μBERT: Mutation Testing using Pre-Trained Language Models

Renzo Degiovanni
SnT, University of Luxembourg, Luxembourg

Mike Papadakis
SnT, University of Luxembourg, Luxembourg

***Abstract*—We introduce μBERT, a mutation testing tool that uses a pre-trained language model (CodeBERT) to generate mutants. This is done by masking a token from the expression given as input and using CodeBERT to predict it. Thus, the mutants are generated by replacing the masked tokens with the predicted ones. We evaluate μBERT on 40 real faults from Defects4J and show that it can detect 27 out of the 40 faults, while the baseline (PiTest) detects 26 of them. We also show that μBERT can be 2 times more cost-effective than PiTest, when the same number of mutants are analysed. Additionally, we evaluate the impact of μBERT's mutants when used by program assertion inference techniques, and show that they can help in producing better specifications. Finally, we discuss about the quality and naturalness of some interesting mutants produced by μBERT during our experimental evaluation.**

I. Introduction

Mutation testing seeds faults using a predefined set of simple syntactic transformations, aka mutation operators, that are (typically) defined based on the grammar of the targeted programming language [29]. As a result, mutation operators often alter the program semantics in ways that often lead to unnatural code (unnatural in the sense that the mutated code is unlikely to be produced by a competent programmer).

Such unnatural faults may not be convincing for developers as they might perceive them as unrealistic/uninteresting [3], thereby hindering the usability of the method. Additionally, the use of unnatural mutants may have actual impact on the guidance and assessment capabilities of mutation testing [13]. This is because unnatural mutants often lead to exceptions, or segmentation faults, infinite loops and other trivial cases.

To deal with this issue, we propose forming mutants that are in some sense natural; meaning that the mutated code/statement follows the implicit rules, coding conventions and generally representativeness of the code produced by competent programmers. We define/capture this naturalness of mutants using language models trained on big code that learn (quantify) the occurrence of code tokens given their surrounding code.

In particular, recent research has developed pre-trained models, such as CodeBERT [10], using a corpus of more than 6.4 million programs, which could be used to generate natural mutants. Such pre-trained models have been trained to predict (complete) missing tokens (masked tokens) from token sequences. For example, given the masked sequence `int a = <mask>;`, CodeBERT predicts that `0`, `1`, `b`, `2`, and `10` are the (five) most likely tokens/mutants to replace the masked one (ordered in descending order according to their score – likelihood).

In view of this, we present μBERT, a mutation testing tool that uses a pre-trained language model (CodeBERT) to generate mutants by masking and replacing tokens. μBERT combines mutation testing and natural language processing to form natural mutants. In contrast to resent research [5], [24] that aims at mutant selection, μBERT directly generates mutants without relying on any syntactic-based mutation operators. This approach is further appealing since it simplifies the creation of mutants and limits their number.

Although, there are many ways to tune μBERT by considering mutants' locations and their impact, in our preliminary analysis, we seed faults in a brute-force way, similarly to mutation testing, by iterating every program statement and masking every involved token. In particular, we make the following steps: (1) select and mask one token at a time, depending on the type of expression being analysed; (2) feed CodeBERT with the masked sequence and obtain the predictions; (3) create mutants by replacing the masked token with the predicted ones; and (4) discard non-compilable and duplicate mutants (mutants syntactically the equal to original code). Figure 1 shows an overview of μBERT workflow.

To show the potential of μBERT we perform a preliminary evaluation on the following two use cases:

Fault Detection: We focus on a mutation testing scenario and analyse the fault detection capabilities of suites designed to kill μBERT's mutants, and compare them with those of a popular mutation testing tool, i.e., PiTest [6]. We consider a total of 40 bugs from Defects4J [19] for 3 projects, namely Cli, Collections and Csv. Our results show that test suites guided by μBERT finds 27 out of the 40 bugs, while PiTest's mutants helps in finding 26 out of the 40 bugs. 3 of the bugs found by μBERT are not found by PiTest, while 2 of the bugs found by PiTest are not found by μBERT. Moreover, we show that μBERT is (up to 100%) more cost-effective than PiTest.

Assertion inference: We study the usefulness of μBERT's mutants in the context of program assertion inference techniques, that use mutants to rank and discard candidate assertions [16] (typically, assertions that kill more mutants are preferred among others, and assertions not killing any mutant are discarded). In particular, we focus on the 4 cases recently reported in [26] in which traditional mutation testing did not perform well. We show that μBERT can complement and contribute with interesting mutants than can help in improving

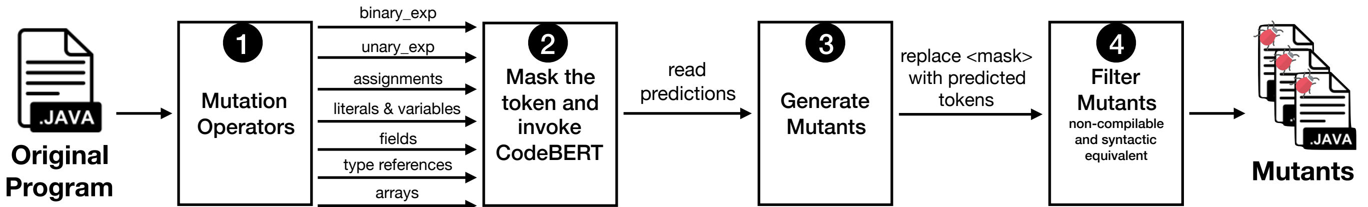


Fig. 1: μBERT Workflow: (1) it parses the Java code given as input, and extracts the expressions to mutate according to the mutation operators; (2) it masks the token of interest and invokes CodeBERT; (3) it generates the mutants by replacing the masked token with CodeBERT predictions; and (4) it discards non-compilable and syntactic the same mutants.

the quality of the assertions inferred.

Finally, we show examples of the mutants generated by μBERT with interesting properties, demonstrating their differences from traditional mutation.

