Hybrid Multi-level Crossover for Unit Test Case Generation

Olsthoorn, Mitchell; Derakhshanfar, P.; Panichella, A.

DOI
10.1007/978-3-030-88106-1_6

Publication date
2021

Document Version
Accepted author manuscript

Published in
Search-Based Software Engineering - 13th International Symposium, SSBSE 2021, Proceedings

Hybrid Multi-level Crossover for Unit Test Case Generation

Mitchell Olsthoorn[0000−0003−0551−6690], Pouria Derakhshanfar[0000−0003−3549−9019], and Annibale Panichella[0000−0002−7395−3588]

Delft University of Technology, Delft, The Netherlands
M.J.G.Olsthoorn@tudelft.nl, P.Derakhshanfar@tudelft.nl, A.Panichella@tudelft.nl

Abstract. State-of-the-art search-based approaches for test case generation work at test case level, where tests are represented as sequences of statements. These approaches make use of genetic operators (*i.e.,* mutation and crossover) that create test variants by adding, altering, and removing statements from existing tests. While this encoding schema has been shown to be very effective for many-objective test case generation, the standard crossover operator (single-point) only alters the structure of the test cases but not the input data. In this paper, we argue that changing both the test case structure and the input data is necessary to increase the genetic variation and improve the search process. Hence, we propose a hybrid multi-level crossover (*HMX*) operator that combines the traditional test-level crossover with data-level recombination. The former evolves and alters the test case structures, while the latter evolves the input data using numeric and string-based recombinational operators. We evaluate our new crossover operator by performing an empirical study on more than 100 classes selected from open-source Java libraries for numerical operations and string manipulation. We compare *HMX* with the single-point crossover that is used in EvoSuite *w.r.t.* structural coverage and fault detection capability. Our results show that *HMX* achieves a statistically significant increase in 30 % of the classes up to 19 % in structural coverage compared to the single-point crossover. Moreover, the fault detection capability improved up to 12 % measured using strong mutation score.

Keywords: search-based software testing · test case generation · crossover operator · empirical software engineering

1 Introduction

Genetic operators are a fundamental component of evolutionary search-based test case generation algorithms. These operators create variation in the test cases to help the search process explore new possible paths. The main genetic operators are mutation, which makes changes to a single test case, and crossover, which exchanges information between two test cases.

Over the years, related work has used three types of encoding schemata to represent test cases for search algorithms, namely data-level, test case-level, and test suite-level. These schemata typically implement genetic operators at the same level as the encoding. For example, the crossover operator at the data-level exchanges data between two input vectors [12]. The test case-level crossover exchanges statements between two parent test cases [19]. Lastly, the test suite-level crossover swaps test cases within two test suites [10]. Recent studies have shown that the test case-level schema combined with many-objective (MO) search is the most effective at generating test cases with high coverage [6, 15].

The current many-objective approaches use the single-point crossover to recombine groups of statements within test cases. Test cases consist of both test structures (method sequences) and test data [19]. Hence, the crossover operator only changes the test structure and simply copies over the corresponding input data. Therefore, input data has to be altered by the mutation operator, usually with a small probability.

In this paper, we argue that better genetic variation can be obtained by designing a crossover operator that alters the structure of the test cases and also the input data by creating new data that is in the neighborhood of the parents' data. To validate this hypothesis, we propose a new operator, called Hybrid Multi-level Crossover (*HMX*), that combines different crossover operators on multiple levels. We implement *HMX* within EvoSuite [10], the state-of-the-art unit-test generation tool for Java.

To evaluate the effectiveness of our operator, we performed an empirical study where we compare *HMX* with the single-point crossover used in EvoSuite, a state-of-the-art test case generation tool for Java, *w.r.t.* structural coverage and fault detection capability. To this aim, we build a benchmark with 116 classes from the Apache Commons and Lucene Stemmer projects, which include classes for numerical operations and string manipulation.

Our results show that *HMX* achieves higher structural coverage for ~30 % of the classes in the benchmark. On average, *HMX*, covered 6.4 % and 7.2 % more branches and lines than our baseline, respectively (with a max improvement of 19.1 % and 19.4 %). Additionally, the proposed operator improved the fault detection capability in ~25 % of the classes with an average improvement of 3.9 % (max. 14 %) and 2.1 % (max. 12.1 %) for weak and strong mutation, respectively.

In summary, we make the following contributions:

1. A novel crossover that works at both test case and input data-level to increase genetic variation in the search process. The data-level recombination combines multiple different techniques depending on the data type.
2. An open-source implementation of our operator in EvoSuite.
3. A full replication package containing the results and the analysis scripts [13].

The outline for the remainder of this paper is as follows. Section 2 explains the fundamental concepts used in the paper. Section 3 introduces our new crossover operator, called *HMX*, and breaks down how it works. Section 4 sets out our research questions and describes the setup of our empirical study. Section 5 details our results and highlights our findings. Section 6 discusses the threats

to validity and Section 7 draws conclusions and identifies possible directions for future work.

