 3]

Java/9 From a given list of integers, generate a list of rolling maximum element found until the given moment in the sequence.
>>> rollingMax(Arrays.asList(1, 2, 3, 2, 3, 4, 2))
[1, 2, 3, 3, 3, 4, 4]

Java/26 From a list of integers, remove all elements that occur more than once. Keep order of elements left the same as in the input.
>>> removeDuplicates(Array.asList(1, 2, 3, 2, 4))
[1, 3, 4]

Java/33 This function takes a list l and returns a list l’ such that l’ is identical to l in the indicies that are not divisible by three, while its values at the indicies that are divisible by three are equal to the values of the corresponding indicies of l, but sorted.
>>> sortThird(Arrays.asList(1, 2, 3)) 1em[1, 2, 3]
>>> sortThird(Arrays.asList(5, 6, 3, 4, 8, 9, 2)) [2, 6, 3, 4, 8, 9, 5]

Java/37 This function takes a list l and returns a list l’ such that l’ is identical to l in the odd indicies, while its values at the even indicies are equal to the values of the even indicies of l, but sorted.
>>> sortEven(Arrays.asList(1, 2, 3))
[1, 2, 3]
>>> sortEven(Arrays.asList(5, 6, 3, 4))
[3, 6, 5, 4]

Java/41 Imagine a road that’s a perfectly straight infinitely long line. n cars are driving left to right; simultaneously, a different set of n cars are driving right to left. The two sets of cars start out being very far from each other. All cars move at the same speed. Two cars are said to collide when a car that’s moving left to right hits a car that’s moving right to left. However, the cars are infinitely sturdy and strong; as a result, they continue moving in their trajectory as if they did not collide. This function outputs the number of such collisions.

Java/69 You are given a non-empty list of positive integers. Return the greatest integer that is greater than zero, and has a frequency greater than or equal to the value of the integer itself. The frequency of an integer is the number of times it appears in the list. If no such a value exists, return -1.
Examples: search(Arrays.asList(4, 1, 2, 2, 3, 1)) == 2
search(Arrays.asList(1, 2, 2, 3, 3, 3, 4, 4, 4)) == 3
search(Arrays.asList(5, 5, 4, 4, 4)) == -1

Java/74 Write a function that accepts two lists of strings and returns the list that has total number of chars in all strings of the list less than the other list. if the two lists have the same number of chars, return the first list.
Examples:
totalMatch(Arrays.asList(), Arrays.asList()) -¿ []
totalMatch(Arrays.asList(”hi”, ”admin”), Arrays.asList(”hI”, ”Hi”))−>[”hI”, ”Hi”]
totalMatch(Arrays.asList(”4”), Arrays.asList(”1”, ”2”, ”3”, ”4”, ”5”))−>[”4”]

Java/75 Write a function that returns true if the given number is the multiplication of 3 prime numbers and false otherwise. Knowing that (a) is less than 100.
Example:
isMultiplyPrime(30) == true
30 = 2 * 3 * 5

Java/76 Your task is to write a function that returns true if a number x is a simple power of n and false in other cases. x is a simple power of n if n**int=x
For example:
isSimplePower(1, 4) => true
isSimplePower(5, 3) => false

Java/84 Given a positive integer N, return the total sum of its digits in binary.
Example:
For N = 1000, the sum of digits will be 1 the output should be ””1””.
For N = 150, the sum of digits will be 6 the output should be ””110””.
For N = 147, the sum of digits will be 12 the output should be ””1100””.
Variables: @N integer
Constraints: 0 <= <= 10000.
Output: a string of binary number

Java/93 Write a function that takes a message, and encodes in such a way that it swaps case of all letters, replaces all vowels in the message with the letter that appears 2 places ahead of that vowel in the english alphabet. Assume only letters.
Examples:
>>> encode(””test””)
””TGST””
>>> encode(””This is a message””)
””tHKS KS C MGSSCGG””

Java/108 Write a function countNums which takes an array of integers and returns the number of elements which has a sum of digits ¿ 0. If a number is negative, then its first signed digit will be negative:
e.g. -123 has signed digits -1, 2, and 3.
>>> countNums(Arrays.asList()) == 0
>>> countNums(Arrays.asList(-1, 11, -11)) == 1
>>> countNums(Arrays.asList(1, 1, 2)) == 3

Java/109 We have an array ’arr’ of N integers arr[1], arr[2], ..., arr[N].The numbers in the array will be randomly ordered. Your task is to determine if it is possible to get an array sorted in non-decreasing order by performing the following operation on the given array:
You are allowed to perform right shift operation any number of times.
One right shift operation means shifting all elements of the array by one position in the right direction. The last element of the array will be moved to the starting position in the array i.e. 0th index.
If it is possible to obtain the sorted array by performing the above operation then return true else return False. If the given array is empty then return true.
Note: The given list is guaranteed to have unique elements.
For Example:
moveOneBall(Arrays.asList(3, 4, 5, 1, 2)) ==>true
Explanation: By performin 2 right shift operations, non-decreasing order can be achieved for the given array.
moveOneBall(Arrays.asList(3, 5, 4, 1, 2)) ==>False
Explanation:It is not possible to get non-decreasing order for the given array by performing any number of right shift operations.”