II. Pre-Trained Language Models

CodeBERT [10] is a powerful bimodal pre-trained language model that produces general-purpose representations for natural language, in six programming languages, including Java. It supports several tasks, such as, natural language code search and code documentation. Particularly, CodeBERT supports the Masked Language Modelling (MLM) task that consists of randomly masking some of the tokens from the input, and the objective is to predict the original tokens of the masked word based only on its context. To do so, CodeBERT uses multi-layer bidirectional Transformer [33] to capture the semantic connection between the tokens and the surrounding code, meaning that the predictions are context-dependent (e.g. the same variable name, masked in different program locations, will likely get different predictions).

Precisely, CodeBERT can be fed with sequences of up to 512 tokens (maximum sequence length supported) that include exactly one (1) masked token (`<mask>`). Hence, when fed with a masked sequence, CodeBERT will predict the 5 most likely tokens to replace the masked one. Despite the good precision of CodeBERT in reproducing the original (masked) token, μBERT uses all the predicted tokens to introduce mutations in the original program. We argue that mutations introduced by μBERT will be in some sense natural, since CodeBERT was pre-trained on a large corpus (near 6.4 million programs) and thus, the mutated statements will follow frequent/repetitive coding conventions and patterns produced by programmers learned by the pre-trained language model.

It is worth noticing that μBERT uses CodeBERT as a black-box, so it will benefit from any improvement that the pre-trained model can bring in the future, as well as, other language models (supporting MLM task) can be integrated. Perhaps more importantly, generative pre-trained language models simplify the creation and selection of mutants to a standard usage of the model.

III. μBERT: CodeBERT-based Mutant Generation

μBERT is an automated approach that uses a pre-trained language model (namely, CodeBERT) to generate mutants for Java programs. Figure 1 describes the workflow of μBERT that can be summarised as follows:

1) μBERT starts by parsing the Java class given as input, and extracts the candidate expressions to mutate.
2) The mutation operators analyse and mask the token of interest for each java expression (e.g., the binary expression mutation will mask the binary operator), and then invoke CodeBERT to predict the masked token. μBERT will try to feed CodeBERT with sequences covering as much surrounding context as possible of the expression under analysis (512 tokens maximum).
3) μBERT takes CodeBERT predictions, and generate mutants by replacing the masked token with the predicted tokens (5 mutants are created per masked expression).
4) Finally, mutants that do not compile, or are syntactic the same as the original program (cases in which CodeBERT predicts the original masked token), are discarded.

Our prototype implementation supports a wide variety of Java expressions, being able to mutate unary/binary expressions, assignment statements, literals, variable names, method calls, object field accesses, among others. This indicates that for the same program location, several mutants can be generated. For instance, for a binary expression like `a + b`, μBERT will create (potentially 15) mutants from the following 3 masked sequences: `<mask> + b`, `a <mask> b`, and `a + <mask>`. Bellow we provide some examples that demonstrate the different mutation operators supported by μBERT.

A. Binary Expression Mutation

Given e = `<exp> <op> <exp>`, a binary expression of a method M in program P to mutate, where `<exp>` and `<op>` denote a Java expression and a binary operator, respectively, μBERT creates a new expression e' = `<exp>` **`<mask>`** `<exp>` by replacing (masking) the binary operator `<op>` with the special token `<mask>`. Then, a new method $M' = M[e \leftarrow e']$ is created that looks exactly as M, but expression e is replaced by masked expression e'. μBERT invokes CodeBERT with the largest code sequence from method M' that, includes e' and, does not exceed the maximum sequence length (512 tokens). CodeBERT returns a set with the 5 predicted tokens ($t_1, \ldots, t_5$). Hence, μBERT generates 5 mutants, namely $P_1, \ldots, P_5$, such that each mutant P_i replaces the mutated operator `<op>` by the predicted one t_i. That is, $P_i = P[e \leftarrow e_i]$, where e_i = `<exp>` t_i `<exp>` and

$i \in [1..5]$. Finally, μBERT discards non-compilable mutants, and those that are syntactic the same as the original program (i.e., when <op> $= t_i$).

Figure 2 shows one example of mutants that μBERT can generated for binary expressions. Function isLeapYear returns true if a calendar year given as input is leap. One of the binary expressions to mutate is e : year % 4. To do so, μBERT masks binary operator %, leading to masked expression e' : year <mask> 4. The entire masked method is used to feed CodeBERT, for which it predicts the following 5 tokens: t_1 : ' %', t_2 : '/', t_3 : '%', t_4 : '-' and t_5 : ' /'. First notice that tokens t_1 and t_3 only differs in a space and coincides with the original token, so these mutants will be discarded. Second, tokens t_2 and t_5 are the same, except the extra space in t_5, so only one will be used for generating the mutant. Finally, μBERT produces 2 compilable mutants, based of the expressions e_2 : year / 4 and e_4 : year - 4.

```
boolean isLeapYear(int year) {
  // if the year is divided by 4
  // and not 100, except 400.
  if ((year % 4 == 0) &&
        ((year % 100 != 0) ||
         (year % 400 == 0)))
    return true;
  else
    return false;
}
```

μBERT
Expression to mutate
e : year % 4
Masked expression:
e': year <mask> 4
CodeBERT predictions:
t1 :' %' t2 :'/' t3 :'%'
t4 :'-' t5 :' /'

Fig. 2: μBERT's mutation operator for binary expressions.

B. Unary Expression Mutation

When dealing with unary expressions, μBERT distinguishes two cases, depending if the operator appears before or after the expression (e.g.++x and x--). For the sake of simplicity, consider that $e =$ <op><exp> is the unary expression to mutate. Then, μBERT will mask the operator token <op>, leading to masked expression $e' =$ **<mask>** <exp>, and the masked sequence is then fed to CodeBERT. μBERT takes CodeBERT predictions $(t_1, \ldots, t_5)$ and creates mutants $P_1, \ldots, P_5$ by replacing the unary operator <op> by the predicted tokens t_i. That is, $P_i = P[e \leftarrow e_i]$, where $e_i = t_i$ <exp> and $i \in [1..5]$. Duplicated, syntactic the same and non-compilable mutants are finally discarded.

