Syntactic Versus Semantic Similarity of Artificial and Real Faults in Mutation Testing Studies

Milos Ojdanic, Aayush Garg, Ahmed Khanfir, Renzo Degiovanni, *Member, IEEE*, Mike Papadakis, and Yves Le Traon

***Abstract*—Fault seeding is typically used in empirical studies to evaluate and compare test techniques. Central to these techniques lies the hypothesis that artificially seeded faults involve some form of realistic properties and thus provide realistic experimental results. In an attempt to strengthen realism, a recent line of research uses machine learning techniques, such as deep learning and Natural Language Processing, to seed faults that look like (syntactically) real ones, implying that fault realism is related to syntactic similarity. This raises the question of whether seeding syntactically similar faults indeed results in semantically similar faults and, more generally whether syntactically dissimilar faults are far away (semantically) from the real ones. We answer this question by employing 4 state-of-the-art fault-seeding techniques (PiTest - a popular mutation testing tool, IBIR - a tool with manually crafted fault patterns, DeepMutation - a learning-based fault seeded framework and μBERT - a mutation testing tool based on the pre-trained language model CodeBERT) that operate in a fundamentally different way, and demonstrate that syntactic similarity does not reflect semantic similarity. We also show that 65.11%, 76.44%, 61.39% and 9.76% of the real faults of Defects4J V2 are semantically resembled by PiTest, IBIR, μBERT and DeepMutation faults, respectively.**

***Index Terms*—Fault injection, fault seeding, machine learning, mutation testing, semantic model, syntactic distance.**

I. Introduction

FAULT seeding techniques, such as mutation testing, are extensively used in controlled studies to evaluate and compare testing techniques [41], [42]. These techniques allow researchers to seed faults under experimentally controlled conditions and thus perform reproducible test assessments. In a sense, by comparing the number of seeded faults revealed by test methods, researchers can form a proxy metric that approximates the fault-revealing potential of the performed testing [9], [30], [43].

Although popular, such techniques have been criticised for producing unrealistic faults [12], [23], [45], [55], i.e., faults that are significantly different from real ones in terms of syntax [23], and as a result, numerous propositions have been made claiming to produce seeded faults that are syntactically similar to real ones. The most recent research, in particular, motivated by the code naturalness hypothesis [25],[1] aims at forming realistic faults that are, in fact, artificial faults that have some form of syntactic similarity to real ones, i.e., usually following particular syntactic fault patterns. We call this line of work as *fault mimicking* approaches.

Table I lists a set of recent fault-mimicking techniques that aim, in diverse forms, at generating (syntactically) realistic faults. By inspecting the table, the research trend becomes evident as these techniques seek the realism of fault-seeding, which is defined and evaluated by some form of non-semantic metrics, i.e., mainly syntactic-based metrics (number of tokens changed, BLEU score, etc.) from real faults. This means that many studies are solely guided by syntactic metrics and not semantic ones. Nevertheless, such approaches may indeed succeed at generating some exact matches of targeted real faults and may indeed be effective in their domain. However, since those fault mimicking methods are guided by non-semantic metrics, a key question that remains is whether they are suitable for fault-based test assessment [42], as is the typical use of mutation testing in research studies [9], [41], [42].

Mutation testing is based on the basis that fault seeding should be performed using untargeted program syntactic changes [18], [37], [38]. These changes are defined using the programming language grammar and are completely unaware of any fault semantics. The key assumption is that simple syntactic changes, although syntactically dissimilar to complex faults, result in semantic deviations that are coupled with complex and real faults[2] [15], [38] and can be used for test assessment [9].

In contrast, the key strategy followed by fault mimicking is to identify program locations where fault opportunities emerge and perform relevant changes, following a pattern observed in some fault instances, that alter the program behaviours similarly to real faults. This implies an underlying assumption that *seeding faults with frequent syntactic fault patterns that have similarities with a real fault will result in faults that are subtle or semantically similar to real ones*. Similarly, another assumption is that *seeding faults that are syntactically dissimilar to real ones results in*

Manuscript received 18 June 2022; revised 8 May 2023; accepted 12 May 2023. Date of publication 26 May 2023; date of current version 18 July 2023. This work was supported by the Luxembourg National Research Fund (FNR) through the CORE project under Grant C19/IS/13646587/RASoRS and PayPal. Recommended for acceptance by L. Mariani. *(Corresponding author: Milos Ojdanic.)*

The authors are with the University of Luxembourg L-1359, Luxembourg (e-mail: milos.ojdanic@uni.lu; aayush.garg@uni.lu; ahmed.khanfir@uni.lu; renzo.degiovanni@uni.lu; michail.papadakis@uni.lu; yves.letraon@uni.lu).

Digital Object Identifier 10.1109/TSE.2023.3277564

[1]Naturalness hypothesis states that programs exhibit properties similar to text and thus, natural language process techniques can be used to support code analysis techniques.

[2]In this paper we use the term "real faults" to refer to the set of reproducible curated faults provided at the Defects4J dataset [29]. Therefore, our results reflect the similarities between the artificial faults and their corresponding curated fault.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

TABLE I
FAULT MIMICKING TECHNIQUES

Approach	Year	Aim	Evaluation metric	Venue
Bug Creation for Neural Bug Detectors [51]	2022	Derive contextual mutation operators to inject more realistic faults	Syntactic match	ICST
A Study of Codex, Pre-Trained Language Model on Code [10]	2022	Evaluate code manipulation and code generation tasks such as code mutation	Exact syntactic match and manual analysis	arXiv
Self-Supervised Bug Detection and Repair [8]	2021	Produce and detect hard-to-detect faults	Syntactic match	NeurIPS
SemSeed: Token Embeddings [46]	2021	Derive syntactic patterns that are syntactically similar to real faults	Exact syntactic match	ESEC/FSE
Mutation Monkey [11]	2021	Deriving common fault syntactic patterns	Detection Ratio	ICSE
A SBST Framework of Source Code Embedding [47]	2021	Generate adversarial code snippets that can fool a downstream task	Number of tokens changed	ICST
DeepMutation: Learning-based Mutations [54]	2020	Produce mutants syntactically similar to real faults	Syntactic distance from real faults	ICSE
Learning-based Mutations [56]	2019	Derive syntactic patterns from bug-fixes	Syntactic distance from real faults	ICSME
Wild-Caught Mutations [12]	2017	Deriving simple syntactic patterns from bug-fixes	Token similarity, Compilability	ESEC/FSE
Analysis of real faults and mutants [23]	2014	Syntactic similarities of bug-fixes and mutants	Number of tokens changed	ISSRE

unrealistic faulty semantics, i.e., the seeded fault semantics are quite different from those of real faults.

These assumptions may appear intuitive but have absence of evidence, except of course, in the case where seeded faults match exactly real ones. Early research on the coupling effect [38] stated that "simple faults can cascade or couple to form other emergent faults", implying that fault instances couple independently of their pattern. Additionally, recent studies report large semantic overlaps between simple and complex faults [28], [35], [41], questioning the role of the syntactic-based metrics.

This raises the question of whether syntactically similar, or dissimilar, faults are also semantically similar, or dissimilar. More generally, a question of whether the use of such techniques results in faults that: *a) are semantically similar to real faults, b) resemble (semantically) more faults than the dissimilar ones, and c) are subsumed by simple untargeted syntactic deviations as done by mutation testing, i.e., whether they form a useful addition to mutation testing*.

We answer the above questions by employing four fundamentally different fault-seeding techniques. These include PiTest [16], a popular mutation testing tool [33], that uses simple syntactic patterns, IBIR [31], a mutation testing tool with manually crafted fault patterns, DeepMutation [53], a deep learning-based tool that derives patterns from real bug-fixes [55], and μBERT [17], a mutation testing tool that uses a pre-trained language model (CodeBERT [19]). Hence, we investigate the ability of all faults produced by these techniques to form similar semantic deviations as the real faults of Defects4J V2 [29] and check their potential utility within mutation-based test assessment.

Perhaps surprisingly, our results show that syntactic similarity does not reflect semantic similarity, indicating that syntactic distance cannot be used as an evaluation metric in the context of mutation testing. Additionally, our results show that the real faults of Defects4J V2 can be semantically resembled and subsumed by μBERT, PiTest, IBIR and DeepMutation faults, respectively.

Moreover, we also show that simple faults introduced by IBIR subsume almost all faults introduced by other tools, being complemented in $\approx 2\%$ by PiTest and μBERT. Furthermore, when controlling the number of seeded faults, we find that μBERT resembles similar number of real faults as PiTest, while IBIR keeps a significantly higher ratio compared with the rest of the tools in $\approx$10%. Additionally, we find that other techniques probably subsume DeepMutation, whose technique produces significantly fewer mutants which are, at the same time, easier to kill (on average, DeepMutation's mutants are killed by 10 more tests compared to other tools).

Overall, our work aims at raising awareness on the use of semantic and syntactic evaluation metrics in fault seeding studies. Our key contribution regards the use of non-semantic metrics, where we expose and refute the use of syntactic metrics and provide evidence related to the utility of recent fault seeding advances in the test assessment context. Our findings also significantly improve our understanding on the role of the faults' syntactic nature with respect to program semantics and the use of the semantic-based metrics in the context of fault-based test assessment.

II. Syntactic and Semantic Similarity of Artificial and Real Faults

Mutation seeds artificial faults, called *mutants*, by performing simple syntactic modifications to the program under analysis [42]. For instance, in expressions like `a < b`, faults are seeded by mutating the expression to the following one `a < b`. Mutant faults are then used to assess the effectiveness and thoroughness of a test suite in detecting these artificial faults. A test case that detects a mutant fault, i.e., that is capable of producing distinguishable observable outputs between the mutant and the original program, is said to be able to *kill* the mutant. A mutant is said to be *killed* if it is detected by a test case or a test suite; otherwise, it is called *live* or *survived*. Test adequacy is called *mutation score* and is computed as the ratio of killed mutants over the total number of generated mutants.

Two types of metrics are usually used to evaluate fault seeding: syntactic and semantic similarity. Table I lists studies using predominantly syntactic similarity, while several studies have used some form of semantic similarity [28], [30], [31], [43]. Intuitively, *syntactic similarity* refers to the distance between the text representations of the mutant and the real faulty code,

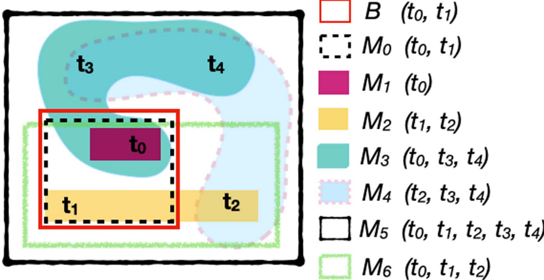


Fig. 1. Semantic similarity. Mutant M_0 perfectly resembles (semantically) fault B, while M_1, M_2 and M_6 resemble it partially. M_6 resembles B better than M_2 and M_1 since it correctly detected 2 out of 3 cases that detect either B or $M6$, while M_2 detected 1 out of 3 cases and M_1 1 out of 2. M_1 underestimates test effectiveness as it does not capture t_1, M_2 does not capture t_0 and overestimates effectiveness as it mistakenly captures t_2, while M_6 mistakenly captures t_2.

while *semantic similarity* to the program *behaviour* similarities between the mutant and the real fault.