Java/115 You are given a rectangular grid of wells. Each row represents a single well, and each 1 in a row represents a single unit of water. Each well has a corresponding bucket that can be used to extract water from it, and all buckets have the same capacity.
Your task is to use the buckets to empty the wells.
Output the number of times you need to lower the buckets.
Example 1:
Input: grid : [[0,0,1,0], [0,1,0,0], [1,1,1,1]], bucket_capacity : 1
Output: 6
Example 2:
Input: grid : [[0,0,1,1], [0,0,0,0], [1,1,1,1], [0,1,1,1]], bucket_capacity : 2
Output: 5
Example 3:
Input: grid : [[0,0,0], [0,0,0]], bucket_capacity : 5
Output: 0
Constraints:
* all wells have the same length * 1 <= grid.length <= 10^2
* 1 <= grid[:,1].length <= 10^2 * grid[i][j] −> 0 — 1
* 1 <= capacity <= 10

Java/118 You are given a word. Your task is to find the closest vowel that stands between two consonants from the right side of the word (case sensitive). Vowels in the beginning and ending doesn't count. Return empty string if you didn't find any vowel met the above condition. You may assume that the given string contains English letter only.
Example:
getClosestVowel(""yogurt"") ==> ""u""
getClosestVowel(""FULL"") ==> ""U""
getClosestVowel(""quick"") ==> """"
getClosestVowel(""ab"") ==> """"

Java/119 You are given a list of two strings, both strings consist of open parentheses ""("" or close parentheses "")"" only. Your job is to check if it is possible to concatenate the two strings in some order, that the resulting string will be good. A string S is considered to be good if and only if all parentheses in S are balanced. For example: the string ""(())()"" is good, while the string "")())"" is not. Return ""Yes"" if there""s a way to make a good string, and return ""No"" otherwise.
Examples:
matchParens(Arrays.asList(""()("", "")"")) == ""Yes""
matchParens(Arrays.asList("")"", "")"")) == ""No""

Java/120 Given an array arr of integers and a positive integer k, return a sorted list of length k with the maximum k numbers in arr.
Example 1: Input:arr = [-3, -4, 5], k = 3
Output: [-4, -3, 5]
Example 2: Input: arr = [4, -4, 4], k = 2
Output: [4, 4]
Example 3: Input: arr = [-3, 2, 1, 2, -1, -2, 1], k = 1
Output: [2]
Note: * 0 <= k <= len(arr)
* The length of the array will be in the range of [1, 1000].
* The elements in the array will be in the range of [-1000, 1000].

Java/122 Given a non-empty array of integers arr and an integer k, return the sum of the elements with at most two digits from the first k elements of arr.
Example:
Input: arr = [111,21,3,4000,5,6,7,8,9], k = 4
Output: 24 # sum of 21 + 3
Constraints:
1. 1 <= len(arr) <= 100
2. 1 <= k <= len(arr)

Java/127
You are given two intervals, where each interval is a pair of integers. For example, interval = (start, end) = (1, 2). The given intervals are closed which means that the interval (start, end) includes both start and end. For each given interval, it is assumed that its start is less or equal its end. Your task is to determine whether the length of intersection of these two intervals is a prime number.
Example, the intersection of the intervals (1, 3), (2, 4) is (2, 3) which its length is 1, which not a prime number. If the length of the intersection is a prime number, return ""YES"", otherwise, return ""NO"". If the two intervals don't intersect, return ""NO"".
[input/output] samples:
intersection((1, 2), (2, 3)) ==> ""NO""
intersection((-1, 1), (0, 4)) ==> ""NO""
intersection((-3, -1), (-5, 5)) ==> ""YES""

Java/129 Given a grid with N rows and N columns (N >= 2) and a positive integer k, each cell of the grid contains a value. Every integer in the range [1, N * N] inclusive appears exactly once on the cells of the grid. You have to find the minimum path of length k in the grid. You can start from any cell, and in each step you can move to any of the neighbor cells, in other words, you can go to cells which share an edge with you current cell. Please note that a path of length k means visiting exactly k cells (not necessarily distinct). You CANNOT go off the grid. A path A (of length k) is considered less than a path B (of length k) if after making the ordered lists of the values on the cells that A and B go through (let's call them lst_A and lst_B), lst_A is lexicographically less than lst_B, in other words, there exist an integer index i (1 <= i <= k) such that lst_A[i] < lst_B[i] and for any j (1 <= j < i) we have lst_A[j] = lst_B[j]. It is guaranteed that the answer is unique. Return an ordered list of the values on the cells that the minimum path go through.
Examples:
Input: grid = [[1,2,3], [4,5,6], [7,8,9]], k = 3
Output: [1, 2, 1]
Input: grid = [[5,9,3], [4,1,6], [7,8,2]], k = 1
Output: [1]