2 Background and Related Work

Search-based unit test generation. Prior studies introduced search-based software test generation (SBST) approaches utilizing meta-heuristics (*e.g.,* genetic algorithm) to automate test generation for different testing levels [12], such as unit [10], integration [9], and system-level testing [3]. Search-based unit-test generation is one of the widely studied topics in this field, where a search process generates tests fulfilling various criteria (*e.g.,* structural coverage, mutation score) for a given class under test (CUT). Studies have shown that these techniques are effective at achieving high code coverage [6,16] and fault detection [1].

Single-objective unit test generation. Single-objective techniques specify one or more fitness functions to guide the search process towards covering the search targets according to the desired criteria. Rojas *et al.* [18] proposed an approach that aggregates all of the fitness functions for each criterion using a weighted sum scalarization and performs a single-objective optimization to generate tests. Additionally, Gay [11] empirically showed that combining different criteria in a single-objective leads to detect more faults compared to using each criterion separately.

Dynamic many-objective sorting algorithm (*DynaMOSA*). In contrast with single-objective unit test generation, Panichella *et al.* have proposed a many-objective evolutionary-based approach, called *DynaMOSA* [15]. This approach considers each coverage targets from multiple criteria as an independent search objective. *DynaMOSA* utilizes the hierarchy of dependencies between different coverage targets (*e.g.,* line, branch, mutants) to select the search objectives during the search dynamically. Moreover, recent work [17] introduced a multi-criteria variant of *DynaMOSA* that extends the idea of dynamic selection of the targets, based on an enhanced hierarchical dependency analysis. This recent study showed that this multi-criteria variant outperforms single-objective search-based unit test generation *w.r.t.* structural and mutation coverage and, therefore, can achieve a higher fault detection rate. These results have also been confirmed independently by Campos *et al.* [6]. Consequently, *DynaMOSA* is currently used as the default algorithm in EvoSuite.

Crossover operator. Like any other evolutionary-based algorithms, all variations of *DynaMOSA* need crossover and mutation operators for evolving the individuals in the current population to generate the next population. Since *DynaMOSA* encodes tests at a test case-level, the mutation operator alters statements in a selected test case according to a given *mutation probability*. This search algorithm uses the single-point crossover to recombine two selected individuals (parents) into new tests (offspring) for the next generation. This crossover operator randomly selects two positions in the selected parents and split them

Algorithm 1: *HMX*: hybrid multi-level crossover

```
   Input: Two parent test cases P1 and P2
   Output: Two offspring test cases O1 and O2
 1 begin
 2     O1, O2 ← SINGLE-POINT-CROSSOVER(P1, P2)
       // Constructor data store
 3     C1 ← Map<signature, constructor[ ]> // For P1
 4     C2 ← Map<signature, constructor[ ]> // For P2
       // Method data store
 5     M1 ← Map<signature, method[ ]> // For P1
 6     M2 ← Map<signature, method[ ]> // For P2
 7     forall (S1, S2), in S1 ∈ O1 and S2 ∈ O2 do
 8         if SIGNATURE(S1) == SIGNATURE(S2) then
 9             if S1 is constructor then
10                 C1[SIGNATURE(S1)].add(S1)
11                 C2[SIGNATURE(S2)].add(S2)
12             else if S1 is method then
13                 M1[SIGNATURE(S1)].add(S1)
14                 M2[SIGNATURE(S2)].add(S2)
15     foreach SIG ∈ C1.keys ∪ C2.keys do
           // choose random constructor with same signature
16         S1 ← random.choice(C1[SIG])
17         S2 ← random.choice(C2[SIG])
18         O1, O2 ← DATA-CROSSOVER(O1, O2, PARAMS(S1), PARAMS(S2))
19     foreach SIG ∈ M1.keys ∪ M2.keys do
           // choose random method with same signature
20         S1 ← random.choice(M1[SIG])
21         S2 ← random.choice(M2[SIG])
22         O1, O2 ← DATA-CROSSOVER(O1, O2, PARAMS(S1), PARAMS(S2))
23     return O1, O2
```

into two parts. Then, it remerges each part with the opposing part from the other parent. A more detailed explanation of this operator is available in Section 3.

While the single-point crossover brings diversity to the structure of the generated test cases, it does not work at the data-level (*i.e.*,crossover between the test inputs). Hence, this study introduces a hybrid multi-level crossover, called *HMX*, for the state-of-the-art in search-based unit test generation.

3 Approach

This section details our new crossover operator, called Hybrid Multi-level Crossover (*HMX*). This operator combines the traditional *single-point* test case-level crossover with multiple data-level crossovers.