Figure 3 shows an example of mutants that μBERT can generate for unary expressions. Function printArray prints the elements of the array arr given as input in reverse order. Consider that μBERT is going to mutate unary expression e : --i, for which it generates masked expression e' : **<mask>** i that is fed into CodeBERT. μBERT receives the following predictions: t_1 : '++', t_2 : '--', t_3 : ' --', t_4 : ' ++' and t_5 : '!'. μBERT discards mutants syntactic the same as the original (tokens t_2 and t_3), and considers two candidate mutants (t_1 and t_5), but only mutation t_1 compiles (obtaining e_1 : ++i).

C. Literal and Variable Name Mutation

This mutation is straightforward. For the sake of simplicity, consider that expression $e =$ <cons> to mutate is a

```
void printArray(String[] arr){
//print elements in reverse order
 for (int i = arr.length; --i >= 0; )
    print(arr[i]);
}
```

μBERT
Expression to mutate
e : --i
Masked expression:
e': <mask> i
CodeBERT predictions:
t1 :'++' t2 :'--' t5 :'!'
t3 :' --' t4 :' ++'

Fig. 3: μBERT's mutation operator for unary expressions.

literal (constant). μBERT starts by masking e, leading to $e' =$ **<mask>** that is used to feed CodeBERT. μBERT creates mutants $P_1, \ldots, P_5$ by replacing the mutated literal name by the predicted tokens (i.e., $P_i = P[e \leftarrow t_i]$ for $i \in [1..5]$).

Consider again function isLeapYear from Figure 2, where literal expression e : 4 is the expression to mutate (from year % 4). After replacing e with mask token, CodeBERT returns the following 5 predictions: t_1 : '4', t_2 : '100', t_3 : '400', t_4 : '10' and t_5 : '2'. Notice that, tokens t_2 and t_3 are present in the context of the mutated expression. Also note that first prediction (t_1) coincides with the original token, so it is discarded. Finally, μBERT returns 4 compilable mutants, generated by replacing the masked token with predicted tokens t_2, t_3, t_4 and t_5.

D. More Mutation Operators

μBERT is also able to mutate assignments, method calls, object field accesses, array reading and writing, and reference type expressions. Bellow we provide examples of the resulting masked sequences that μBERT generates to mutate these kind of expressions. Following the same process already described before, μBERT will generate the mutants by replacing the masked token with CodeBERT predictions. Notice that the shown predictions were observed during our experimentation, but these will likely change if are evaluated under different surrounding context.

- For an assignment expression like avg += it_result, μBERT produces the masked expression avg **<mask>**= it_result. Typical CodeBERT predictions are +, −, ⋆ and / leading to potential compilable mutants, e.g., avg -= it_result.
- In a method call expression, such as children.add(c) in Figure 4, μBERT masks the method name, producing children.**<mask>**(c). CodeBERT predicts the following method names: add, addAll, push, remove and added. μBERT discards equally the same and non-compilable mutants, obtaining two mutants: children.push(c) and children.remove(c).
- In expressions that access to particular object fields, μBERT masks the object field name. For instance, for an expression like list.head = new_node, μBERT produces the masked expression list.**<mask>** = new_node. CodeBERT predictions that we usually get cover head, next, tail, last and first.
- In array reading (and/or writing) expressions, μBERT masks the entire index used to access to the array. For instance,

```
void addChild(Composite c) {
 if (c == null)
   throw new IllegalArgumentException();
 if ((c == this) || (c.parent != null)
     || (!c.children.isEmpty()))
   throw new IllegalArgumentException();
 c.setParent(this);
 children.add(c);
 update(c);
}
```

μBERT
Expression to mutate
e : children.add(c);
Masked expression:
e': children.<mask>(c);
CodeBERT predictions:
t_1 :'add' t_2 :'addAll' t_3 :'push'
t_4 :'remove' t_5 :'added'

Fig. 4: μBERT's mutation operator for method calls.

for the expression `arr[mid-1]` in Figure 5, μBERT produces `arr[<mask>]` masked expression. Then, CodeBERT predictions are `0`, `n`, `mid`, `1` and `low`, allowing to μBERT generate 5 compilable mutants (variables `n`, `low` and `mid` are present in the context). It is worth noticing that the array name (`arr`) and the index expression `mid - 1` will be mutated by the variable name mutation operator and binary expression mutation operator, respectively.

```
int peakElement(int[] arr,int n) {
 int low=0;  int high=n-1;
 while(low<=high){
  int mid=(low+high)/2;
  if((mid==0 || arr[mid]>=arr[mid-1])
     &&(mid==n-1 || arr[mid]>=arr[mid+1]))
    return mid;
  else if(arr[mid]<=arr[mid+1])
    low=mid+1;
  else
    high=mid-1;
 }
 return -1;
}
```

μBERT
Expression to mutate
e : arr[mid-1]
Masked expression:
e': arr[<mask>]
CodeBERT predictions:
t_1 :'0' t_2 :'n' t_3 :'mid'
t_4 :'1' t_5 :'low'

Fig. 5: μBERT's mutation operator for array expressions.

- In expressions that refers to some type, such as `int number = (int)(Math.random() * 10)`, μBERT masks that class name of the referred type. In this case, μBERT produces the masked expression `int number = (int)(<mask>.random() * 10)`. For this example, predictions we obtained refer to `Math`, `random`, `Random` and `System`, leading to mutants such as `int number = (int)(Random.random() * 10)`.

IV. Research Questions

We start our analysis by investigating the fault detection capabilities of test suites designed to kill μBERT's mutants. Thus, we ask:

RQ1 *How effective are the mutants generated by μBERT in detecting real faults? How does μBERT compare with PiTest in terms of fault detection?*

To answer this question we evaluate the fault detection ability of test suites selected to kill the mutants produced by μBERT and PiTest [6], our baseline. The fault detection ability is approximated by using a set of real faults taken from Defects4J [19].