To compute the syntactic similarity between two sequences of tokens, we employ the Bilingual Evaluation Understudy (BLEU) score [44], which is widely used for quantifying machine-translated text in NLP [27], [34], [49], [51]. Given a text of reference, the BLEU score takes the candidate text, breaks it into n-grams, and computes how many n-grams appear in the reference text. We report the geometric mean of all n-grams up to 4, similar to previous work [55].

To compute the semantic similarity we resort to dynamic test executions since capturing all program behaviours is an undecidable problem. We, thus, use a similarity coefficient, the Ochiai coefficient, to compute the similarity of the passing and failing test cases. This is a common practice in many different lines of work, such as mutation testing [28], [31], [43] program repair [24] and code analysis [22] studies. Since semantic similarity compares the behaviour between two program versions using a reference test suite, Ochiai coefficient [37] approximates program semantics using passing and failing test cases.

Intuitively, the Ochiai coefficient represents the ratio between the set of tests that fail in both versions over the total number of tests that fail in sum of the two. Precisely, let P_1, P_2, fTS_1 and fTS_2 be two programs and their respective set of failing tests, then the Ochiai coefficient between programs P_1 and P_2 is computed as $Ochiai(P_1,P_2) = \frac{|fTS_1 \cap fTS_2|}{\sqrt{|fTS_1| \times |fTS_2|}}$, where $|\cdot|$ denotes the set size.

Fault Resemblance: A mutant M *resembles* fault B, if and only if its semantic similarity is equal to 1, i.e., $Ochiai(B,M) = 1$.

Example of Semantic Similarity: Let B be a real fault, $M = \{M_0,\ldots,M_6\}$ a set of mutants and $T = \{t_0,\ldots,t_4\}$ a set of test cases. Fig. 1 depicts the mutant killings of T and M. We observe that tests t_0 and t_1 detect fault B. Particularly, mutant M_0 is killed by the same tests, t_0 and t_1, resulting in a semantic similarity with the fault B equal to 1. Mutant M_1, is killed by test t_0 that also finds fault B, but is not killed by test t_1, and thus its semantic similarity is $Ochiai(B,M_1) = |\{t_0\}|/\sqrt{|\{t_0,t_1\}| \times |\{t_0\}|} = 1/\sqrt{2\times 1} = 0.71$. Mutant M_2 is killed by tests t_1 and t_2, so its semantic similarity is $Ochiai(B,M_2) = 1/\sqrt{2\times 2} = 0.50$.

TABLE II
SEMANTIC SIMILARITY BETWEEN THE REAL FAULT AND THE MUTANTS CAPTURED FROM FIG. 1. MUTANT M_0 PERFECTLY RESEMBLES (SEMANTICALLY) FAULT B

	Tests	t_0	t_1	t_2	t_3	t_4	$Ochiai(B, M_i)$
	B	✓	✓				-
Seeded Faults	M_0	✓	✓				1.00
	M_1	✓					0.70
	M_2		✓	✓			0.50
	M_3	✓			✓	✓	0.40
	M_4			✓	✓	✓	0.00
	M_5	✓	✓	✓	✓	✓	0.63
	M_6	✓	✓	✓			0.82

The semantic similarity between mutant M_3 and the fault B is 0.40 ($Ochiai(B,M_3) = 1/\sqrt{2\times 3} = 0.40$). Semantic similarity of M_4 (w.r.t. fault B) is 0, since all tests killing mutant M_4 do not detect the fault. Mutant M_5 is killed by all tests ($t_0,\ldots,t_4$) and has a semantic similarity of $Ochiai(B,M_5) = 2/\sqrt{2\times 5} = 0.63$. Notice that mutant M_6 is killed by tests t_0, t_1 and t_2, where 2 of them also find the fault, leading to a semantic similarity of $Ochiai(B,M_6) = 2/\sqrt{2\times 3} = 0.82$. Table II summarises the Ochiai coefficient between the mutants and the real fault B.

III. MOTIVATING EXAMPLE

We demonstrate the potential differences between syntactic and semantic deviations in fault seeding by using an example from the work of Tufano et al. [55]. Consider the following example[3]:

```
//Original (abstracted) code in abstract representation
     (representation used by Tufano et al.)
public TYPE_1 remove ( int index ) {
  TYPE_2 < TYPE_1 > VAR_1 = this . VAR_2 . remove ( index
       ) ;
  return null != VAR_1 ? VAR_1 . get ( ) : null ; }
```

In this example, the `remove` method first accesses one of the attributes of the invoking object (`this.VAR_2`) and invokes the method `remove` recursively, saving the result in variable `VAR_1`. Then, it returns `null` in the case that `VAR_1` was `null`, otherwise, it returns the result of invoking `VAR_1.get()`. Tufano et al. in their work seed a fault that resembles exactly the real faulty instance, which is the following:

```
//Successful fault seeding by Tufano et al. (the fault
     resembles exactly the real fault)
public TYPE_1 remove ( int index ) {
  return this . VAR_2 . remove ( index ) . get ( ) ; }
```

The fault is caused because of the conditional check that is skipped and, indeed, resembles a real fault made by developers [55]. In particular, this fault removes the check on whether the result of the recursive call is `null`.

[3]This example was taken from [55, Figure 2] and demonstrates a successful case where the fault seeded by Tufano et al. matches exactly a real fault.

Consider now a particular fault seeded by "traditional" mutation testing, using simple syntactic changes (e.g., generated by the REMOVE_CONDITIONALS[4] operator from PiTest [16]):

```
//Fault seeded using mutation testing, simple syntax-based
    mutation
public TYPE_1 remove ( int index ) {
  TYPE_2 < TYPE_1 > VAR_1 = this . VAR_2 . remove ( index
      ) ;
  return true ? VAR_1 . get ( ) : null ; }
```

This mutant replaces the condition `null != VAR_1` by `true`, causing the guarded statements (i.e., `VAR_1.get()`) to be executed irrespective of the condition.

Interestingly, by comparing the two faulty instances, one can easily observe that they are syntactically different despite being semantically equivalent. One can also observe that a simple syntactic transformation, such as the one used by mutation testing, perfectly matches the complex transformation learned by Tufano et al. To make the differences concrete, we can compute the BLEU scores (syntactic similarity between seeded and real fault), i.e., the evaluation metric used by Tufano et al. [55], and see that the returned scores are *1* and *0.48*, respectively. However, as the mutants are equivalent and resemble a real fault, their semantic similarity is 1 despite the large difference in the BLEU scores.

The above example clearly shows that seeded faults do not necessarily need to be similar to real faults in order to resemble them. At the same time, the above example demonstrates the *fault coupling* [38], i.e., simple syntactic transformations, such as those used by mutation testing, couple to more complex faults. In this particular case, the transformation performed by mutation is significantly smaller than Tufano et al. as it has a BLEU score (syntactic similarity from the original code) of 0.85, while Tufano et al. has 0.39.

IV. Research Questions

We start our analysis by recording the syntactic and semantic similarity between seeded and real faults. We perform this analysis to understand the general relation between seeded and real faults and check if there are any associations between these two variables. The existence of such a relationship will provide evidence that fault seeding techniques, instead of using grammar-based (simple) transformations as is traditionally done in mutation testing, should attempt to form frequent fault patterns and design fault seeding techniques guided by actual fault instances, in a sense follow a similar path to static code analysis [26], [47]. Therefore, we ask:

RQ1 *How semantically and syntactically similar are seeded and real faults?*

The answer to this question will provide evidence on the use of syntactic distance metrics in evaluating fault seeding methods in the context of mutation-based test assessment. More precisely, whether seeded faults with small (or big) syntactic distance from the actual faults are indeed semantically close (or far) to actual faults (at least closer than those not syntactically similar).

Syntactic evaluation metrics are used by recent research (Table I), and there is no empirical evidence of their suitability in test assessment. This means that we want to check whether the techniques of Table I could be used in mutation testing studies and whether syntactic distance metrics are appropriate in this context.

Answering the above question aims to investigate general trends among seeded and real faults. However, it does not say much about the extent to which real faults are resembled by seeded ones and does not provide quantitative evidence on the real faults that can be resembled (have high semantic similarity) by syntactically close and far seeded faults. Thus, we ask:

RQ2 *How many real faults we can (semantically) resemble by using syntactically similar and dissimilar seeded faults?*

In case we find many syntactically similar seeded faults being semantically similar to real ones, we have evidence that syntactic distance actually leads to "True Positives" and may be used in mutation testing. On the contrary, if we find many syntactically similar seeded faults that are semantically dissimilar to real ones, we have evidence that syntactic distance leads to many "False Positives". Similarly, if we find many syntactically dissimilar seeded faults that are semantically similar to real ones, then we have evidence that fault-mimicking techniques produce many "False Negatives". By putting all cases together, we have evidence on how effective fault-mimicking techniques are.

While we investigated the relationship between seeded and real faults, we have not said much about how the faults from different seeding techniques differ, w.r.t., the resemblance of real faults by different techniques. Hence we ask:

RQ3 *How do the employed techniques compare to each other in resembling real faults?*

Knowing how different techniques compare provides evidence in support of semantic fault resemblance. In particular, we check whether there is a compliment in fault resemblance or subsumption between fundamentally different approaches.

Overall, answering these questions raises awareness on the use of semantic and syntactic metrics in fault seeding and provides evidence on fault resemblance by fault seeding techniques.

V. Fault Seeding

PiTest (PIT) [16] is a state-of-the-art mutation testing tool that works by analyzing bytecode sequences and by looking for a possible location, i.e., instruction, to seed faults using syntactic transformation rules (aka mutant operators). The mutation operators are categorized into 29 task-specific distinct groups. Examples of groups include Conditionals Boundary and Return Value mutators, which seed variations concerning relational operators and method call return values. PiTest has over 120 mutant operators, among which are many experimental mutants used for scientific purposes. For this study, we take into consideration all mutants generated by PiTest.

[4]https://pitest.org/quickstart/mutators/#REMOVE_CONDITIONALS

IBIR [31] is a fault seeding tool that uses an information-retrieval-based fault localization model (IRFL) combined with automatic program repair inverted fix-patterns. It favours the generation of few but realistic mutants (similar to real ones). It takes as input the git repository of the program to mutate and a bug report, written in natural language and seeds faults (introducing multiple faulty versions) that emulate the fault described in the bug report.

IBIR starts by analysing the given bug report using IRFL [57] to identify locations that are likely to be related to the features impacted by the corresponding fault. It then applies fault patterns on the identified locations, which are inverted fix-patterns used in pattern-based automated program repair approaches [31]. As the fix patterns are crafted from real bug-fixes, their inverse would induce faults similar to real faults. IBIR repeats this process until exhausts all pre-defined patterns. In this study, we run IBIR on the classes changed by the bug-fix on Defects4J to exclude the mutants from other classes, and we apply all pre-defined patterns exhaustively on every location, instead of mutating only the lines predicted by the IRFL. This will allow us to explore more faulty patterns and study more broadly their relationship with real faults. Under this setting, IBIR results in producing a large number of mutants for the studied subjects.

DeepMutation [53] generates mutants by employing Neural Machine Translation [55] aka NMT, which is also used by many recent studies [20], [21], [52], [54]. It uses an NMT model trained on a large corpus ($\sim$787 k) of existing bug-fixing commits mined from GitHub repositories. It takes a Java method as input and outputs a mutant. In this study, we use beam search to generate a maximum of 10 mutants per method, which provides us with a more thorough study of the correlation with real faults.