Algorithm 1 outlines the pseudo-code of our crossover operator. *HMX* first performs the traditional *single-point* crossover at line 2. The *single-point* crossover is chosen for the test case-level operator as previous studies have shown that it is effective in producing a variation in the population over time [19]. It is also the default crossover operator used in the state-of-the-art test case generation tool EvoSuite [19]. This operator takes two parent test cases as input and selects a random point among the statements within the parents test cases. The parents are then split at this point, and their resulting parts are then recombined with its opposing part of the other parent to produce two new offspring test cases. Since these offspring test cases use a random crossover point, they might contain incomplete sequences of statements (*e.g.,* missing variable definition) and, therefore, will not compile. To make the crossover more effective, these broken references are fixed by introducing new random variable definitions that match the type of the broken reference [10]. Lines 3-22 contain the selection logic of the data-level crossover. Unlike the test case-level crossover, the data-level crossover can not be applied to every combination of input data. Performing the crossover on input data with different types (*e.g.,* strings and numbers) would not produce any meaningful output as there is no logical way to combine these dissimilar types. Furthermore, we should not perform a crossover on two identical data types from different methods. If the data-level crossover would be applied to parameters of the same type that belong to different methods, it could produce offspring that are farther from the desired objective than the original. Hence, the algorithm has to select which combinations of input data are compatible. *HMX* achieves this by selecting compatible functions (*i.e.,* constructors and methods calls) and applying the crossover pairwise to the function's parameters.

In lines 3-6, two pairs of maps are created that store the compatible functions for each parent for both constructors and methods. Each map stores a list of functions that share the same signature; The signature is the key of the map, and the functions are the values. The signature of the function is a string derived from the class name, function name, parameters types, and return type using the following format:

```
CLASS_NAME|FUNCTION_NAME(PARAM1_TYPE, PARAM2_TYPE, ...)RETURN_TYPE
```

In lines 7-14, *HMX* loops over all combinations of statements S_1 and S_2 in the offspring produced by the single-point crossover. For each combination, it checks if the signatures of the two functions match (line 8). If both statements are either constructors or methods, they are stored in their corresponding map with the signature as a key in lines 10-11 and 13-14, respectively. Note that if the test case contains constructor or method calls for other classes than the CUT, these are also considered by the selection of compatible functions. For example, additional objects (*e.g.,* strings, lists) might be needed as an input argument to one of the CUT's functions.

When all possible matching functions have been found, the operator loops through the signatures of the two function types separately in lines 15-18 and 19-22. For each signature, *HMX* selects a random function instance matching the

signature from each parent. The operator then performs the data-level crossover on the parameters of these two randomly selected functions in lines 18 and 22. For each signature in the map, *HMX* only selects one function instance per parent to proceed with the genetic recombination.

The data-level recombination pairwise traverses the parameters of the two compatible functions selected in lines 16-17 (for constructors) and 20-21 (for methods). For each pair of parameters, Algorithm 1 checks their types and determines if they are numbers or strings, the two supported types of *HMX*. If the two parameters are numbers (*i.e.,* byte, short, int, long, float, double, boolean, and char), the operator applies the *Simulated Binary Crossover* (SBX), which is described in Section 3.1. If the parameters are strings, it applies the string crossover described in Section 3.2. Lastly, in line 23, *HMX* returns the produced offspring.

Listing 1.1: Parent 1

```
1 @Test
2 public void test1() {
3   Fraction f0 = new Fraction(2, 3);
4   Fraction f1 = new Fraction(2, -1);
5   f0.divideBy(f1);
6   f0.add(Fraction.ZERO);
7 }
```

Listing 1.2: Parent 2

```
1 @Test
2 public void test2() {
3   Fraction f0 = new Fraction(3, 1);
4   Fraction f1 = new Fraction(1, 3);
5   f0.add(f1);
6   f0.pow(2.0);
7 }
```

To provide a practical example, let us consider the two parent test cases in Listings 1.1 and 1.2. Both parent 1 and parent 2 contain two invocations of the `Fraction` constructor. Since these constructors share the same signature: `Fraction|<init>(int, int)Fraction`; they are compatible. Similarly, the method `add` of the `Fraction` class is present in both parents, with the same signature: `Fraction|add(Fraction)V`; and are compatible, as well. In contrast, for example, method `divideBy`, in parent 1, and method `add`, in parent 2, are not compatible since their signatures are different.