Another application case of mutation testing regards the program assertion generation. In particular, using mutation testing for selecting and discarding assertions by program assertion inference techniques. In view of this, we ask:

RQ2 *Is μBERT successful in selecting "good" assertions? How does it compare with PiTest?*

To answer this question we use a dataset composed by manually written assertions (ground-truth) that was recently used for evaluating SpecFuzzer tool [26], a state-of-the-art specification inference technique. Particularly, we select 4 manually written assertions that were mistakenly discarded by SpecFuzzer, since they do not kill any mutant. We thus, investigate whether μBERT can help in selecting these assertions and compare it with PiTest.

Finally, we qualitatively analyse some of the mutants generated with μBERT and ask:

RQ3 *Does μBERT generates different mutants than traditional mutation testing operators?*

We showcase the mutants generated by μBERT that help in detecting faults not found by PiTest, and mutants that help SpecFuzzer in preserving assertions from the ground-truth, that are discarded by mutants from PiTest.

V. Experimental Setup

A. Faults and Assertions (Ground-truth)

For the fault detection analysis, we use Defects4J [19] v2.0,0, which contains the build infrastructure to reproduce (over 800) real faults for Java programs. Every bug in the dataset consists of the faulty and fixed versions of the code and a developer's test suite accompanying the project that includes at least one fault triggering test that fails in the faulty version and passes in the fixed one. Since this is a preliminary evaluation, we target projects with low number of bugs in the dataset. Precisely, we consider a total of 40 bugs, reported for the following 3 projects: Cli (22), Collections (2) and Csv (16).

For the assertion assessment analysis, we use the dataset from SpecFuzzer, a specification inference technique recently introduced by Molina et al. [26], that includes (41) assertions manually written by developers. Each subject contains the source code, the test suite used during the inference process, and the set of manually written expected assertions. Particularly, we focus on 4 methods of the dataset (StackAr.pop, StackAr.topAndPop, Angle.getTurn and Composite.addChild) in which 6 assertions from the ground-truth are discarded since they do not kill any mutant (cf. [26, Table 4]). We study whether μBERT can help SpecFuzzer in selecting the discarded assertions, and compare with PiTest.

B. Experimental Procedure

To answer RQ1, we start by generating mutants with μBERT and PiTest for the fixed version of each fault. Table I summarises the number of mutants generated by the tools. Then, we make an objective comparison between the techniques in terms of the number of generated mutants and faults detected. We select minimal test cases, from the developer test suites, that kill the same number of mutants for both tools and check whether they detect the associated real faults or not. This is important since μBERT generates by far less mutants than PiTest. We then, perform a cost-effective analysis by simulating a scenario where a tester selects mutants based on which he designs tests to kill them. We start by taking

TABLE I: Number of (compilable) mutants generated by μBERT and PiTest for each project.

Project	μBERT	PiTest
Cli (22 bugs)	4.282	19.482
Collections (2 bugs)	280	1.162
Csv (16 bugs)	4.515	18.378
Total	**9.077**	**39.022**

the set of mutants created by a tool, randomly picking up a mutant and selecting a test that kills it or judging the mutant as equivalent and discard it. We then run this test with all mutants in the set and discarding those that are killed. We repeat this process until we reach a maximum number of mutants killed. We adopt as effort/cost metric the number of times a developer analyses mutants (either these result to a test or not). This means that effort is the number of tests selected plus the number of mutants judged as equivalent. We then check if the generated test suite detect or not the real faults. We repeat this process 100 times to reduce the impact of the random selection of mutants and killing tests on our results. This cost-effective evaluation aims at emphasising the effects of the different mutant generation approaches.

To answer RQ2, we start by generating mutants with μBERT and PiTest for the four methods under analysis. Then we run the inference tool, SpecFuzzer [26], to obtain the a set of valid assertions for the method of interest (i.e., never falsified by the test suite). SpecFuzzer then performs a mutation analysis on the inferred assertions, and discards the ones that do not kill any mutant. We confirm that the 6 assertions from the ground-truth are discarded in this process. Hence, we run again the mutation analysis of SpecFuzzer, but in this case we consider mutants from μBERT and PiTest, and analyse whether the ground-truth assertions are discarded or not.

To answer RQ3, we discuss on some examples from μBERT and the potential benefits that it can provide to mutation testing and assertion inference approaches.

C. Implementation

μBERT uses Spoon [30][1] for manipulating the Java programs. It employs the current pre-trained version of CodeBERT[2], and provides the scripts to integrate other pre-trained language models if required. The source code, a set of examples, and the results of our preliminary evaluation are publicly available at: **https://github.com/rdegiovanni/mBERT**.

VI. Experimental Results

A. RQ1: Fault Detection Analysis

Figure 6 summarises the fault detection capabilities of μBERT and PiTest. Figure 6a shows that test suites killing *all the mutants from* μBERT can detect 27 out of 40 faults (67.5%). While suites killing *all PiTest mutants* can detect 26 out of 40 faults (65.0%). There are 11 faults (27.5%) not detected neither by μBERT and PiTest. When we check for overlapping, we observe that 3 faults detected by μBERT were

[1]https://spoon.gforge.inria.fr

[2]https://github.com/microsoft/CodeBERT

TABLE II: Manually written assertions discarded by SpecFuzzer, because they do not kill any mutant [26, Tablle 4]. When SpecFuzzer uses the mutants generated by μBERT, it does not discard 3 out of the 6 valid assertions. When it uses PiTest mutants, it preserves 2 out of the 6 assertions, but it analyses many more mutants (up to 10 times).

Subject	Assertions	μBERT			PiTest		
		Suc.	#M	#K	Suc.	#M	#K
StackAr.pop	theArray[old(top)] == null		4	4		42	29
StackAr.topAndPop	theArray[old(top)] == null		6	6		46	39
Angle.getTurn	abs(res) <= 1	✓	23	23	✓	81	15
Composite.addChild	c.value == old(c.value)	✓	86	42	✓	96	52
	children == old(children)	✓					
	ancestors == old(ancestors)						

not detected by PiTest, and 2 faults detected by PiTest were not detected by μBERT. These indicate μBERT's fault detection effectiveness is comparable with the one of PiTest, and μBERT mutants can potentially complement other mutation testing techniques.