Java/131 Given a positive integer n, return the product of the odd digits. Return 0 if all digits are even.
For example:
digits(1) == 1 digits(4) == 0 digits(235) == 15

Java/138 Evaluate whether the given number n can be written as the sum of exactly 4 positive even numbers
Example : isEqualToSumEven(4) == false isEqualToSumEven(6) == false
isEqualToSumEven(8) == true

Java/142 This function will take a list of integers. For all entries in the list, the function shall square the integer entry if its index is a multiple of 3 and will cube the integer entry if its index is a multiple of 4 and not a multiple of 3. The function will not change the entries in the list whose indexes are not a multiple of 3 or 4. The function shall then return the sum of all entries.
Examples:
For lst = [1,2,3] the output should be 6
For lst = [] the output should be 0
For lst = [-1,-5,2,-1,-5] the output should be -126

Java/147 You are given positive integer n. You have to create integer array a of length n.
For each i (1 <= i <= n), the value of a[i] = i * i - i + 1.
Return the number of triples (a[i], a[j], a[k]) of a where i < j < k,
and a[i] + a[j] + a[k] is a multiple of 3.
Example :
Input: n = 5 Output: 1
Explanation: a = [1, 3, 7, 13, 21]
The only valid triple is (1, 7, 13).

Java/154 You are given 2 words. You need to return true if the second word or any of its rotations is a substring in the first word
cycpatternCheck(""abcd"",""abd"") => false
cycpatternCheck(""hello"",""ell"") => true
cycpatternCheck(""whassup"",""psus"") => false
cycpatternCheck(""abab"",""baa"") => true
cycpatternCheck(""efef"",""eeff"") => false
cycpatternCheck(""himenss"",""simen"") => true

REFERENCES

[1] 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.

[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] T. T. Chekam, M. Papadakis, T. F. Bissyandé, Y. L. Traon, and K. Sen, "Selecting fault revealing mutants," *Empirical Software Engineering*, vol. 25, no. 1, pp. 434–487, 2020.

[4] S. J. Kaufman, R. Featherman, J. Alvin, B. Kurtz, P. Ammann, and R. Just, "Prioritizing mutants to guide mutation testing," in *International Conference on Software Engineering*, 2022, p. 1743–1754.

[5] M. Tufano, J. Kimko, S. Wang, C. Watson, G. Bavota, M. Di Penta, and D. Poshyvanyk, "Deepmutation: A neural mutation tool," in *International Conference on Software Engineering: Companion Proceedings*, ser. ICSE, 2020, p. 29–32.

[6] 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.

[7] M. Papadakis, Y. Jia, M. Harman, and Y. L. Traon, "Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique," in *37th IEEE/ACM International Conference on Software Engineering, ICSE*, 2015, pp. 936–946.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

[15] Z. Tian, J. Chen, Q. Zhu, J. Yang, and L. Zhang, "Learning to construct better mutation faults," in *Proceedings of the International Conference on Automated Software Engineering*, 2022, pp. 1–13.

[16] 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.

[17] 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.

[18] K. Herzig and A. Zeller, "Untangling changes," *Unpublished manuscript, September*, vol. 37, pp. 38–40, 2011.

[19] 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.

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

[21] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," in *Conference on Empirical Methods in Natural Language Processing: Findings, EMNLP*, 2020, pp. 1536–1547.

[22] 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.

[23] 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.

[24] "Github copilot," https://github.com/features/copilot.

[25] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code.(2021)," *arXiv:2107.03374*, 2021.

[26] "Amazon codewhisperer," https://aws.amazon.com/codewhisperer/.

[27] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv:1810.04805*, 2018.

[28] M. Papadakis, T. T. Chekam, and Y. L. Traon, "Mutant quality indicators," in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops*, 2018, pp. 32–39.

[29] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv:2107.03374*, 2021.

[30] Q. Zheng, X. Xia, X. Zou, Y. Dong, S. Wang, Y. Xue, L. Shen, Z. Wang, A. Wang, Y. Li *et al.*, "Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x," in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2023, pp. 5673–5684.

[31] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[32] B. Kurtz, P. Ammann, M. E. Delamaro, J. Offutt, and L. Deng, "Mutant subsumption graphs," in *International Conference on Software Testing, Verification, and Validation Workshops ICSTW*, 2014, p. 176–185.