3.1 Simulated Binary Crossover

The *Simulated Binary Crossover* (SBX) is a recombination operator commonly used in numerical problems with numerical decision variables and fixed-length chromosomes. It has been shown that Evolutionary Algorithms (EAs) that use

this crossover operator produce better results compared to traditional numerical crossover operators [8]. The equation below outlines the algorithm of *SBX*:

$$u = rand_u \tag{1}$$

$$\beta = \begin{cases} 2 \cdot u^{1/(\eta_c+1)} & \text{if } u < 0.5 \\ 1 & \text{if } u = 0.5 \\ \frac{0.5}{1.0-u}^{1/(\eta_c+1)} & \text{if } u > 0.5 \end{cases} \tag{2}$$

$$b = rand_b \tag{3}$$

$$v = \begin{cases} ((v_1 - v_2) \cdot 0.5) - (\beta \cdot 0.5 \cdot |v_1 - v_2|) & \text{if } b = true \\ ((v_1 - v_2) \cdot 0.5) + (\beta \cdot 0.5 \cdot |v_1 - v_2|) & \text{if } b = false \end{cases} \tag{4}$$

where v (Eq. (4)) is the new value of parameter v_1, v_1 is the original value of the parameter, and v_2 is the value of the opposing parameter (the corresponding parameter from the matched function). η_c is the *distribution index* and it measures how close the new values should be to original values (proximity). For *HMX*, this variable is set to 2.5 as this is within the recommended range [2;5] [8]. *SBX* first creates a random *uniform* variable u (Eq. (1)), which is used to select one of three strategies for β. This scaling variable β (Eq. (2)), is used to scale an offset. This offset is either subtracted or added depending on the random *boolean* variable b. In general, *SBX* generates new values centered around the original parents, either in between the parents' values (contracting) or outside this range (expending) depending on the value of u. The algorithm is performed on both matching parameters, and the resulting new values are used as a replacement of the original values.

As an example, consider the two compatible constructors `Fraction(2,3)` (line 3 in Listing 1.1) and `Fraction(1,3)` (line 4 in Listing 1.2). The *SBX* recombination operator is applied for the following pairwise combinations: (2, 1) and (3, 3). To calculate the new value of the first element of the first pair, $v_1 = 2$ and $v_2 = 1$. Similarly, the second element can be calculated by switching the values of v_1 and v_2. The same procedure can be applied to calculate the new values of the second pair.

3.2 String Crossover

The single-point string crossover is used to exchange information between two string parameters of matching functions [12]. By recombining parts of each string, it makes it possible for promising substrings to collect together. The operator achieves this by picking two random numbers, $0 \leq x_i < \text{length}(x)$ and $0 \leq y_i < \text{length}(y)$ for both strings, respectively. It then recombines the two strings by concatenating the substrings in the following way: $x = x[: x_i] \mid\mid y[y_i :]$ and $y = y[: y_i] \mid\mid x[x_i :]$.

For example, given the following string $x =$ "*lorem*" and $y =$ "*ipsum*" and the random variables $x_i = 1$ and $y_i = 3$, the new values will be: $x =$ "*lom*" and $y =$ "*ipsurem*".

Table 1: Projects in our empirical study. # indicates the number of CUTs. `cc` indicates the cyclomatic complexity of CUTs. σ indicates the standard deviation. `min` and `max` indicate the minimum and maximum value of the metric, respectively. Also, $\overline{\text{str-par}}$ and $\overline{\text{nr-par}}$ are the average number of string and number input parameters for the selected CUTs.

Project	#	**CCN**				**String parameter**				**Number parameter**			
		$\overline{\text{cc}}$	σ	min	max	$\overline{\text{str-par}}$	σ	min	max	$\overline{\text{nr-par}}$	σ	min	max
CLI	4	1.7	0.9	3.0	1.1	14.5	14.2	34.0	4.0	8.5	13.7	29.0	1.0
Geometry	13	1.8	0.4	2.5	1.2	3.4	5.5	21.0	1.0	10.2	6.7	21.0	1.0
Lang	34	3.0	1.6	7.4	1.1	17.4	36.7	209.0	1.0	26.6	48.3	249.0	1.0
Logging	1	3.0	-	3.0	3.0	6.0	-	6.0	6.0	3.0	-	3.0	3.0
Math	27	2.9	1.6	7.7	1.1	2.5	1.8	9.0	1.0	10.0	10.5	45.0	1.0
Numbers	5	2.8	1.1	4.5	1.6	1.4	0.9	3.0	1.0	31.6	33.5	89.0	4.0
RNG	4	3.3	1.4	5.0	1.7	2.2	2.5	6.0	1.0	2.0	1.4	4.0	1.0
Stemmer	16	1.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

4 Empirical Study

To assess the impact of *HMX* on search-based unit test generation, we perform an empirical evaluation to answer the following research questions:

RQ1 *To what extent does HMX improve structural coverage compared to the single-point crossover?*

RQ2 *How does HMX impact the fault-detection capability of the generated tests?*

Benchmark. For this study, we selected the CUTs from the Apache Commons and Snowball Stemmer libraries. The former is a commonly-used project containing reusable Java components for several applications [1]. The latter is a well-known library for stemming strings, which is part of the Apache Lucene [2]. As described in Section 3, *HMX* brings more advantages for search-based test generation in projects that utilize strings and numbers. Hence, to show the effect of this new crossover operator, we selected 100 classes from 9 components in Apache Commons that have numeric and string input data: (i) Math a library of lightweight, self-contained mathematics and statistics components; (ii) Numbers includes utilities for working with complex numbers; (iii) Geometry provides utilities for geometric processing; (iv) RNG a library of Java implementations of pseudo-random generators; (v) Statistics a project containing tools for statistics; (vi) CLI an API processing and validating a command line interface; (vii) Text a library focused on algorithms working on strings; (viii) Lang contains extra functionality for classes in `java.lang`; and (ix) Logging an adapter allowing configurable bridging to other logging systems.