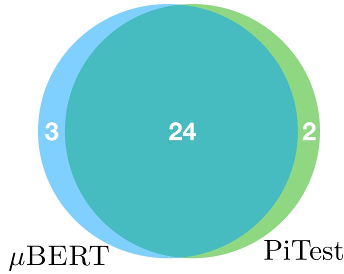
Figure 6b summarises the cost-effective evaluation of the techniques; fault detection effectiveness (y axis) in relation to the same number of analysed mutants (effort) (x axis). An effort of 100% means that the maximum possible number of mutants were analysed (for μBERT), which in the case of PiTest is the same number as by μBERT to enable a fair comparison. As Table I noted, PiTest produces way many more mutants than μBERT and thus killing all its mutants requires way more effort than μBERT. We observe that μBERT is more cost-effective, indicating that suites selected based on the mutants of μBERT are more likely to find real faults than those selected by PiTest, when the same number of mutants are analysed. Figure 6c emphasises this cost-effective comparison, and particularly focus the fault detection ratio when the maximum number of mutants was analysed (i.e., total number of mutants generated by μBERT). In average (mean), test suites killing all the mutants from μBERT have 40.0% (46.0%) of likelihood of detecting a real fault; while, suites killing exactly the same number of PiTest mutants have 20.0% (39.5%).

B. RQ2: Assertion Assessment Analysis

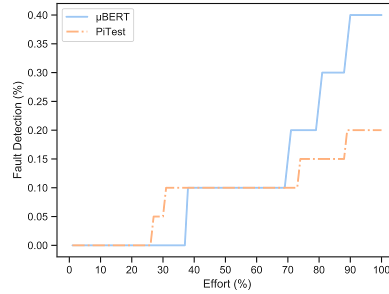
Table II summarises the performance of SpecFuzzer when uses the mutants from μBERT and PiTest for selecting the assertions. For each tool, we report if the assertions in the ground-truth were selected or discarded (Suc. column), we also report the number of generated and killed mutants (#M and #K, respectively). We can observe that 3 out of the 6 assertions under analysis, kill some mutant produced by μBERT and thus, SpecFuzzer does not discard them. In the case of PiTest, it helps in preserving 2 out of 6 assertions from the ground-truth, but in general in produces many more mutants than μBERT (e.g., up to 10 times in StackAr program) what affects the time require for filtering the assertions.

C. RQ3: Qualitative Analysis of μBERT Mutants

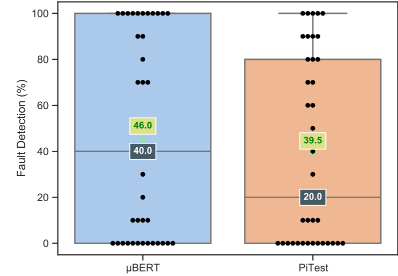
Table III shows examples of mutants produced by μBERT that help in finding the three real faults (namely, faults with ids Cli_10, Csv_15 and Csv_16) not found by PiTest. For each case, we report the diff between the fixed and the buggy



(a) μBERT detects 27 out of 40 faults (67.5%), while PiTest detects 26 (65.0%). μBERT detects 3 faults not detected by PiTest, but misses 2 faults detected by PiTest. A total of 11 faults out of 40 (27.5%) were not detected neither by μBERT and PiTest.



(b) Effort (x-axis) indicates the number of mutants analysed by a tester, while the effectiveness (y-axis) indicates the fault detection ratio of the tools. Effort of 100% means the maximum number of mutants analysed (number of tests and number mutants considered as equivalent) by a tester when using μBERT.



(c) Test suites killing all the mutants from μBERT have 40.0% (median) of likelihood of detecting a real fault (46.0% in average); while suites killing exactly the same number of PiTest mutants have 20.0% (median) of likelihood in succeeding (39.5% in average).

Fig. 6: RQ1: Fault detection comparison between the μBERT and PiTest.

version, as well as, the diff between the fixed version and the mutants generated by μBERT. Lines in red correspond to the fixed version, while lines in green correspond to the buggy version and the mutants.

The real fault denoted by Cli_10, located in file Parser.java, resides inside function `setOptions` and the problem is that it creates an aliasing between the internal object field `requiredOptions` and same field from the object `options` given as parameter. μBERT generates mutants that interact with this field trough the getter method `getRequiredOptions()`. For instance, MUTANT 1 changes an if condition regarding the size of list containing the required options. MUTANT 2 changes method call `remove` by `add`, then the list `requiredOptions` will add an element instead of removing it.

Csv_15 is a real fault inside method `printAndQuote`, located in class CSVFormat.java, in which some chars in the sequence to print were causing a failure in the parser. μBERT generates mutants that change predefined special tokens, later used to print the strings. For instance, MUTANT 3 changes the return value of function `getDelimiter()` by returning always 0, instead of the preset delimiter token. MUTANT 4 replaces object `value` with object `this` when calling `toString` in a condition that initialises the values to print.

Fault denoted as Csv_16 is present in file CVSParser.java, precisely inside class `CSVRecordIterator` that implements an iterator that returns the records of the csv. μBERT generates mutants that change the control flow of the program, for instance, the mutated expression `this.current == current` in MUTANT 5, will always evaluate to true, and MUTANT 6 introduces an infinite recursion in function `isClosed`. MUTANT 7 modifies the initialization of variable `inputClean` in method `addRecordValue`, that is later used by the iterator.

If the reader prefer, please refer to the appendix to find more examples of mutants generated by μBERT useful for detecting these faults.

Table IV shows the mutants generated by μBERT that help to SpecFuzzer to not discard good assertions, taken from the ground-truth. Particularly, the 3 mutants created for method `Angle.getTurn` clearly violate the assertion `abs(res) <= 1` and thus, it will not be discarded.