In particular, every method is abstracted, in which the user-defined variable names and literals are replaced by predefined identifiers to obtain an abstracted code representation (as shown in Section III). These abstracted code representations are then input into the trained NMT model to produce abstracted mutants. The user-defined variable names and literals are restored to obtain source-code mutants.

We use the publicly available trained model of DeepMutation [2] to generate the mutants and `src2abs` [7] tool to perform the abstraction process. We followed the guidelines [53] and used beam search to generate 10 mutants per method.

μBERT [17] is a mutation testing tool that uses a pre-trained language model (CodeBERT) to generate mutants by masking and replacing tokens. μBERT takes a Java class, extracts tokenized expressions, which are then masked for token replacement (mutation), e.g., for binary expressions μBERT masks the binary operator, and invokes CodeBERT to complement the masked sequence. For instance, in sequence `int mid = (low + high) / 2;` μBERT mutates the variable name expression `low` by feeding CodeBERT with the masked sequence `int mid = (<mask> + high) / 2;`. CodeBERT predicts the 5 most likely tokens to replace the masked one, e.g., it predicts `low`, `mid`, `Low`, `high`, and `medium` for the given masked sequence. μBERT uses these predictions to generate mutants by replacing the masked token with the predicted ones (5 mutants are created per masked token). μBERT discards non-compilable mutants and those that are syntactically the same as the original program, which are the cases in which CodeBERT predicts the original masked token (aka duplicated mutants [32]).

VI. Experimental Setup

A. Real Faults

We used Defects4J [29] v2.0,0, which contains over 800 faults with supporting build infrastructure and forms one of the largest collections of reproducible real faults for Java programs.

Every fault in the dataset consists of the faulty and fixed versions of the code, a developer's test suite accompanying the project, and information regarding the commit modified classes and the patches produced to fix the fault. The faults have been manually minimized, so every irrelevant change to the fix has been removed. The dataset also includes at least one fault-triggering test that fails in the faulty version and passes in the fixed one.

For the purpose of this study, we consider the following projects and the number of faults. We refer to these curated faults (mined through a systematic process) as real faults. We consider Defects4J faults as a good sample since it consists of real, systematically mined faults that have been built independently of the present study. From Apache Commons [1] family, consisting of a collection of projects of Java utility classes, we include commons-cli (39), commons-codec (18), commons-compress (33), commons-csv (16), commons-math (101), commons-lang (63), commons-collections (4), commons-jxpath (22). We also include projects from the Jackson [4] family, which is a suite of data-processing tools for Java, we include jackson-core (26), jackson-databind (102), and jackson-dataformat-xml (6). Additionaly, we include faults from: Mockito (28), one of the most popular mocking frameworks in Java; Jsoup (90), a Java library for HTML parsing; Gson (18), a Java library for JSON parsing and generation from and into java objects; and joda-time (26), a project for the Java date and time classes.

Defects4J faults span more than a decade of development history, making it hard to apply all faulty versions with all the studied tools. Therefore, due to unsatisfied build requirements caused by the technical constraints, we do not consider certain faulty versions. The technical issues we encountered included obsolete dependencies not supported by studied tools, old testing frameworks (for example, some faults contain JUnit 3 while the tools work on JUnit4+), Java language versions (some of the tools require java 1.8+ to apply faulty patterns while the project Jfreechart (number of faults 26) and Closure-compiler (174) contain 1.5 or 1.6). Furthermore, five versions of the Math project also fall under this category. Additionally, at the time of conducting this study, we found that 5 faults from the Jsoup project were not compilable due to technical reasons, as already reported [3]. Overall, some of the studied tools have been recently developed and are versions specific, not being able to satisfy all the reported building requirements. In total, we analyzed 592 faults from 15 projects and generated a significant number of artificial faults that portray a representative dataset for our investigation.

TABLE III
MUTANTS USED

Fault Seeding Tool	# of Analysed Faults	# of Mutants
DeepMutation	530	119,017
IBIR	382	1,094,493
μBERT	481	286,763
PiTest	508	1,120,719

B. Artificially Seeded Faults

For each selected faulty project version from Defects4J, we start by identifying the modified classes between the faulty and fixed versions. Next, we generate mutants by employing the selected mutation testing tools for the fixed version of each modified class.

Table III records the number of faults analysed by each tool and the number of mutants generated. Overall, PiTest generated 1,120,719 mutants for the 508 faults that it was successfully applied to. μBERT was successfully applied on 481 faults and produced a set of 286,763 mutants. DeepMutation produces ten mutants per method, and thus, it produced 119,017 mutants for the 530 faults that it was successfully applied. After applying all faulty patterns from IBIR, it produces 1,094,493 mutants, per bug report, for the 382 faults that it operates.

After generation, in the mutant detection phase, we execute relevant tests from Defects4j, as those tests are carefully filtered by the framework to leave out flaky tests. We use Defects4Js predefined $compile$ and $test$ scripts.

C. Experimental Procedure

We start by executing every generated mutant using the Defects4J framework, thus, recording the set of failing tests distinguishing (killing) each mutant. After, we proceed to compute syntactic and semantic similarities between the mutants and the corresponding faults, relying on the metrics defined in Section II. Thus, the syntactic similarity between the mutant (artificially seeded fault) and the real fault will be measured in terms of the BLEU score, while the semantic similarity will be characterised by the Ochiai coefficient between the mutant and the fault. It is worth mentioning that, since PiTest produces the mutations at the bytecode level, we perform the syntactic similarity computation between the bytecode instruction sequences corresponding to mutants and faults.

To answer RQ1, we check the existence of correlations among the syntactic and semantic similarity of the seeded and real faults. We consider all the mutants created by all studied tools and analyse several cases; when mutants located in the project classes and when mutants located on the same methods modified by the related fault fixing patch, according to the information given by Defects4J (modified-methods mutants). In all cases, we aim for general trends that indicate a relationship between syntactic and semantic, over different percentages, similarity (e.g., values greater than 80%). We also check whether high scores for syntactic similarity (i.e., seeded and real faults are syntactically similar) imply high scores for semantic similarity (i.e., seeded and real faults behave the same), and whether low scores for syntactic metrics imply low scores for semantic metrics. To perform this, we sort mutants in ascending order according to their syntactic similarity. Thus, we organise them into four sorted quartiles Q_1, Q_2, Q_3 and Q_4, where Q_1 represents the most syntactically dissimilar mutants, w.r.t., the fault (lowest syntactic scores), while Q_4 represents the most syntactically similar mutants w.r.t. the fault (highest syntactic scores). For the mutants in each quartile, we also analyse their semantic similarity w.r.t. the fault, aiming to observe if there is any evidence that more syntactically dissimilar mutants behave very differently than the faults and whether syntactically similar mutants behave the same as the faults. To avoid potential threats in the quartiles composition, it is worth noting that we do not consider mutants which are extreme cases and introduce noise, such as those with Ochiai equal to zero and those that syntactically exactly match the real faults.

To further strengthen our analysis, we examine whether there are faults that do not have syntactically close mutants that are semantically close. Similarly, we examine if there are faults that do have syntactically distant mutants that are semantically close. We consider semantically close mutants to be the ones with Ochiai $> .8$, and syntactically close or distant, if they belong to previously defined Q_4 or Q_1, respectively. In particular, we count the number of faults with at least one mutant that is semantically close, among all mutants that are syntactically close and distant to the fault. We also do the same count among randomly picked mutants of the same sample size as the syntactically close and distant sets of mutants. This means that a relationship between the two metrics exists if there are many more faults with high semantic similarity to the mutants that are syntactically similar than to either those that are syntactically distant or to the randomly picked ones. Furthermore, we also analyze the extreme cases (i.e., the set of semantically and syntactically most close and most distant mutants). To do so, we repeat the previous analysis, but now from the set of semantically closest mutants, per fault, we select semantic and syntactic relations between the syntactically closest and the syntactically most distant mutant. We plot the data points on a scatter plot and further evaluate their relationship with statistical tests. This process allows us to check whether there is any potential trend in the very extreme case of syntactic similarities. Additionally, we also measure the average semantic or syntactic closeness of the faults and check whether there are any statistical differences among the sets of semantically/syntactically close or distant mutants. Therefore, we study the average semantic closeness of the most syntactically similar mutants (Q_4) and the most syntactically distant mutants (Q_1), plus the average syntactic closeness of the most semantically similar mutants (Q_4) and the most semantically distant mutants (Q_1).

Finally, we check the number of mutants that are in the intersection of the closest and distant syntactically and semantically similar mutant sets. In particular, we compute the average size of the intersection set between the most semantically close and most syntactically close mutants and compare it with that of the most syntactically distant and most semantically distant ones. Suppose the intersection of both sets – semantically close but syntactically distant mutants and vice versa – is similar

to the intersection of mutants semantically and syntactically close. In that case, we can reason that even if exists a small set of mutants semantically and syntactically close, there also exist mutants which are semantically similar but syntactically dissimilar (or vice versa), indicating that semantic similarity exists independently of syntactic similarity. We also measure the average size of the intersection, since a high intersection would suggest that for the small sets of mutants, high syntactic similarity encloses high semantics and vice versa. While the low intersection would further confirm our hypothesis around the absence of a relationship between semantic and syntactic similarity. To reduce the impact of selecting an arbitrary mutant sample size on our obtained results, we repeat this analysis with the closest and most distant 5, 10 and 20 mutants as well as the 5% and 10% closest mutants w.r.t. both metrics.

To answer RQ2, we measure the ratio of real faults for which at least one mutant has semantic similarity equal to 1. We focus the analysis on the same quartile split as done for RQ1 to observe whether syntactically similar or dissimilar mutants yield higher semantic similarity over different projects.

In RQ3, we analyse the percentage of real faults that each tool can resemble. Plus, we study the ability of the tools to resemble different faults. It is noted that we consider the intersection of faults for which each tool can generate mutants. To make a fair comparison, we are controlling for the number of seeded mutants. We, thus, study tool pairs based on the number of mutants each tool generates by randomly selecting and controlling the set of mutants for each tool and calculating the tool's mean ratio to produce a mutant that resembles the real fault. We do this to avoid bias because each tool generates a different number of mutants. To avoid coincidental results, we repeated the experiment 100 times.

Our dataset of generated mutants and results are publicly available in the accompanying website [6].

D. Statistical Analysis

To study the relationships between semantic and syntactic properties, we use a correlation metric since it analyses any statistical relationships between variables, whether causal or not. In particular, we use the Kendall rank coefficient (τ) (Tau-a) and Pearson product-moment correlation coefficient (r). In both cases, we use the 0.05 significance level. Each correlation coefficients measure similarity, taking values from -1 to 1. Values close to both ends represent negative and positive correlations, respectively. While values in a range of absolute 0.2 around zero denote absence and insignificant correlation. In our case, it refers to the degree to which a pair of variables are related. Concretely, the two variables we study characterize the syntactic and semantic similarity between faults and mutants. Particularly, we use the BLEU score as a syntactic similarity metric and the Ochiai coefficient as a semantic similarity metric (later on, during the discussion and threats to validity sections, we also include other syntactic and semantic metrics). Therefore, correlation measures whether the two variables are related and indicate a predictive relationship that can be exploited in practice, i.e., aiming at syntactic similarity instead of semantic as done by many approaches, e.g., DeepMutation. To evaluate the magnitude of difference between observed groups, we calculate the Vargha and Delaney A_{12} effect size [56]. A_{12} values over 0.56, 0.64 or 0.71 indicate a small, medium or large difference between two populations, respectively.