[1] https://commons.apache.org
[2] https://github.com/weavejester/snowball-stemmer

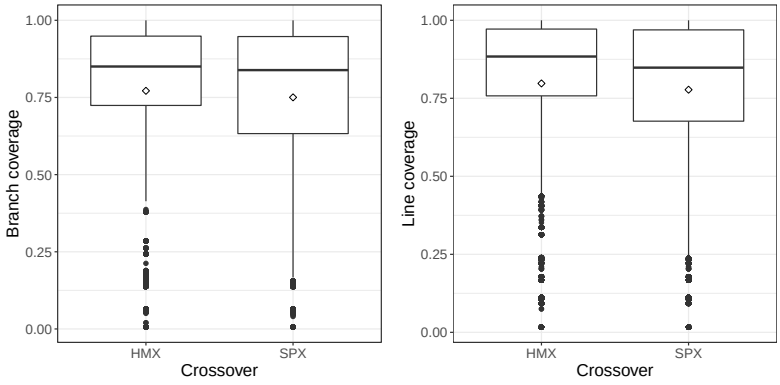
In addition, we added the main 16 classes in Snowball Stemmer to the benchmark, as these focus on string manipulation and were previously used in former search-based unit test generation studies [14].

Due to the large number of classes in the selected Apache Commons components, we used CK [2], a tool that calculates the method-level and class-level code metrics in Java projects using static analysis. We collect the Cyclomatic Complexity (CC) and type of input parameters for each method in the selected 9 components. Using the collected information, we filter out the classes that do not have methods accepting strings or numbers (integer, double, long, or float) as input parameters. Then, we sort the remaining classes according to their average CC and pick the top 100 cases for our benchmark. Table 1 reports CC, number of string, and number arguments for each project used in this study. By doing a preliminary run of EvoSuite on the 116 selected classes, we noticed that this tool fails to start the search process in 9 of the CUTs. These failures stem from an issue in the underlying test generation tool EvoSuite. The tool fails to gather a critical statistic (*i.e.,* TOTAL_GOALS) for these runs in both the baseline and *HMX*. We also encountered 4 classes that did not produce any coverage for both the baseline and our approach. Consequently, we filtered out these classes from the experiment and performed the final evaluation on 103 remaining classes.

Implementation. We implemented *HMX* in EvoSuite [10], which is the state-of-the-art tool for search-based unit test generation in Java. By default, this tool uses the single-point crossover for test generation. We have defined a new parameter `multi_level_crossover` to enable *HMX*. Our Implementation is openly available as an artifact [13].

Preliminary Study. We performed a preliminary study to see how the probability of applying our data-level crossover influences the result. The single-point test case-level crossover is applied with a predefined probability. We experimented with how often the data-level crossover should be applied whenever the test case-level crossover was applied. From the probabilities we tried (*i.e.,* 0.25, 0.50, 0.75, 1.00), we found out that always applying the data-level crossover when the test case-level crossover produced the best results according to statistical analysis.

Parameter Settings. We run each search process with EvoSuite's default parameter values. As confirmed by prior studies [5], despite the impact of parameter tuning on the search performance, the default parameters provide acceptable results. Hence, we run each search process with a two-minute search budget and set the population size to 50 individuals. Moreover, we use mutation with a probability of $1/n$ (n = length of the generated test). For both crossover operators that we used in this study (single-point crossover for the baseline and our novel *HMX*), the crossover probability is 0.75. For the Simulated Binary Crossover (SBX), we used the *distribution index* $\eta_c = 2.5$ [8]. The search algorithm is the multi-criteria DynaMOSA [17], which is the default one in EvoSuite `v1.1.0`.



(a) Branch Coverage (b) Line Coverage

Fig. 1: Boxplot of structural coverage comparing *HMX* to the baseline SPX. The diamond point indicates the mean coverage of the benchmark.

Experimental Protocol. We apply both default EvoSuite with single-point crossover and EvoSuite + *HMX* to each of the selected CUTs in the benchmark. To address the random nature of search-based test generation tools, we repeat each execution 100 times, with a different random seed, for a total number of 23 200 independent executions. We run our evaluation on a system with an AMD EPYC™ 7H12 using 240 cores running at 2.6 GHz. With each execution taking 5 minutes on average (*i.e.,* search, minimalization, and assertion generation), the total running time is 80.6 days of sequential execution.