In the case of `Composite.addChild` we can observe that MUTANT 4 replaces the invocation `c.setParent(this)` by `c.update(this)`. This mutant makes that the value of the child object c (`c.value`) be updated with the value of object `this` (the parent). Then, assertion `c.value == old(c.value)` will be clearly violated by this mutant and thus, will not be discarded by SpecFuzzer.

Similarly, MUTANT 5 replaces invocation `ancestors.add(p)` by `children.add(p)`. This mutant clearly can change `children` set values. Assertion `children == old(children)` clearly kills this mutant, so SpecFuzzer will preserve it.

VII. Threats to Validity

One of the threats related to *external validity* relies on the election of the projects from Defects4J used in our evaluation (Cli, Collections and Csv). This is a preliminary study and we do not exclude the threat of having different results when conducting the same study on other projects from other domains. Other threat is related to the use of the mutation testing tool PiTest as a baseline in our experiments. Despite that this is one of the state-of-the-art tools for creating mutants, the results may change when compared with other mutant generation techniques.

Internal validity threats may relate with our implementation of μBERT. To mitigate this threat we made publicly available our implementation, repeated several times the experiments, and manually validated the results. Other threat may arise from the type of expressions selected to mutate (mutation operators), whose effectiveness can be affected when applied to other projects, or implemented in other programming language.

TABLE III: Examples of “good” mutants generated by μBERT that help in detecting the faults for Cli_10, Csv_15 and Csv_16, not found by PiTest.

BugID: Cli_10. **Class:** Parser.java

```
@@ PATCH  -44,7 +43,7 @@
- this.requiredOptions = new ArrayList(options.getRequiredOptions());
+ this.requiredOptions = options.getRequiredOptions();

@@ MUTANT 1: -306,7 +306,7 @@
- if (getRequiredOptions().size() > 0)
+ if (getRequiredOptions().size() > 1)

@@ MUTANT 2: -402,7 +402,7 @@
- getRequiredOptions().remove(opt.getKey());
+   getRequiredOptions().add(opt.getKey());
```

BugID: Csv_15. **Class:** CSVFormat.java

```
@@ PATCH -1186,7 +1186,9 @@
- if (c <= COMMENT) {
+ if (newRecord && (c < 0x20 || c > 0x21 && c < 0x23 || c > 0x2B && c < 0x2D || c > 0x7E)) {
+     quote = true;
+ } else if (c <= COMMENT) {

@@ MUTANT 3: -763,7 +763,7 @@
      public char getDelimiter() {
-         return delimiter;
+         return 0;
      }

@@ MUTANT 4: -1081,7 +1081,7 @@
- charSequence = value instanceof CharSequence ? (CharSequence) value : value.toString();
+ charSequence = value instanceof CharSequence ? (CharSequence) value : this.toString();
```

BugID: Csv_16. **Class:** CSVParser.java

```
@@ PATCH
@@ -286,7 +286,6  -355,7 +354,6  -522,10 +520,7  -573,6 +568,7 @@
-     private final CSVRecordIterator csvRecordIterator;
-         this.csvRecordIterator = new CSVRecordIterator();
      public Iterator<CSVRecord> iterator() {
-         return csvRecordIterator;
-     }
-
-     class CSVRecordIterator implements Iterator<CSVRecord> {
+         return new Iterator<CSVRecord>() {
              private CSVRecord current;
              private CSVRecord getNextRecord() {
                  throw new UnsupportedOperationException();
              }
          };
+     }

@@ MUTANT 5: -542,7 +542,7 @@
-    if (this.current == null) {
+    if (this.current == current) {

@@ MUTANT 6: -505,7 +505,7 @@
      public boolean isClosed() {
-         return this.lexer.isClosed();
+         return this.isClosed();
      }

@@ MUTANT 7: -363,7 +363,7 @@
-    final String inputClean = this.format.getTrim() ? input.trim() : input;
+    final String inputClean = "";
```

TABLE IV: μBERT generates these mutants that are killed by the ground-truth assertions and thus, SpecFuzzer does not discard them.

Subject: `Angle.getTurn`
Assertion: `abs(res) <= 1`

```
@@ MUTANT 1: -43,7 +43,7 @@
     if (crossproduct > 0) {
-         res = 1;
+         res = 2;

@@ MUTANT 2: -43,7 +43,7 @@
     if (crossproduct > 0) {
-         res = 1;
+         res = 255;

@@ MUTANT 3: -43,7 +43,7 @@
     if (crossproduct > 0) {
-         res = 1;
+         res = 360;
```

Subject: `Composite.addChild`
Assertion: `c.value == old(c.value)`

```
@@ MUTANT 4: @@ -70,7 +70,7 @@
-          c.setParent(this);
+          c.update(this);
```

Subject: `Composite.addChild`
Assertion: `children == old(children)`

```
@@ MUTANT 4: @@ -82,7 +82,7 @@
-          ancestors.add(p);
+          children.add(p);
```

To mitigate this threat, μBERT mutates expressions typically handled by mutation testing tools, such as PiTest, and it is also possible to extend our implementation to provide further mutation operators if required. The performance of CodeBERT can also affect μBERT's effectiveness. Currently, μBERT uses CodeBERT as a black-box, so it can be benefit for future improvements of the pre-trained model. Moreover, generated mutants may change if a different pre-trained model is employed for predicting the masked tokens.

Regarding construct validity threats, our assessment metrics, such as the number of analysed mutants and the number of found faults, may not reflect the actual testing cost / effectiveness values. However, these metrics have been widely used by the literature [2], [22], [29] and are intuitive, since the number of analyzed mutants essentially simulate the manual effort involved by testers, while the test suites selected to kill the mutants can also be used to measure its effectiveness in finding the fault. In our experiments, test cases were selected from the pool of tests provided by Defects4J, which may not reflect the real cost/effort in designing such test cases.