VII. Empirical Evaluation

A. RQ1: Syntactic and Semantic Similarity Between Seeded and Real Faults

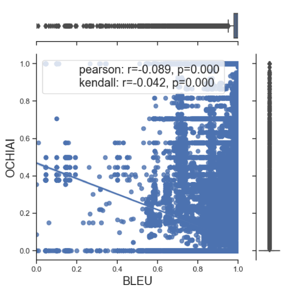
Fig. 2(a) shows the syntactic and semantic similarity values of the mutants created with different tools. Interestingly, we notice that while many of the mutants have high syntactic similarity, their semantic similarity is scattered from 0 to 1. This seems to imply that the relationship between the two metrics is weak. Fig. 2(b) depicts syntactic and semantic similarity values for all mutants with a syntactic similarity greater than 0.8. We notice that the mutants behaving as faults (obtaining Ochiai 1) are both syntactically similar and dissimilar to the faults (see the plots' top values, y-axis). We also observe that most mutants that are syntactically close to real faults (BLEU near 1) behave very differently (Ochiai near 0), indicating that the relationship is weak even when seeded faults are syntactically close to real ones.

These results are on the class granularity level, and therefore their syntactic and semantic changes may be impacted by the "size" of the seeded faults. We, thus, analyse the results at method-level granularity as well. Fig. 2(c) shows the syntactic and semantic similarities for the mutants that reside on the same methods as the real faults. In this case, we see a similar trend with the class-level results, i.e., both syntactically similar and dissimilar mutants behave exactly like real faults. Additionally, when syntactic similarity is close to 1, the semantic similarity is scattered from 0 to 1.

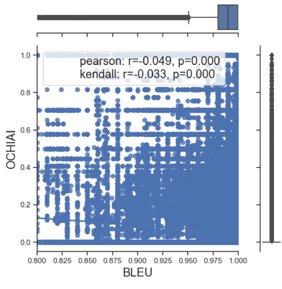
To further analyze this relationship, we investigate whether seeding faults with small syntactic distance results in semantically close faults. The key objective is to check whether there is some effect when we have a high syntactic similarity as well as high semantic similarity.

Fig. 2(d) shows the distribution of semantic similarities when we group mutants according to their syntactic similarity. We observe that semantic similarity is uniformly distributed between mutants that are syntactically similar and dissimilar to the real faults. This observation is visible even when considering only mutants with very close semantic similarity. This evidence that smaller syntactic transformations do not imply smaller semantic changes; at the same time, bigger syntactic changes do not imply bigger semantic changes.

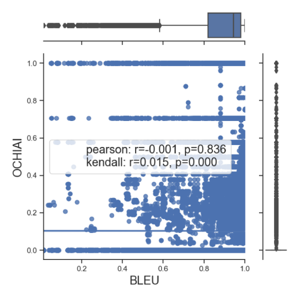
Additionally, we find that 39% of studied faults do not have syntactically close mutants that are semantically close (Ochiai $>$ .8). This percentage is roughly the same as that of syntactically distant mutants that are semantically close (41%). This observation indicates no relation between metrics since analysing either syntactically close or syntactically distant mutants, leads



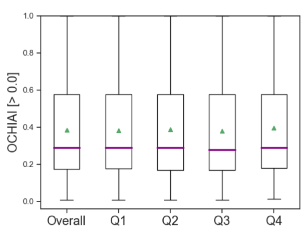
(a) At class granularity level. Weak link between syntactic and semantic similarity.



(b) When syntactic similarity is greater than 80%. We observe no link between syntactic and semantic similarity.

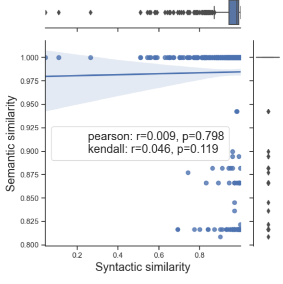


(c) At method granularity level. Faults on the same method have diverse semantic similarity.

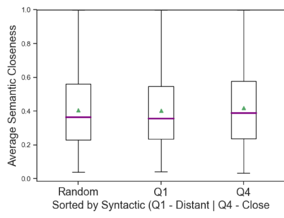


(d) Distribution of Semantic similarities across different levels of syntactic similarities. Medians depicted with lines, Means with triangles

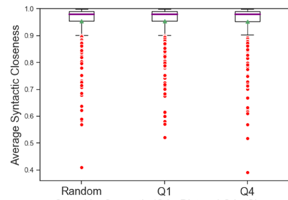
Fig. 2. Syntactic and semantic similarity between seeded and real faults (RQ1).



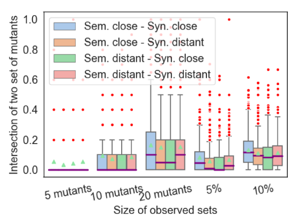
(a) Syntactic similarity of the semantic closest mutants. We observe no link between syntactic and semantic similarity.



(b) Average Semantic Closeness, per studied fault, of the most syntactically similar mutants (Q4), the most syntactically distant mutants (Q1), and a quartile of randomly sampled mutants.



(c) Average Syntactic Closeness, per studied fault, of the most semantically similar mutants (Q4), the most semantically distant mutants (Q1), and a quartile of randomly sample mutants.



(d) The average intersection size, per fault, between sets of mutants with closest and distant syntactical and semantical similarity.

Fig. 3. Syntactic and semantic similarity between seeded and real faults (RQ1).

to the same number of faults (w.r.t., mutants being semantically similar independently of their syntactic similarity). Fig. 3(a) shows the same data for the semantically closest mutants. From these data, we observe no relationship even in the extreme cases of the most syntactically close and distant mutants. By observing this plot, it becomes evident that there is no relationship between the two variables (note Pearson and Kendall correlation coefficients to be both $< .1$, indicating no relationship; plus, via Wilcoxon statistical test, we find no sign that higher syntactic similarity leads to higher values of semantic similarity ($p < 0.05$).

We also studied what is the average closeness of the semantically and syntactically closest fault mutant pairs (respectively, the Fig. 3(b) and (c)). Our results show that the average semantic closeness of the syntactically closest fault mutant pairs is 0.4192; the closeness of the syntactically distant mutant pairs is 0.4020, while the closeness of the randomly picked mutants is 0.4058. The difference between these averages is negligibly small to suggest a link between syntactic and semantic similarity metrics. Wilcoxon statistical test further confirms these observations by showing no statistically significant difference between the average semantic closeness of syntactically close and the average semantic closeness of syntactically distant mutants ($p < 0.05$), plus, no statistically significant difference between the average semantic closeness of syntactically close and randomly sampled mutants. Similar results are observed, leading to the same conclusions when calculating the average syntactic closeness of the semantically closest (0.9550) and semantically distant mutants (0.9530) and randomly sampled mutants (0.9543).

Fig. 3(d) depicts a follow-up analysis in which we study the intersection of a set of semantically closest and a set of syntactically closest mutants. Our results show that the intersection between the top five syntactically and semantically closest mutants is of only 5%, refuting any implication between both metrics in 95% of the cases. We observe the same pattern and small semantic and syntactic similarity overlapping for the top 10, 20 mutants, and 5 and 10 percent of mutants, 10%, 17%, 8% and 13%, respectively. Our results discover a small difference (2%) when comparing semantically and syntactically closest mutants overlapping with the overlapping of the semantically closest and syntactically distant ones (and vice versa). This analysis again confirms our previously shown evidence, i.e., even for a set of mutants with the metrics closest to real faults, semantics behaves independently of syntactic and vice versa.

TABLE IV
RQ2 Percentage of Resembled Real Faults - Quartiles Represent Mutants Sorted by Syntactic Similarity

Project	Faults	∃ Semantic Mutant	Q1	Q2	Q3	Q4
Cli	39	71.79	53.84	61.53	48.71	58.97
Codec	18	72.22	44.44	44.44	44.44	50.0
Collections	4	75.0	75.0	25.0	50.0	50.0
Compress	33	93.94	81.81	90.90	78.78	81.81
Csv	16	81.25	37.50	75.0	62.5	43.75
Gson	18	72.22	50.0	61.11	38.89	66.67
JacksonCore	26	88.46	73.07	65.38	73.07	76.92
JacksonDatabind	102	77.45	60.78	57.84	53.92	62.74
JacksonXml	6	83.33	83.33	83.33	50.0	83.33
Jsoup	90	12.22	11.11	12.22	7.77	7.77
JxPath	22	68.18	54.54	54.54	59.59	63.63
Lang	63	68.25	58.73	63.49	64.49	60.31
Math	101	67.32	54.45	52.47	59.40	59.40
Mockito	28	60.71	39.28	32.14	50.0	46.42
Time	26	65.38	50.0	53.84	50.0	57.69
Total/Average	592	70.51	55.19	55.55	52.67	57.96

TABLE V
RQ2: Mean Ratio of Mutants Resembling Real Faults - Quartiles Represent Mutants Sorted by Syntactic Similarity

Project	Ratio	Q1	Q2	Q3	Q4	Exact Matches
Cli	12.79	2.61	2.82	3.15	4.05	0.13
Codec	5.54	1.33	0.94	1.38	1.61	3.74
Collections	12.75	6.25	1.15	3.0	2.0	0.17
Compress	12.39	2.72	3.48	2.36	3.72	2.72
Csv	9.12	0.93	4.3	1.93	2.0	0.25
Gson	7.94	1.38	3.0	1.05	2.61	0.09
JacksonCore	19.46	4.88	4.23	4.42	6.00	0.54
JacksonDatabind	14.20	3.17	3.79	3.48	3.71	0.32
JacksonXml	11.66	4.33	2.33	2.33	3.0	0.37
Jsoup	5.13	1.12	1.68	1.01	1.12	0.86
JxPath	13.77	3.40	4.04	2.86	3.31	0.02
Lang	22.06	5.23	5.28	5.49	5.96	1.29
Math	12.16	2.98	2.61	2.78	3.71	0.10
Mockito	8.42	2.60	1.42	1.75	2.67	0.58
Time	9.26	2.15	2.76	1.92	2.23	0.08
Total/Average	11.77	3.01	2.95	2.59	3.20	0.75

By examining the faults that do not have syntactically close mutants that are semantically close and the average closeness of semantically and syntactically closest fault mutant pairs, we confirm the previous results and find no indication of a pattern or relationship that would suggest that syntactic measurement leads to the semantic closeness of mutants and real faults.

> Many seeded faults behave similarly to real faults (high semantic similarity), while they have low syntactic similarity to real faults. We find no evidence suggesting any link between syntactic and semantic similarity, except in the cases of exact matches.

B. RQ2: Semantically Resembling Real Faults

Table IV summarizes the results related to the percentage of resembled faults, i.e., having at least one mutant that semantically resembles the fault. The column Faults refers to the number of faults studied per project, and column ∃ Semantic Mutant refer to the percentages of faults with at least one semantically similar mutant. We sort mutants based on their syntactic similarity and group them into 4 buckets/quartiles (columns Q1, Q2, Q3 and Q4 in increasing order). Table V records the ratio of mutants that are semantically similar per fault (column Ratio) and the ratio per syntactically similar bucket.