For our analysis, we report the average (median) results across the 100 repeated runs. To determine if the results (*i.e.,* structural code coverage and fault detection capability) of the two crossover operator are statistically significant, we use the unpaired Wilcoxon rank-sum test [7] with a threshold of 0.05. The Wilcoxon test is a non-parametric statistical test that determines if two data distributions are significantly different. Additionally, we use the Vargha-Delaney statistic [20] to measure the magnitude of the result, which determines how large the difference between the two operators is.

5 Results

This section discusses the results of our study with the aim of answering the research questions formulated in Section 4. All differences in results in this section are presented in absolute differences (percentage points).

5.1 Result for RQ1: Structural Coverage

Fig. 1 shows the structural coverage achieved by our approach, *HMX*, compared to the baseline, SPX, on the benchmark. In particular, Fig. 1a shows branch coverage and Fig. 1b shows line coverage. The boxplots show the median, quartiles,

Table 2: Statistical results of *HMX* vs. SPX on structural coverage. #Win indicates the number of times that *HMX* is statistically better than SPX. #Lose indicates the opposite. #No diff. indicates that there is no statistical difference. Negl., Small, Medium, and Large denote the $\hat{A}_{12}$ effect size.

Metric	#Win				#Lose				#No diff.
	Negl.	Small	Medium	Large	Negl.	Small	Medium	Large	
Branch	2	5	3	22	0	1	0	0	70
Line	3	1	3	19	0	1	0	0	76

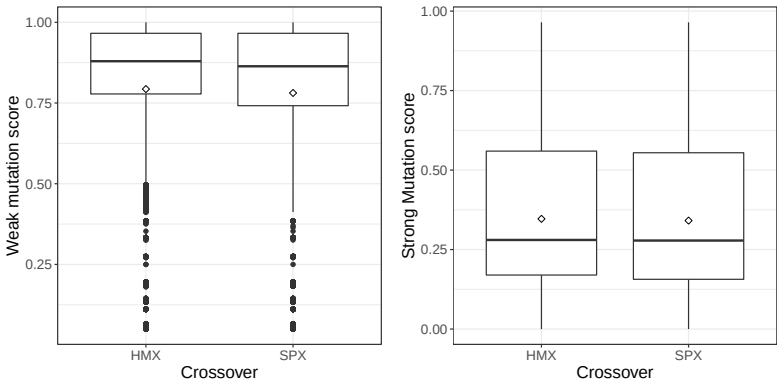
variability in the results, and the outliers for all classes together. The diamond point indicates the mean of the results.

Fig. 1a and Fig. 1b show that, on average, *HMX* has higher 1^{st} quartile, median, mean, and 3^{rd} quartile values than the baseline, SPX, for both test metrics. On average, *HMX* improves the branch coverage by +2.0 % and the line coverage by +1.9 %. The largest differences are visible for the lower whisker and for the first quartile (25th percentile). In particular, the differences for the lower whisker are around +20% branch and line coverage when using *HMX*; the improvements in the first quartile are around +10% and +8% for branch and line coverage, respectively. These results indicate that *HMX* improves both line and branch coverage for some of the CUTs in our benchmark. Finally, as we can see in both of the plots in Fig. 1, the variation in the results for *HMX*, measured by the Interquartile Range (IQR), is smaller than for SPX. This observation shows that *HMX* helps EvoSuite to generate tests with a more stable structural coverage.

Table 2 shows the results of the statistical comparison between *HMX* and the baseline, SPX, based on a p-value ≤ 0.05. *#Win* indicates the number of times that *HMX* has a statistically significant improvement over SPX. *#Equal* indicates the number of times that there is no statistical difference in the results between the two operators; *#Lose* indicates the number of times that *HMX* has statistically worse results than SPX. The *#Win* and *#Lose* columns also include the magnitude of the difference through the $\hat{A}_{12}$ effect size, classified in *Small*, *Medium*, *Large*, and *Negligible*.

From Table 2, we can see that *HMX* has a statistically significant non-negligible improvement in 30 and 23 classes for branch and line coverage, respectively. For the branch coverage metric, *HMX* improves with a large magnitude for 22 classes, medium for 3 classes, and small for 5 classes. For line coverage, *HMX* improves with a large magnitude for 19 classes, medium for 3 classes, and small for 1 class. *HMX* only loses in one case in comparison to the baseline for both branch and line coverage: `StrSubstitutor` from the `Lang` project. However, in this case, the effect size is small (magnitude).

For branch coverage, we observe a maximum increase in coverage of +19.1 % for the *finnishStemmer* class from the `Stemmer` project. For line coverage, the class with the maximum increase in coverage is `hungarianStemmer` (also from `Stemmer`) with an average improvement of +19.4 %. Compared to the baseline, all classes in the Snowball Stemmer string manipulation library improve



(a) Weak Mutation Score (b) Strong Mutation Score

Fig. 2: Boxplot of structural coverage comparing *HMX* to the baseline SPX.