VIII. Related Work

Mutation testing has a long history with multiple advances [29], either on the faults that it injects or on the processes that it supports. Despite the rich history, the creation of "good" mutants is a question that remains.

The problem has traditionally been addressed by the definition of mutation operators using the underlying programming language syntax. These definitions span across languages [7], [8], [25], artefacts (such as specification languages and behavioural models) [14], [21], [28], and specialised applications (such as energy-aware [15] and security-aware [23] operators).

More recent attempts include the composition of mutation operators (composition of fault patterns) using historical fault fixing commits. These approaches are either mined using simple syntactic changes [4], or more complex patterns manually crafted [20], or automatically crafted patterns using machine translation techniques [32].

Independently of the way mutants are created, they are often too many to be used, with many of them being of different "quality" [27], as they are either trivially killed or simply redundant. To this end, recent attempts aim at selecting mutants that are likely killable [5], [9], [31], likely to couple with real faults [5], likely subsuming [11], [12], [18] or relevant to regression changes [24].

Our notion of mutant naturalness is somehow similar to the n-gram based notion of naturalness used by Jimenez et al. [17]. Though, we differ as we generate mutants instead of selecting and rely on a transformer-based neural architecture that captures context both before and after the mutated point.

IX. Conclusion and Future Work

We presented μBERT, a mutation testing approach that generates "natural" mutants by leveraging self-supervised model pre-training of big code. As such it does not require any training on historical faults, or other mutation testing data that are expensive to gather, but rather large corpus of source code that are easy to gather and use. Interestingly, our analysis showed that μBERT's performance is comparable with traditional mutation testing tools, and even better in some cases, both in terms of fault detection and assertion inference. These results suggests that "natural" mutants do not only concern readability but also test effectiveness. Perhaps more importantly, μBERT is the first attempt that leverage self-supervised language methods in mutation testing, thereby opening new directions for future research.

There are a few lines of future work that we plan to explore. We plan to extend our evaluation to the entire datasets of Defects4J and SpecFuzzer for analysing μBERT's fault detection and assertion inference capabilities. We also plan to include other mutation testing tools than PiTest in the comparison. So far μBERT uses CodeBERT as a black-box and mutants are generated in a brute-force way, i.e., we iterate on every program statement to mask and generate mutants. We plan to analyse CodeBERT's embedding and predictions to study whether it is possible to predict "interesting" locations to mutate, for instance, locations where *subsuming mutants* can be generated from [1], [11].

Acknowledgment

This work is supported by the Luxembourg National Research Funds (FNR) through the INTER project grant, INTER/ANR/18/12632675/SATOCROSS.

References

[1] Paul Ammann, Marcio Eduardo Delamaro, and Jeff Offutt. Establishing theoretical minimal sets of mutants. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 2014.

[2] James H. Andrews, Lionel C. Briand, Yvan Labiche, and Akbar Siami Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Trans. Software Eng.*, 32(8):608–624, 2006.

[3] Moritz Beller, Chu-Pan Wong, Johannes Bader, Andrew Scott, Mateusz Machalica, Satish Chandra, and Erik Meijer. What it would take to use mutation testing in industry - A study at facebook. In *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP)*, pages 268–277. IEEE, 2021.

[4] David Bingham Brown, Michael Vaughn, Ben Liblit, and Thomas W. Reps. The care and feeding of wild-caught mutants. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE*, pages 511–522. ACM, 2017.

[5] Thierry Titcheu Chekam, Mike Papadakis, Tegawendé F. Bissyandé, Yves Le Traon, and Koushik Sen. Selecting fault revealing mutants. *Empirical Software Engineering*, 25(1):434–487, 2020.

[6] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. PIT: a practical mutation testing tool for java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA*, pages 449–452. ACM, 2016.

[7] Márcio Eduardo Delamaro, José Carlos Maldonado, and Aditya P. Mathur. Interface mutation: An approach for integration testing. *IEEE Trans. Software Eng.*, 27(3):228–247, 2001.

[8] Lin Deng, Jeff Offutt, Paul Ammann, and Nariman Mirzaei. Mutation operators for testing android apps. *Inf. Softw. Technol.*, 81:154–168, 2017.

[9] Alejandra Duque-Torres, Natia Doliashvili, Dietmar Pfahl, and Rudolf Ramler. Predicting survived and killed mutants. In *13th IEEE International Conference on Software Testing, Verification and Validation Workshops*, pages 274–283. IEEE, 2020.

[10] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings, EMNLP*, volume EMNLP 2020 of *Findings of ACL*, pages 1536–1547. Association for Computational Linguistics, 2020.

[11] Aayush Garg, Milos Ojdanic, Renzo Degiovanni, Thierry Titcheu Chekam, Mike Papadakis, and Yves Le Traon. Cerebro: Static subsuming mutant selection. *IEEE Trans. Software Eng.*

[12] Rohit Gheyi, Márcio Ribeiro, Beatriz Souza, Marcio Augusto Guimarães, Leo Fernandes, Marcelo d'Amorim, Vander Alves, Leopoldo Teixeira, and Baldoino Fonseca. Identifying method-level mutation subsumption relations using Z3. *Inf. Softw. Technol.*, 132:106496, 2021.

[13] Rahul Gopinath, Carlos Jensen, and Alex Groce. Mutations: How close are they to real faults? In *25th IEEE International Symposium on Software Reliability Engineering, ISSRE 2014*, pages 189–200. IEEE Computer Society, 2014.

[14] Robert M. Hierons and Mercedes G. Merayo. Mutation testing from probabilistic and stochastic finite state machines. *J. Syst. Softw.*, 82(11):1804–1818, 2009.

[15] Reyhaneh Jabbarvand and Sam Malek. μdroid: an energy-aware mutation testing framework for android. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE*, pages 208–219. ACM, 2017.

[16] Gunel Jahangirova, David Clark, Mark Harman, and Paolo Tonella. Oasis: oracle assessment and improvement tool. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA*, pages 368–371. ACM, 2018.