TABLE VI
RQ3: Mean Ratios of Mutants Resembling Real Faults

Fault Seeding Tool	Overall	Q1	Q2	Q3	Q4
DeepMutation	1.99	0.39	0.42	0.56	0.59
IBIR	2.79	0.53	0.58	0.55	1.05
μBERT	2.94	0.69	0.54	0.74	0.89
PiTest	2.66	0.47	0.54	0.65	0.93

TABLE VII
RQ3 Percentage of Resembled Real Faults - Quartiles Represent Mutants Sorted by Syntactic Similarity

Fault Seeding Tool	Total	Q1	Q2	Q3	Q4	Exact Matches
DeepMutation	9.76	5.11	6.04	6.04	5.11	1.07
IBIR	76.44	38.60	46.04	59.06	59.65	1.98
μBERT	61.39	39.06	34.41	41.39	45.11	1.24
PiTest	65.11	42.32	45.11	46.97	56.27	0.01

For the 592 real faults, we observe that seeding techniques can produce at least one artificial fault that is semantically similar to the real one for 417 of them (70.51%). From the distribution of the results over different quartiles, we see the absence of trends suggesting that higher syntactic similarity does imply higher semantic similarity. For example, the project with the highest number of faults studied (JacksonDatabind – 102 faults) has quite similar ratios among the quartiles, i.e., 60.78%, 57.84%, 53.92%, 62.74% Overall, on average, across all studied faults, the distribution of faults that can be resembled is 55.19%, 55.55%, 52.67%, 57.96%.

Table V shows a similar distribution across quartiles. On average, the percentage of seeded mutants with semantic similarity is 11.77%, while over different levels of syntactic similarity, the distribution is 3.01%, 2.95%, 2.59% and 3.20%, respectively.

> Our results indicate that real faults are resembled by artificially seeded faults independently of their syntactic similarity.

C. RQ3: Comparing Seeding Techniques

Table VI records the mean ratios of mutants that resemble the real faults. We observe that, on average, between 1.99%-2.94% of mutants resemble the real faults, independently of their syntactic similarity. Table VII records the percentage of real faults that were resembled by at least one mutant produced by each tool. We observe that PiTest resembles 65.11% of the real faults, while μBERT resembles 61.39%, DeepMutation 9.76% and IBIR 76.44%. Interestingly, μBERT and PiTest resemble a similar number of the real faults, while μBERT identifies 0.6% of real faults not identified by other tools (Fig. 4), while PiTest identifies 1.7%. IBIR resembles 76.44% of faults, and 5.2% is not identified by the other tools. However, when we compare the performance when controlling the number of seeded faults (Table VIII), we observe that when generating a number of mutants equal to the number of mutants generated by DeepMutation, μBERT, IBIR and PiTest perform similarly, resembling real faults with 43.84%, 45.34% and 41.72%, respectively. When

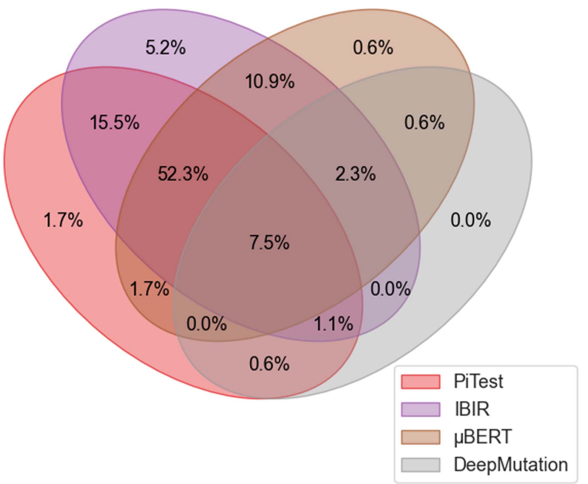


Fig. 4. Real faults with at least one semantically similar mutant by each tool. PiTest - IBIR - μBERT - DeepMutation.

TABLE VIII
RQ3 Percentage of Resembled Real Faults When the Number of Mutants is Controlled - Different Tools Used as a Baseline

Fault Seeding Tool	Mutant Selection Control Groups			
	DeepMutation	μBERT	PiTest	IBIR
DeepMutation	9.76	-	-	-
μBERT	43.84	61.39	-	-
PiTest	41.72	51.48	65.11	-
IBIR	45.34	62.34	73.16	76.44

generating the same number of mutants with μBERT, we observe that both IBIR and μBERT outperform PiTest by around 10%. And when the number of seeded faults is higher, (equal to what PiTest produces), IBIR outperforms PiTest for 8.05% of real faults resembled. We observe that PiTest performance is the lowest indicating a large number of redundancies. Regarding exact matching, IBIR successfully resembles 1.98% of the faults, outperforming the rest of the tools, which only managed to match around 1%. DeepMutation resembles real faults also resembled by other tools, indicating that it is probably subsumed by them.

> IBIR resembles 76.44% of the real faults, PiTest (65.11%), μBERT (61.39%) and DeepMutation (9.76%). When seeding the same number of mutants, IBIR shows better performance than all other tools (≈10%).

VIII. Discussion

A. Use Cases of Fault Seeding

Over the years, fault seeding has served multiple purposes, e.g., testing, dependability analysis, debugging etc., as found by the survey work of Papadakis et al. [42]. The survey also identifies that fault seeding is primarily used in research for *a) mutation-based test assessment*, i.e., empirical and experimental evaluation of test techniques (seeded faults are used as a means to compare test techniques based on the number of faults they detect), and *b) mutation-guided testing*, i.e., guiding testers to write test cases (by using seeded fault as objectives to be covered).

These two use cases are often confused and considered as being equal [15], while in fact they are different–though related. This is not only because of the different underlying processes but also due to the involved assumptions.

Process: In mutation-based test assessment, a) case, tests are independently produced by external parties, while in mutation-guided testing, b) case, tests aim at detecting specifically targeted faults. This implies an untargeted case (case a)) that starts from independent test cases and aims at estimating their fault detection potential versus a targeted one (case b)) that starts from seeded faults then goes to tests that aim at finding real faults.

In terms of injected faults, this difference means that in case a), one needs seeded faults that are as close (semantically) as possible to the real ones in order to estimate their test potential, while in case b), one needs seeded faults that lead to tests that detect faults. In essence, real faults in case a) should be detected *by every test case that detects a seeded fault*, while in case b) should be detected by *the subset of test cases that detect a seeded fault*. For example, consider the seeded faults M_0 and M_1 from Fig. 1 that both are detected by tests that also detect the real fault B. This means that for the case b) both M_0 and M_1 are equally useful since they can both lead to a test that detects the real fault. However, for the case a) (test assessment), M_0 is better than M_1 since it does not underestimate the fault detection potential of t_1.

Assumptions: In mutation-based test assessment, a) case, it is assumed that the tested/asserted program behaviour is correct, while in mutation-guided testing, b) case, this is not the case (testers judge the observed behaviour). These assumptions often imply differences in both the definition of fault detection (deciding whether mutants are killed) and the used artifact (by experimental studies). The difference in the definitions is usually that, in a) case, the behaviour of seeded faults is contrasted with that of the specifications (through test assertions) while, in b) case, that is contrasted with that of the program under test, which may or may not be correct. Similarly, experiments targeting case a) are applied on program versions where test suites pass, while in case b) experiments are applied on buggy program versions where test suites fail, i.e., detect some real faults.

In essence, the differences in the assumptions necessitate a different treatment in the way seeded faults are detected and used. For instance, it is unclear what causes a test failure when executing a test in a buggy program version where a fault has been seeded. Typically this case is treated by considering the behaviour delta between the buggy and the faulty seeded versions (seeded on the buggy version) [15]. However, this behaviour delta between the buggy and the seeded fault is different from the behaviour delta between the specifications and the buggy version as has been demonstrated by Chekam et al. [15].

Similarly, the seeded faults on the fixed and the buggy program versions differ. Consider, for example, the case of an omission fault where an if condition is missing. This fault is easy to emulate if one seeds faults in the non-buggy version by

TABLE IX
SEMANTIC SIMILARITY AND FAULT DETECTION PROBABILITY (FDP) BETWEEN THE MUTANTS AND THE FAULT

	Tests	t_0	t_1	t_2	t_3	t_4	$Ochiai(B, M_i)$	$FDP(B, M_i)$
Seeded Faults	B	✓	✓				-	-
	M_0	✓	✓				**1.00**	**1.00**
	M_1	✓					**0.70**	**1.00**
	M_2		✓	✓			**0.50**	0.50
	M_3	✓			✓	✓	0.40	0.33
	M_4			✓	✓	✓	0.00	0.00
	M_5	✓	✓	✓	✓	✓	**0.63**	0.40
	M_6	✓	✓	✓			**0.82**	0.66

simply deleting the related condition. However, the fault is hard to detect if one seeds faults in the buggy version since the related code is not there (it is hard to seed a fault that can detect such a bug).

The above discussion aims at detailing the differences among the two main use cases of fault seeding and motivating the need for appropriate metrics that fit well with the envisioned application use cases. In the following subsection we demonstrate the importance and appropriateness of using semantic similarity, as opposed to fault detection estimates, used by previous studies [13], [40], in the context of test assessment.

B. Semantic Similarity versus Fault Detection Probabilities in Test Assessment

Mutation testing has long been based on the notion of fault coupling [18], [38] that assumes couplings among different types and (syntactic) sizes of faults. This assumption has been validated by recent studies that report large semantic overlaps between simple and complex faults [28], [38], [41]. Undoubtedly, faults that couple with real ones are the most important when one performs mutation-guided testing (the b) use case of fault seeding that was described in the previous section) since the starting point is the seeded faults. However, this is not necessarily the case for the test assessment (the a) use case of fault seeding that was described in the previous section) since we want accurate estimations of test effectiveness.

Previous studies [13], [40] have defined *FDP*, as a probabilistic form of fault coupling, as a target metric of fault seeding that measures subsumption of a real fault by a mutant, in the context of mutation-guided testing. In essence, the metric form an approximation of the *fault detection probability*, w.r.t. a real fault B, of the tests that detect a seeded fault M. The metric is, therefore, computed as the ratio of the number of tests detecting both M and B to the number of tests detecting M. Precisely, $FDP(B, M) = \frac{|fTS_B \cap fTS_M|}{|fTS_M|}$, where fTS_B and fTS_M denote the set of tests detecting the fault and killing the mutant, respectively.

To illustrate this concept, let us consider the example of Fig. 1. The mutant killing matrix is presented in Table IX, together with the semantic similarity and fault detection probabilities between the mutants and the fault (columns $Ochiai(B, M_i)$ and $FDP(B, M_i)$, respectively).

An interesting observation from Table IX is that the FDP of mutants M_0 and M_1 is 1, since all the tests killing them also find the fault, but the $Ochiai$ coefficients (semantic similarity metric) distinguish between these mutants ($Ochiai(B, M_0) = 1$ but $Ochiai(B, M_1) = 0.70$). This example shows that in the case of mutation-guided testing, both M_0 and M_1 are of equal value (since targeting either of these faults leads to tests that detect the real fault). However, in the case of test assessment, M_0 is preferable over M_1 since it does not underestimate the test potential of t_1. Consider, for example, 10 combinations of test suites of two tests ($C(|\{t_{0\ldots4}\}|, 2) = \binom{|\{t_{0\ldots4}\}|}{2} = 10$). M_1 mistakenly evaluates the fault detection potential of $t_1 - t_2$, $t_1 - t_3$, $t_1 - t_4$ (3 out of 10) as being non-effective, while M_0 correctly evaluates them all. Similarly, M_6 is better than M_1 since it mistakenly evaluates the fault detection potential in 2 out of 10 cases, i.e., considers that $t_2 - t_3$, $t_2 - t_4$, are effective while they are not.