Table 3: Statistical results of *HMX* vs. SPX for fault-detection capability.

Metric	#Win				#Lose				#No diff.
	Negl.	Small	Medium	Large	Negl.	Small	Medium	Large	
Weak mutation	3	3	3	21	0	1	0	0	72
Strong mutation	0	8	0	15	0	3	0	0	77

based on branch and line coverage with an average improvement of +11.4 % and +11.0 %, respectively. For the Apache Commons library, *HMX* significantly improves the branch and line coverage in 16 (9 string-related and 7 number-related) and 10 (6 string-related and 4 number-related) classes, respectively.

In summary, the proposed *HMX* crossover operator achieves significantly higher (~30 % of the cases) or equal structural code coverage for unit test case generation compared to the baseline SPX.

5.2 Result for RQ2: Fault Detection Capability

Fig. 2 shows the fault detection capability of *HMX* compared to SPX measured through the mutation score. Fig. 2a shows the weak mutation score and Fig. 2b shows the strong mutation score. The boxplots show the median, quartiles, variability in the results, and the outliers for all classes in the benchmark together. The diamond point indicates the mean of the results. From Fig. 2a, we can see that, on average, *HMX* improves the weak mutation score by +1.2 % compared to SPX. However, from Fig. 2b we can see that overall, the strong mutation scores only show marginal improvements (+0.5 %).

Table 3 shows the statistical comparison between *HMX* and SPX, based on a p-value ≤ 0.05. Similarly to Table 2, *#Win* indicates the number of times that

HMX has a statistically significant improvement over SPX, *#Equal* indicates the number of times that there is no statistical difference in the results of the two operators, and *#Lose* indicates the number of times that *HMX* has statistically worse results than SPX. The *#Win* and *#Lose* columns additionally also indicate the magnitude of the difference through the $\hat{A}_{12}$ effect size. From Table 3, we can see that *HMX* has a statistically significant non-negligible improvement in 27 and 23 cases for weak and strong mutation, respectively. For weak mutation, *HMX* improves with a large magnitude for 21 classes, medium for 3 classes, and small for 3 classes. For strong mutation, *HMX* improves with a large magnitude for 15 classes and a small magnitude for 8 classes. *HMX* performes worse in one case (`Fraction` from the `Lang` project) for weak mutation and three cases (`AdaptiveStepsizeFieldIntegrator` and `MultistepIntegrator` from the `Math` project, and `SphericalCoordinates` from the `Geometry` project) for strong mutation, all with a small effect size.

We observe a maximum increase in weak mutation score of +14.0 % for the `hungarianStemmer` class (`Stemmer`) and +12.2 % for the `ExtendedMessageFormat` class (`Text`) on strong mutation score. Among the classes that improve on weak and strong mutation score, 27 and 20, respectively, also improve *w.r.t.* branch coverage. Interestingly, four classes among both mutation scores improve *w.r.t.* mutation score without improving the structural coverage.

In summary, *HMX* achieves significantly higher (~25 % of the cases) or equal fault detection capability compared to SPX and is outperformed in one and three classes for weak and strong mutation, respectively.

6 Threats to Validity

This section discusses the potential threats to the validity of our study.

Construct validity: Threats to *construct validity* stem from how well the chosen evaluation metrics measure the intended purpose of the study. Our study relies on well-established evaluation metrics in software testing to compare the proposed hybrid multi-level crossover with the current state-of-the-art, namely structural coverage (*i.e.,* branch and line) and fault detection capability (*i.e.,* weak and strong mutation). As the stopping condition of the search process, we used a time-based budget rather than a budget based on the number of test evaluations or generations. A time-based budget provides a fairer measure since the two crossover operators have a different overhead and execution time and might otherwise provide an unfair advantage to our operator.

Internal validity: Threats to *internal validity* stem from the influence of other factors onto our results. The only difference between the two approaches in our study is the crossover operator. Therefore, any improvement or diminishment in the results must be attributed to the difference in the two crossover operators.

External validity: Threats to *external validity* stem from the generalizability of our study. We selected 116 classes from popular open-source projects based

on their cyclomatic complexity and type of input parameters to create a representative benchmark. These classes have previously been used in the related literature on test case generation [14, 15].

Conclusion validity: Threats to *conclusion validity* stem from the deduction of the conclusion from the results. To minimize the risk of the randomized nature of EAs, we performed 100 iterations of the experiment in our study with different random seeds. We have followed the recommended guidelines for running empirical experiments with randomized algorithms using sound statistical analysis as recommend in the literature [4]. We used the unpaired Wilcoxon rank-sum test and the Vargha-Delaney $\hat{A}_{12}$ effect size to determine the significance and magnitude of our results.