[17] Matthieu Jimenez, Thierry Titcheu Chekam, Maxime Cordy, Mike Papadakis, Marinos Kintis, Yves Le Traon, and Mark Harman. Are mutants really natural?: a study on how "naturalness" helps mutant selection. In Markku Oivo, Daniel Méndez Fernández, and Audris Mockus, editors, *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2018, Oulu, Finland, October 11-12, 2018*, pages 3:1–3:10. ACM, 2018.

[18] Claudinei Brito Junior, Vinicius H. S. Durelli, Rafael Serapilha Durelli, Simone R. S. Souza, Auri M. R. Vincenzi, and Márcio Eduardo Delamaro. A preliminary investigation into using machine learning algorithms to identify minimal and equivalent mutants. In *13th IEEE International Conference on Software Testing, Verification and Validation Workshops, ICSTW*, pages 304–313. IEEE, 2020.

[19] René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, page 437–440, New York, NY, USA, 2014. Association for Computing Machinery.

[20] Ahmed Khanfir, Anil Koyuncu, Mike Papadakis, Maxime Cordy, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Ibir: Bug report driven fault injection, 2020.

[21] Willibald Krenn, Rupert Schlick, Stefan Tiran, Bernhard K. Aichernig, Elisabeth Jöbstl, and Harald Brandl. Momut: : UML model-based mutation testing for UML. In *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015*, pages 1–8. IEEE Computer Society, 2015.

[22] Bob Kurtz, Paul Ammann, Jeff Offutt, Márcio Eduardo Delamaro, Mariet Kurtz, and Nida Gökçe. Analyzing the validity of selective mutation with dominator mutants. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 571–582, 2016.

[23] Thomas Loise, Xavier Devroey, Gilles Perrouin, Mike Papadakis, and Patrick Heymans. Towards security-aware mutation testing. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST*, pages 97–102. IEEE Computer Society, 2017.

[24] Wei Ma, Thierry Titcheu Chekam, Mike Papadakis, and Mark Harman. Mudelta: Delta-oriented mutation testing at commit time. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 897–909. IEEE, 2021.

[25] Yu-Seung Ma, Yong Rae Kwon, and Jeff Offutt. Inter-class mutation operators for java. In *13th International Symposium on Software Reliability Engineering (ISSRE)*, pages 352–366. IEEE Computer Society, 2002.

[26] Facundo Molina, Marcelo d'Amorim, and Nazareno Aguirre. Fuzzing class specifications. In *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA*. ACM, 2022.

[27] Mike Papadakis, Thierry Titcheu Chekam, and Yves Le Traon. Mutant quality indicators. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops*, pages 32–39. IEEE Computer Society, 2018.

[28] Mike Papadakis, Christopher Henard, and Yves Le Traon. Sampling program inputs with mutation analysis: Going beyond combinatorial interaction testing. In *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST*, pages 1–10. IEEE Computer Society, 2014.

[29] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Chapter six - mutation testing advances: An analysis and survey. *Advances in Computers*, 112:275–378, 2019.

[30] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience*, 46:1155–1179, 2015.

[31] Samuel Peacock, Lin Deng, Josh Dehlinger, and Suranjan Chakraborty. Automatic equivalent mutants classification using abstract syntax tree neural networks. In *14th IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST*, pages 13–18. IEEE, 2021.

[32] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. Learning how to mutate source code from bug-fixes, 2019.

[33] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc.

APPENDIX
EXTRA EXAMPLES

TABLE V: This table presents more mutants generated by μBERT for the faults Cli_10, Csv_15 and Csv_16.

BugID: Cli_10. **Class:** Parser.java

```
@@ MUTANT: -306,7 +306,7 @@
- if (getRequiredOptions().size() > 0)
+ if (getRequiredOptions().size() > 2)

@@ MUTANT: -321,7 +321,7 @@
- throw new MissingOptionException(buff.substring(0, buff.length() - 2));
+ throw new MissingOptionException(buff.substring(0, buff.length()+2));
```

BugID: Csv_15. **Class:** CSVFormat.java

```
@@ MUTANT: -790,7 +790,7 @@
public String[] getHeaderComments() {
-        return headerComments != null ? headerComments.clone() : null;
+        return headerComments==null ? headerComments.clone() : null;
}

@@ MUTANT: -879,7 +879,7 @@
    public boolean getTrailingDelimiter() {
-        return trailingDelimiter;
+        return true;
  }

@@ MUTANT: -1081,7 +1081,7 @@
- charSequence = value instanceof CharSequence ? (CharSequence) value : value.toString();
+ charSequence = value instanceof Object ? (CharSequence) value : value.toString();

@@ MUTANT: -1726,7 +1726,7 @@
 return new CSVFormat(delimiter, quoteCharacter, quoteMode, commentMarker, escapeCharacter,
- ignoreSurroundingSpaces, ignoreEmptyLines, recordSeparator, nullString, headerComments, header,
+ ignoreSurroundingSpaces, ignoreEmptyLines, null, nullString, headerComments, header,
   skipHeaderRecord, allowMissingColumnNames, ignoreHeaderCase, trim, trailingDelimiter, autoFlush);
```

BugID: Csv_16. **Class:** CSVParser.java

```
@@ MUTANT: -362,7 +362,7 @@
-        final String input = this.reusableToken.content.toString();
+        final String input = this.toString();

@@ MUTANT: -557,7 +557,7 @@
-            if (next == null) {
+            if (current == null) {

@@ MUTANT: -363,7 +363,7 @@
-        final String inputClean = this.format.getTrim() ? input.trim() : input;
+        final String inputClean = this.format.getTrim() ? input.trim() : "";

@@ MUTANT: -463,7 +463,7 @@
-        if (formatHeader != null) {
+        if (format != null) {

@@ MUTANT: -463,7 +463,7 @@
-        if (formatHeader != null) {
+        if (this != null) {

@@ MUTANT: -546,7 +546,7 @@
-            return this.current != null;
+            return this != null;
```