The above reflects the differences between the metrics of Table IX. The $Ochiai$ coefficient for mutant M_6 is 0.82, being the second top-ranked fault, making it preferable over M_1. While FDP would prefer M_0 and M_1 over M_6, which is actually the case if one is guided by the faults. Another difference occurs between mutants M_2 and M_5; while $Ochiai$ for M_5 is higher than for M_2, the opposite happens when we use FDP.

These examples aim at demonstrating the use of the metrics in the fault injection context. By considering these examples and the differences outlined in the previous section, it should be clear that both metrics are closely related, meaning that one could approximate the other, but semantic similarity ($Ochiai$) is a better fit for test assessment, while fault detection estimates (FDP) are a better fit for mutation-guided testing.

To empirically demonstrate these differences, we design a related test assessment experiment, reflecting the example given above. We thus, treat semantic similarity and fault detection estimates as estimators of the actual test assessment potential of the mutants and compute their related error. To measure this, we use the Mean Squared Error (MSE) of the estimators with respect to the actual ratios of fault detection potential of test suites, i.e., the ratio of test suites that detect both the seeded and the real fault over the number of test suites that detect either of them. The MSE is typically used as a quality indicator of the estimated values, in this case, semantic similarity and fault detection estimates, and aims at reflecting the associated risk of using them.

In this analysis, we randomly pick 100 test sets of equal size, determined by the ratios of tests detecting the real faults, in order to have a balance between failing and passing sets. We then computed the MSE values on the faults of our dataset, which have more than one failing test, 114 faults in total, of both semantic similarity and fault detection estimates for all available mutants. We selected cases with more than one failing test because the metrics are almost the same in all other cases.

Fig. 5 depicts the MSE errors of both estimators. These results show that both metrics have low error rates, with semantic similarity yielding statistically significantly lower errors with sizable differences, i.e., A_{12}, indicating that semantic similarity is better suited for test assessment.

When exploring further, we observe that despite the clear relationship between both semantic metrics, they prioritize mutants

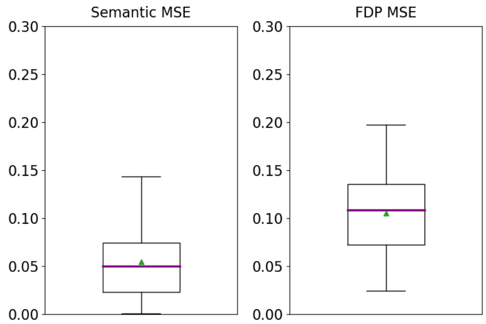


Fig. 5. Semantic Similarity yields significantly lower Mean Squared Errors in its assessments than Fault Detection Probability. The results are statistically significant with 99% of confidence and with a high effect size of 0.82.

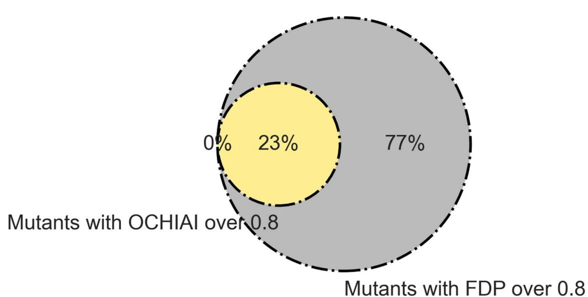


Fig. 6. Overlapping between the set of mutants with Ochiai coefficient greater than 0.8, and the set of mutants with FDP greater than 0.8. The figure shows significantly low overlapping between mutant sets, indicating that the metrics are appropriate for different use cases even though they are strongly related.

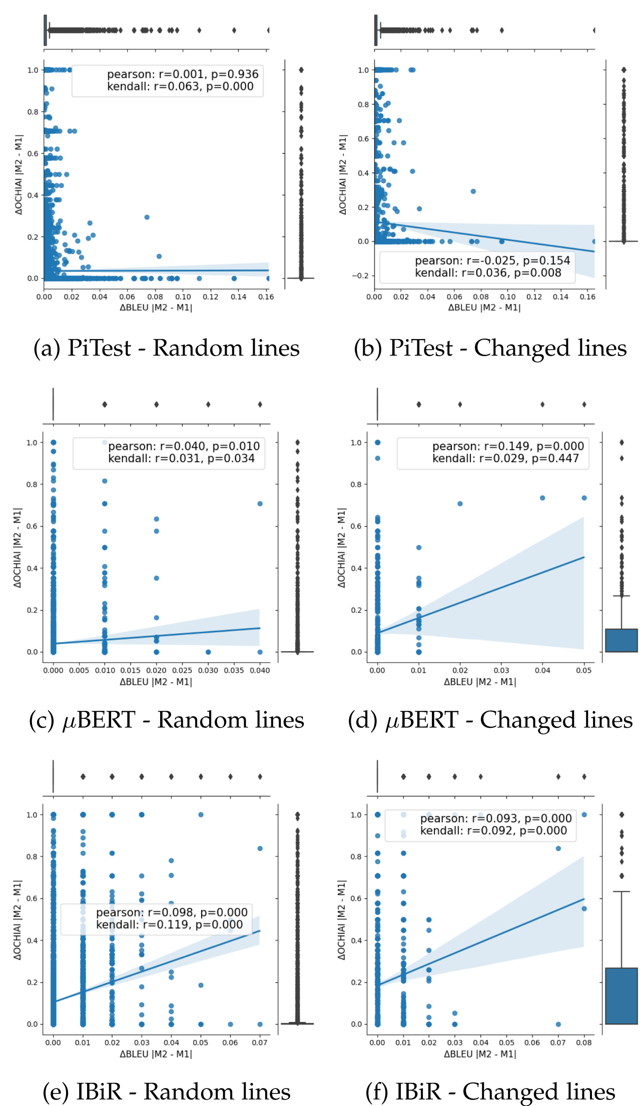


Fig. 7. Sensitivity of mutants from the same location ($\Delta BLEU|M_2 - M_1|$ over $\Delta Ochiai|M_2 - M_1|$). Small syntactic changes lead to diverse semantic changes.

differently. For instance, Fig. 6 shows two sets of mutants - one with high Ochiai and one with high FDP - and provides evidence that the overlapping is significantly low; only 23% of the mutants with a FDP greater than 80% have also an Ochiai greater than 80%. This may explain why the sets of mutants are distinct for different use cases, and particularly the preference of the semantic similarity metric (Ochiai) for test assessment.

Nevertheless, we also study if there is any relationship between syntactic similarity and FDP (fault subsumption) as a semantic similarity metric, instead of using Ochiai. We find that there is no relation and the message conveyed is the same as when using Ochiai as a semantic similarity metric, that real faults are subsumed independently of their syntactic similarity. On average all tools subsume 80.39% of studied faults, while different quartiles $Q1$, $Q2$, $Q3$, $Q4$ show 66.41%, 66.40%, 63.02% and 67.78%, respectively. We also find that different tools (PiTest, IBIR, μBERT, and DeepMutation) subsume 74.41%, 85.58%, 71.16%, 12.09% of real faults, respectively - keeping the same distribution as we report studying semantic similarity. Overall, the answer to the studied research questions would be similar if we would adopt a related but different semantic metric such as FDP. We provide further figures and tables regarding the subsumption of faults with FDP on the complementary web page:

https://mutationtesting-user.github.io/bugs_vs_mutants/

C. Sensitivity to Program Locations

One may wonder how sensitive the syntactic and semantic similarity metrics are with respect to the seeded faults' locations. In other words, these metrics may reflect the utility of the locations and not of the faults. Thus, we study the variance of the syntactic and semantic similarity of mutant pairs generated from the same location (we do not consider DeepMutation since it creates only one mutant per method). Fig. 7 records the distances between the syntactic and semantic similarity of mutant pairs, taken from a) the same randomly picked locations and b) from the bug-fixing locations. We observe that while there is almost no syntactic difference between mutants from the same location, the semantic similarity varies significantly. There are a few outliers in which syntactic similarity varies up to 16% between mutants from the same location. Some PiTest mutations remove a complete line or replace entire boolean conditions with true (as shown in the motivating example), affecting the bytecode

generated. Fig. 7 records a similar trend for all studied tools. Please refer to the accompanying website for results related to additional studied syntactic metrics [6].

Overall, these results support the conclusion that there is no link between syntactic and semantic similarity. Interestingly, even small syntactic changes in the same instruction can have a large and diverse impact on the program semantics.

D. Seeding Faults With DeepMutation

Guided by intuition, one would assume that DeepMutation - as a mutant generation tool based on deep learning - shall be able to generate more complex (stronger) faults and thus complement and subsume other tools. However, we observed that the tool seeds faults which are easy to kill or, in other words, the tests cannot miss behaviour produced by those kinds of seeded faults as they are overly complex, i.e., replacing too many code elements and thus significantly changing code logic. In particular, we found that DeepMutation mutants are, on average, identified by 10 tests more than the mutants from other tools. Moreover, we analysed seeded faults and found that from all seeded, 46% does not compile. In contrast, 29% are duplicates or not killed – leaving 25% of seeded faults suitable for mutation analysis. Out of those seeded faults, no fault can resemble faults that other tools cannot (Fig. 4), making the tool subsumed by other tools.

```
// Defects4j JxPath project - Bug version 7
private int compare(Object l, Object r) {
    double ld = InfoSetUtil.doubleValue(l);
    double rd = InfoSetUtil.doubleValue(r);
    return ld == rd ? 0 : ld < rd ? -1 : 1;
}
```

```
// DeepMutation mutant - does not compile due to the
     return type
private int compare(Object l , Object r) {
    double ld = InfoSetUtil.doubleValue(l);
    double rd = InfoSetUtil.doubleValue(r);
    return ld == rd ? 0 : ld;
}
```

Additionally, to further provide qualitative remarks on seeded faults, we provide two examples in which seeded faults indicate the technique's potential, even though the mutants are considered weak. In the example of a mutant in a method taken from project JxPath, the mutant alters the ternary operator condition, which is a syntactically adequate location for a bug; however, the mutation does not consider the return type, which results in a compilation error. In the second example, where we observe a mutant from the Mockito project, the technique removes the complete conditional check of whether an object is a valid instance, resulting in a weak mutation that cannot escape a test suite. Instead, the conditional check should be altered instead of removed, as those faults are subtle and represent a mistake that a programmer would make. However, the mutant from the second example alters the core logic of the code, which makes it unlikely to escape the majority of test cases (Ochiai metric is ≈ 0.02.