7 Conclusions and Future Work

In this paper, we have proposed a novel crossover operator, called *HMX*, that combines different crossover operators on both a test case-level and a data-level for generating unit-level test cases. By implementing such a hybrid multi-level crossover operator, we can create genetic variation in not only the test statements but also the test data. We implemented *HMX* in EvoSuite, a state-of-the-art Java unit test case generation tool. Our approach was evaluated on a benchmark of 116 classes from two popular open-source projects. The results show that *HMX* significantly improves the structural coverage and fault detection capability of the generated test cases compared to the standard crossover operator used in EvoSuite (*i.e.,* single-point). Based on these promising results, there are multiple potential directions for future work to explore. In this paper, we detailed the crossover operator for two types of primitive test data inputs (*i.e.,* numbers and strings). In future work, we are planning to extend this with additional operators for arrays, lists, and maps. Additionally, we want to experiment with alternative crossover operators for numbers (*e.g.,* parent-centric crossover, arithmetic crossover) and strings (*e.g.,* multi-point crossover).

Acknowledgements

We gratefully acknowledges the Horizon 2020 (EU Commission) support for the project *COSMOS*, Project No. 957254-COSMOS.

References

1. Almasi, M.M., Hemmati, H., Fraser, G., Arcuri, A., Benefelds, J.: An Industrial Evaluation of Unit Test Generation: Finding Real Faults in a Financial Application. In: 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP). pp. 263–272. IEEE (2017)
2. Aniche, M.: Ck calculator v0.0.6. https://doi.org/10.5281/zenodo.35668
3. Arcuri, A.: RESTful API automated test case generation with evomaster. ACM Transactions on Software Engineering and Methodology **28**(1), 1–37 (2019)

4. Arcuri, A., Briand, L.: A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. Software Testing, Verification and Reliability **24**(3), 219–250 (2014)
5. Arcuri, A., Fraser, G.: Parameter tuning or default values? an empirical investigation in search-based software engineering. Empirical Software Engineering **18**(3), 594–623 (2013)
6. Campos, J., Ge, Y., Albunian, N., Fraser, G., Eler, M., Arcuri, A.: An empirical evaluation of evolutionary algorithms for unit test suite generation. Information and Software Technology **104**(August), 207–235 (2018). https://doi.org/10.1016/j.infsof.2018.08.010
7. Conover, W.J.: Practical nonparametric statistics, vol. 350. John Wiley & Sons (1998)
8. Deb, K., Sindhya, K., Okabe, T.: Self-adaptive simulated binary crossover for real-parameter optimization. In: Proceedings of the 9th annual conference on genetic and evolutionary computation. pp. 1187–1194 (2007)
9. Derakhshanfar, P., Devroey, X., Panichella, A., Zaidman, A., van Deursen, A.: Towards integration-level test case generation using call site information. arXiv preprint arXiv:2001.04221 (2020)
10. Fraser, G., Arcuri, A.: Evosuite: Automatic test suite generation for object-oriented software. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. pp. 416–419. ESEC/FSE '11, ACM, New York, NY, USA (2011)
11. Gay, G.: Generating effective test suites by combining coverage criteria. In: International Symposium on Search Based Software Engineering. pp. 65–82 (2017)
12. McMinn, P.: Search-based software test data generation: A survey. Software Testing Verification and Reliability **14**(2), 105–156 (2004)
13. Olsthoorn, M., Derakhshanfar, P., Panichella, A.: Replication package of "Hybrid Multi-level Crossover for Unit Test Case Generation" (Jul 2021). https://doi.org/10.5281/zenodo.5102597
14. Panichella, A., Kifetew, F.M., Tonella, P.: Reformulating branch coverage as a many-objective optimization problem. In: 2015 IEEE 8th international conference on software testing, verification and validation (ICST). pp. 1–10. IEEE (2015)
15. Panichella, A., Kifetew, F.M., Tonella, P.: Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. IEEE Transactions on Software Engineering **44**(2), 122–158 (2017)
16. Panichella, A., Kifetew, F.M., Tonella, P.: A large scale empirical comparison of state-of-the-art search-based test case generators. Information and Software Technology **104**(June), 236–256 (2018). https://doi.org/10.1016/j.infsof.2018.08.009
17. Panichella, A., Kifetew, F.M., Tonella, P.: Incremental control dependency frontier exploration for many-criteria test case generation. In: International Symposium on Search Based Software Engineering. pp. 309–324. Springer (2018)
18. Rojas, J.M., Campos, J., Vivanti, M., Fraser, G., Arcuri, A.: Combining multiple coverage criteria in search-based unit test generation. In: Barros, M., Labiche, Y. (eds.) Search-Based Software Engineering. pp. 93–108. Springer International Publishing, Cham (2015)
19. Tonella, P.: Evolutionary testing of classes. ACM SIGSOFT Software Engineering Notes **29**(4), 119–128 (2004)
20. Vargha, A., Delaney, H.D.: A critique and improvement of the cl common language effect size statistics of mcgraw and wong. Journal of Educational and Behavioral Statistics **25**(2), 101–132 (2000)