Moreover, using semantic similarity for reinforcement metrics for learning algorithms should bring more practical artificial faults than using syntactic metrics. This knowledge can provide practitioners with insights and pave the way to discover more fine-grained metrics to approximate semantics over existing ones.

```
// Defects4j Mockito project - Bug version 20
public MockHandler getHandler(Object mock) {
    if (!(mock instanceof
         MockMethodInterceptor.MockAccess)) {
        return null;
    }
    return ((MockMethodInterceptor.MockAccess)
         mock).getMockitoInterceptor().getMockHandler();
}
```

```
// DeepMutation mutant - mutant compiles but has Ochiai
     0.02 and it cannot escape any test
public MockHandler getHandler(Object mock) {
    return ((MockMethodInterceptor.MockAccess)
         mock).getMockitoInterceptor().getMockHandler();
}
```

E. Implications for Practice

Our key finding regards the mismatch of syntactic and semantic similarity. This implies that research studies should not attempt to approximate semantic similarity through syntactic similarity (as currently done by many methods). Therefore, researchers should focus on measuring the semantic sensitivity of their results and perhaps attempt learning based on semantic features rather than solely syntactic ones. For instance, DeepMutation aims at mimicking syntactically real faults thereby resulting in being relatively weak and probably subsumed by traditional mutants.

Additionally, our results shed light on the semantic similarity aspect of real and seeded faults that has not been researched by the mutation testing literature. Therefore, we believe our work can improve our understanding of this fundamental relationship and offers a starting point for future research on the semantic approximation of real faults.

Moreover, the Ochiai score is a metric used by previous work in order to approximate the semantic similarity between a bug and a seeded fault. We thus expect it to diverge from the true similarity due to the following two reasons. First, there is some noise due to the incompleteness of the test suit used, and second, due to the coarse granularity level of the test failures, i.e., we consider test failures in our approximation and not the exact program output. We aimed at mitigating both factors by using mature and strong test suites, augmented with automatically generated tests and manually estimated the level of error due to the application of semantic similarity at the test failure level(found it $\approx 2.8\%$ as reported in Section IX).

In our work, we also report on the existence of a specific category of seeded faults - the seeded faults that are syntactically dissimilar but semantically similar to real bugs. This category can provide implications for practice towards increased test assessment by providing targeted diversity represented through code comprehension.

IX. Threats to Validity

To reduce external validity threats, we selected a new and significant benchmark of faults that have not been used by previous studies. We excluded some faults for technical reasons, making our study with 592 faults from 15 mature and well-tested open-source real-world projects. Nevertheless, we do not exclude the generalization threat in other domains. As we already discussed, while conducting our experiments, we could not compile or run all the faulty program versions available in Defects4J.

We also acknowledge the threat that Defects4J faults may not be representative. Thus, our results reflect the similarities and the "representativeness" of the mutants we study with the Defects4J curated fault set. We believe that Defects4J faults form a good sample for our study since they have been independently mined, through a systematic procedure that does not favour in any direct way their semantic or syntactic similarity with our mutants. Future research should address this concern through replication studies on other datasets.

Internal validity threats emerge from the tools' specificity and configuration, such as the number of mutants they generate and the source-code locations they are applied to. For instance, DeepMutation produces fewer candidate mutants than any other tools, while μBERT, IBIR, and PiTest generate mutants everywhere, in a brute-force way. To mitigate this threat, we analyze the effectiveness of the tools under *the same number of mutants and the same locations* and observe a similar trend.

Unfortunately, we did not manage to compile and run the latest master-branch [5] version available of DeepMutation. We thus had to handle the tool from the resources and pre-trained artefacts provided in the repository.

Additional threat mitigation actions involved the analysis of mutants at different granularity levels (class, method, and location of patch). We also restricted the scope of analysis to the artefacts where the bug fixes were available to reduce noise from irrelevant mutants and tests. We ensured that all mutants reside on the same class/method/statement as the target faults. Thus, we compare different mutated versions w.r.t. their similarities and distances from the corresponding fixed and faulty version.

To measure semantic similarity, we used the well-known Ochiai score that has been regularly used in the fault-seeding community as a representative metric to capture the semantic similarity between a seeded and real fault. The metric takes into consideration test execution output and neglects the lower level of granularity, i.e., whether the test crashed due to error or due to failure, which may result in a divergence of behaviour between a bug and a mutant.

To study this threat, we conducted a manual follow-up analysis. In the manual study, due to the inability of Defects4J to provide fine-grained test outputs, we sample randomly from the mutant pool and analyze them in isolation. By taking around 4000 mutants that obtained an Ochiai coefficient equal to 1, which gives a confidence level of 99% with a confidence interval of 2%, we found that just approximately 2.8% of mutants show potentially different behaviour than the real fault (one triggers a failure while the other triggers an error), even though the same test captures them. This percentage of mutants does not impact the message we want to convey with our study; nevertheless, we found this concern necessary to inform practitioners.

In addition to BLEU scores, for measuring syntactic similarity, we also used Cosine [48] and Jaccard [36] similarity coefficients [6]. The results did not show any significant differences w.r.t. the ones of BLEU scores. It is worth noting that these metrics appear less often in the literature, and in an attempt to keep the story clear and concise, we provide results on these metrics on an accompanying website. Nevertheless, please refer to the accompanying website [6] for additional details on using Cosine and Jaccard similarity.

Overall, our study aims to raise awareness of using semantic and syntactic evaluation metrics in fault seeding studies since understanding is shaken by the rapid integration of "intelligence" in the current software testing practices. In the spirit of discarding all misinterpretations, we declare that it is far from our intention to generalize the studied approaches and raise the claims regarding their future usage as we believe in very much needed future studies on their utilities and effectiveness that will undoubtedly result in new tools. Our study targets the current state-of-the-art tools and embedded underlined approaches to shed more light on the studied area and pave the way for future work.

X. Related Work

Fault seeding and particularly mutation testing is widely used in experimental studies as a way to compare and assess testing techniques [42]. Assuming that seeded faults include properties that are in some sense similar to real ones [9]. Interestingly, mutation testing, one of the most widely used techniques [42], introduces faults that are syntactically simple and are quite different from real faults that are in their majority more complex [23]. In particular, the study of Gopinath et al. [23] provided empirical evidence showing the misalignment between seeded and real faults that are produced by traditional mutation operators and concluded that real faults are rarely equivalent to mutant faults.

To deal with this issue, Brown et al. [12] proposed inferring fault seeding patterns, w.r.t, mutation operators, by using historical fault-fixing commits. The idea was to form (syntactic) fault patterns that resemble (in terms of syntax) real historical faults. Their results show that syntactic fault patterns can be mined from code versioning systems, and these differ (syntactically) from those used by modern mutation testing tools.

DeepMutation [53], a neural machine translation technique [55] that automatically infers fault patterns from historical fault-fixing commits, was proposed. It was shown that DeepMutation resembles exact matches of 45% of real faulty cases while achieving relatively good syntactic similarity scores in most of the cases. SemSeed [45] aims to infer faulty patterns from bug fixes and to generalize them by appropriately adapting them to the particular local code, i.e., context. Although powerful, SemSeed operators on JavaScript programs make its application in our experiment hard.

More recently, mutation monkey [11] was built by mining frequently occurring faults from complex changes that caused

operational issues at Facebook [11]. The analysis of these faults indicated they were good at finding holes and missing tests in the systems under test. Interestingly, the above studies aim at mimicking (syntactically) real faults, and as a result, they have been evaluated with "static" syntactic-nature metrics such as syntactic similarity. Hence, this raises the question of whether they are suitable for dynamic analysis, such as mutation testing, i.e., incorporating realistic semantic fault properties, and how they compare with traditional mutation testing, which we investigate here.

Traditionally mutation testing aims at seeding faults using simple syntactic changes. Showing empirical evidence of the coupling effect, [18] states that simple faults subsume almost all the complex ones [38]. This implies a more general assumption about the "size" of faults [39], suggesting that seeded faults with small syntactic distance from the original program introduce small semantic deviations (subtle faults), which form valuable test requirements [14] and lead to high fault revealing potential [40].

The coupling between seeded faults has also been considered a source of bias in mutation testing studies as it introduces large overlaps between the seeded fault instances [28], [35], [41]. Nevertheless, the question of how to select optimal mutant-fault sets falls outside the scope of this work.

XI. Future Work

Our work paves the way for researchers to investigate properties which suggest a semantic similarity. In particular, research on semantic similarity may be more fruitful than syntactic metrics. Moreover, in the area of fault seeding, some experimental work uses different seeded fault properties to reinforce learning algorithms. We believe, in particular, that if not semantic metrics (such as Ochiai or Fault Detection Probability), there is a potential interest towards new metrics that capture the fine-grained similarity between seeded and real faults, i.e., traces, data dependencies, code or test assertions actual values etc., which should be investigated with some sort of change impact analysis.

Additionally, we report on the existence of mutants that are syntactically dissimilar but semantically similar. These mutants may be proven useful in fault comprehension and thus could be interesting to be studied further.

We also observe that IBIR is the tool that resembles the most observed real faults and subsumes most of the faults and, thus, other tools. However, we identified a potential for improvement as other tools resemble bugs that IBIR does not, suggesting an investigation into potential faulty patterns that the tool misses. Additionally, as food for thought, we point to the number of faults that IBIR seeds, which is significantly higher than other tools, making another new question arise of how cost-effective the tools are.

XII. Conclusion

We investigated the link between syntactic and semantic similarity of seeded and real faults in the context of mutation-based test assessment. Our results showed that many seeded faults behave similarly to real ones (they have high semantic similarity), while at the same time having low syntactic similarity (to real faults). We also observed the opposite case, i.e., faults with high syntactic similarity having low semantic one. This means that we found no evidence suggesting any link between syntactic and semantic similarity, except, of course, in the case of exact matches. When considering the ability of fault injection tools to resemble real faults, we found that 65.11%, 76.44%, 61.39% and 9.76% of the real faults in Defects4J V2 are semantically resembled by PiTest, IBIR, μBERT and DeepMutation faults, respectively. For further inquiry about our data, figures and examples, please refer to the webpage of the paper:

https://mutationtesting-user.github.io/bugs_vs_mutants/

References

[1] Apache Commons. [Online]. Available: https://github.com/apache
[2] DeepMutation. [Online]. Available: https://github.com/micheletufano/DeepMutation
[3] Defects4j issue- 353. [Online]. Available: https://github.com/rjust/defects4j/issues/353
[4] FasterXML Jackson. [Online]. Available: https://github.com/FasterXML/jackson
[5] Master branch deepMutation. [Online]. Available: https://github.com/micheletufano/DeepMutation/commit/a20882d8fbd107762e2d40f5742d838242dbf1e5
[6] Supplemantary data - webpage. [Online]. Available: https://mutationtesting-user.github.io/bugs_vs_mutants
[7] src2abs. [Online]. Available: https://github.com/micheletufano/src2abs
[8] M. Allamanis, H. Jackson-Flux, and M. Brockschmidt, "Self-supervised bug detection and repair," in *Proc. 34th Proc. Adv. Neural Inf. Process. Syst.*, 2021, pp. 27865–27876.
[9] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?," in *Proc. 27th Int. Conf. Softw. Eng.*, May 15–21 2005, St. Louis, MO, USA, 2005, pp. 402–411.
[10] P. Bareiß, B. Souza, M. d'Amorim, and M. Pradel, "Code generation tools (almost) for free? A study of few-shot, pre-trained language models on code," 2022, *arXiv:2206.01335*.
[11] M. Beller et al., "What it would take to use mutation testing in industry–A study at Facebook," in *Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng. Pract.*, 2021, pp. 268–277.
[12] D. B. Brown, M. Vaughn, B. Liblit, and T. Reps, "The care and feeding of wild-caught mutants," in *Proc. 11th Joint Meeting Found. Softw. Eng.*, 2017, pp. 511–522.
[13] T.T. Chekam, M. Papadakis, F. Tegawendé, Y. L.B. Traon, and K. Sen, "Selecting fault revealing mutants," *Empirical Softw. Eng.*, vol. 25, no. 1, pp. 434–487, 2020.
[14] T.T. Chekam, M.M. P. Cordy, and Y. L. Traon, "Killing stubborn mutants with symbolic execution," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 2, pp. 19:1–19:23, 2021.
[15] T. T. Chekam, M. Papadakis, Y. Le Traon, and M. Harman, "An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption," in *Proc. IEEE 39th Int. Conf. Softw. Eng.*, Buenos Aires, Argentina, May 20–28, 2017, pp. 597–608.
[16] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "PIT: A practical mutation testing tool for Java (DEMO)," in *Proc. 25th Int. Symp. Softw. Testing Anal.*, Saarbrücken, Germany, Jul. 18–20, 2016, pp. 449–452.
[17] R. Degiovanni and M. Papadakis, "BERT: Mutation testing using pre-trained language models," in *Proc. IEEE Int. Conf. Softw. Testing Verification Validation Workshops*, 2022, pp. 160–169.
[18] R. Demillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, pp. 34–41, 1978.
[19] Z. Feng et al., "CodeBERT: A pre-trained model for programming and natural languages," in *Proc. Conf. Empirical Methods Natural Lang. Process.: Findings*, Nov. 16–20, 2020, pp. 1536–1547.
[20] A. Garg, R. Degiovanni, M. Jimenez, M. Cordy, M. Papadakis, and Y. L. Traon, "Learning from what we know: How to perform vulnerability prediction using noisy historical data," *Empirical Softw. Eng.*, vol. 27, no. 7, 2022, Art. no. 169.

[21] A. Garg, M. Ojdanic, R. Degiovanni, T.T. Chekam, M. Papadakis, and Y. L. Traon, "Cerebro: Static subsuming mutant selection," *IEEE Trans. Softw. Eng.*, vol. 49, no. 1, pp. 24–43, Jan. 2023.
[22] N. E. Gold et al., "Generalized observational slicing for tree-represented modelling languages," in *Proc. 11th Joint Meeting Found. Softw. Eng.*, Paderborn, Germany, Sep. 4–8, 2017, pp. 547–558.
[23] R. Gopinath, C. Jensen, and A. Groce, "Mutations: How close are they to real faults?," in *Proc. IEEE 25th Int. Symp. Softw. Rel. Eng.*, 2014, pp. 189–200.
[24] C. L. Goues, T. V. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 54–72, Jan./Feb. 2012.
[25] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. T. Devanbu, "On the naturalness of software," in *Proc. 34th Int. Conf. Softw. Eng.*, Zurich, Switzerland, Jun. 2–9, 2012, pp. 837–847.
[26] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *ACM SIGPLAN Notices*, vol. 39, no. 12, pp. 92–106, 2004.
[27] A. Islam and D. Inkpen, "Semantic text similarity using corpus-based word similarity and string similarity," *ACM Trans. Knowl. Discov. Data*, vol. 2, no. 2, pp. 1–25, Jul. 2008.
[28] Y. Jia and M. Harman, "Higher order mutation testing," *Inf. Softw. Technol.*, vol. 51, no. 10, pp. 1379–1393, 2009.
[29] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *Proc. Int. Symp. Softw. Testing Anal.*, New York, NY, USA, 2014, pp. 437–440.
[30] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Hong Kong, China, Nov. 16–22, 2014, pp. 654–665.
[31] A. Khanfir et al., "IBiR: Bug-report-driven fault injection," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 2, pp. 1–31, 2023.
[32] M. Kintis, M. Papadakis, Y. Jia, N. Malevris, Y. L. Traon, and M. Harman, "Detecting trivial mutant equivalences via compiler optimisations," *IEEE Trans. Softw. Eng.*, vol. 44, no. 4, pp. 308–333, Apr. 2018.
[33] 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," *Empirical Softw. Eng.*, vol. 23, no. 4, pp. 2426–2463, 2018.
[34] P. Koehn, F. J. Och, and D. Marcu, "Statistical phrase-based translation," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics Hum. Lang. Technol.*, 2003, pp. 48–54.
[35] B. Kurtz, P. Ammann, J. Offutt, M. E. Delamaro, M. Kurtz, and N. Gökçe, "Analyzing the validity of selective mutation with dominator mutants," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Seattle, WA, USA, Nov. 13–18, 2016, pp. 571–582.
[36] S. Niwattanakul, J. Singthongchai, E. Naenudorn, and S. Wanapu, "Using of Jaccard coefficient for keywords similarity," *Proc. Int. Multiconference Engineers Comput. Scientists*, vol. 1, no. 6, pp. 380–384, 2013.
[37] A. J. Offutt and R. H. Untch, "Mutation 2000: Uniting the orthogonal," *Mutat. Testing New Century*, vol. 24, pp. 34–44, 2001.
[38] A. J. Offutt, "Investigations of the software testing coupling effect," *ACM Trans. Softw. Eng. Methodol.*, vol. 1, no. 1, pp. 5–20, Jan. 1992.
[39] A. J. Offutt and J. H. Hayes, "A semantic model of program faults," in *Proc. Int. Symp. Softw. Testing Anal.*, San Diego, CA, USA, Jan. 8–10, 1996, pp. 195–200.
[40] M. Papadakis, T.T. Chekam, and Y. L. Traon, "Mutant quality indicators," in *Proc. IEEE Int. Conf. Softw. Testing, Verification Validation Workshops*, Västerås, Sweden, Apr. 9–13, 2018, pp. 32–39.
[41] M. Papadakis, C. Henard, M. Harman, Y. Jia, and Y. Le Traon, "Threats to the validity of mutation-based test assessment," in *Proc. 25th Int. Symp. Softw. Testing Anal.*, Saarbrücken, Germany, Jul. 18–20, 2016, pp. 354–365.
[42] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman, "Chapter six - mutation testing advances: An analysis and survey," *Adv. Comput.*, vol. 112, pp. 275–378, 2019.
[43] M. Papadakis, D. Shin, S. Yoo, and D.-H. Bae, "Are mutation scores correlated with real fault detection?: A large scale empirical study on the relationship between mutants and real faults," in *Proc. 40th Int. Conf. Softw. Eng.*, Gothenburg, Sweden, May 27–Jun. 03, 2018, pp. 537–548.
[44] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "BLEU: A method for automatic evaluation of machine translation," in *Proc. 40th Annu. Meeting Assoc. Comput. Linguistics*, 2002, pp. 311–318.
[45] J. Patra and M. Pradel, "Semantic bug seeding: A learning-based approach for creating realistic bugs," *Proc. 29th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, New York, NY, USA, 2021, pp. 906–918.
[46] M. V. Pour, Z. Li, L. Ma, and H. Hemmati, "A search-based testing framework for deep neural networks of source code embedding," in *Proc. IEEE 14th Conf. Softw. Testing Verification Validation*, 2021, pp. 36–46.
[47] M. Pradel and K. Sen, "DeepBugs: A learning approach to name-based bug detection," *Proc. ACM Program. Lang*, vol. 147, pp. 1–147:25, 2018.
[48] G. Qian, S.Y. S. Gu, and S. Pramanik, "Similarity between Euclidean and cosine angle distance for nearest neighbor queries," in *Proc. ACM Symp. Appl. Comput.*, New York, NY, USA, 2004, pp. 1232–1237.
[49] C. Raffel et al., "Exploring the limits of transfer learning with a unified text-to-text transformer," 2020, *arXiv1910.10683*.
[50] C. Richter and H. Wehrheim, "Learning realistic mutations: Bug creation for neural bug detectors," in *Proc. IEEE 15th Int. Conf. Softw. Testing Verification Validation*, 2022, pp. 162–173.
[51] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proc. 27th Int. Conf. Neural Inf. Process. Syst.*, Cambridge, MA, USA, 2014, pp. 3104–3112.
[52] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2014, vol. 27.
[53] M. Tufano et al., "DeepMutation: A neural mutation tool," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng.: Companion*, New York, NY, USA, 2020, pp. 29–32.
[54] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 4, pp. 19:1–19:29, 2019.
[55] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "Learning how to mutate source code from bug-fixes," *IEEE Int. Conf. Softw. Maintenance Evol.*, pp. 301–312, 2019.
[56] A. Vargha and H. D. Delaney, "A critique and improvement of the "CL" common language effect size statistics of McGraw and Wong," *J. Educ. Behav. Statist.*, vol. 25, no. 2, pp. 101–132, 2000.
[57] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports," in *Proc. IEEE 34th Int. Conf. Softw. Eng.*, Zurich, Switzerland, Jun. 2–9, 2012, pp. 14–24.

Milos Ojdanic received the MSc degree from the Faculty of Innovation, Design, and Technology from the Mälardalen University, Sweden, in 2019. He is a doctoral researcher with the Interdisciplinary Center for Security, Reliability, and Trust (SnT) with the University of Luxembourg. His research interests are in software development, testing, and evolution. In particular, he focuses on evolving systems, change-aware testing criteria, mutation testing, and prediction modeling.

Aayush Garg received the MS degree in computer science with a concentration in Security from Boston University, United States, in 2019 and the PhD degree from the University of Luxembourg. He is a post-doctoral researcher with the Interdisciplinary Center for Security, Reliability and Trust (SnT) with the University of Luxembourg. He has several years of industrial experience as a Software Developer in Fintech organizations. His research areas comprise computer security, computational intelligence in software engineering, and mutation testing.

Ahmed Khanfir received the engineering diploma degree in the same field from the National Engineering School of Sousse (ENISo), Tunisia, in 2013 and the PhD diploma degree in computer sciences. He is a research associate with the Interdisciplinary Center for Security, Reliability and Trust (SnT) with the University of Luxembourg. His research interests are in software engineering, particularly software testing, test assessment and computer security.

Renzo Degiovanni (Member, IEEE) received the PhD diploma degree in computer science from the National University of Cordoba, Argentina. He is a research scientist with the Interdisciplinary Center for Security, Reliability and Trust (SnT) with the University of Luxembourg. His research interests are in software engineering, specifically the validation and verification of software. His research has contributed to the automation of requirements engineering activities, software testing and formal software verification.

Mike Papadakis received the PhD diploma degree in computer science from the Athens University of Economics and Business. He is an associate professor with the Interdisciplinary Center for Security, Reliability and Trust (SnT) with the University of Luxembourg. He is recognised for his work on software testing and in particular in the area of mutation testing. His research interests also include static analysis, prediction modelling and search-based software engineering.

Yves Le Traon is a professor with the University of Luxembourg where he leads the SERVAL (SEcurity, Reasoning and VALidation) research team. His research interests within the group include (1) innovative testing and debugging techniques, (2) Android apps security and reliability using static code analysis, machine learning techniques and, (3) model-driven engineering with a focus on IoT and CPS. His reputation in the domain of software testing is acknowledged by the community. He has been General Chair of major conferences in the domain, such as the 2013 IEEE International Conference on Software Testing, Verification and Validation (ICST), and Program Chair of the 2016 IEEE International Conference on Software Quality, Reliability and Security (QRS). He serves at the editorial boards of several, internationally-known journals (STVR, SoSym, IEEE Transactions on Reliability) and is author of more than 150 publications in international peer-reviewed conferences and journals.