Towards a Fault-Detection Benchmark for Evaluating Software Product Line Testing Approaches

Stefan Fischer
Institute for Software Systems Engineering
Johannes Kepler University
Linz, Austria
Stefan.Fischer@jku.at

Roberto Erick Lopez-Herrejon
Dept. Software Engineering and IT
École de technologie supérieure
Montreal, Canada
Roberto.Lopez@etsmtl.ca

Alexander Egyed
Institute for Software Systems Engineering
Johannes Kepler University
Linz, Austria
Alexander.Egyed@jku.at

ABSTRACT

Software Product Lines (SPLs) are families of related software systems distinguished by the set of features each one provides. The commonly large number of variants that can be derived from an SPL poses a unique set of challenges, because it is not feasible to test all the individual variants. Over the last few years many approaches for SPL testing have been devised. They usually select a set of variants to test based on some covering criterion. A problem when evaluating these testing approaches is properly comparing them to one another. Even though some benchmarks have been proposed, they focus on covering criteria and do not consider fault data in their analysis. Considering the dire lack of publicly available fault data, in this paper we present the first results of our ongoing project to introduce simulated faults into SPLs along with using evolutionary techniques for synthesizing unit test cases for SPL examples.

CCS CONCEPTS

• **Software and its engineering → Software product lines**; **Software testing and debugging**;

KEYWORDS

Software Product Lines, Mutation Testing

ACM Reference Format:
Stefan Fischer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2018. Towards a Fault-Detection Benchmark for Evaluating Software Product Line Testing Approaches. In *Proceedings of SAC 2018 (SAC'18).* ACM, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.475/123_4

1 INTRODUCTION

Software Product Lines (SPLs) are families of related software systems whose member variants are distinguished by the set of features they provide [4]. The application of SPL practices has shown significant technological and economic benefits [24]. *Variability* is the capacity of software artifacts to vary and its effective management and realization lies at the core of successful SPL development [27]. The possible variants that can be derived from an SPL are typically described by a variability model. However, variability makes SPL testing challenging because the number of variants in SPLs is typically large, and therefore it is infeasible to test every single variant individually.

The last few years have seen an increasing interest in SPL testing approaches [8–11]. The common thread among these approaches is to compute a subset of variants, called a *covering array*, that are tested according to a *covering criterion* that specifies which variants are selected. A vast number of different such *covering criteria* have been proposed [20]. However, with all these different approaches for SPL testing, one critical question remains unanswered: *how do they compare?* There have been attempts to answer this question with a benchmark [19]. However, they focus exclusively on coverage but do not consider fault data and its analysis. We argue that including fault data is crucial, because a testing approach should be evaluated on its ability to detect faults. Nevertheless, a testing approach that discovers all the faults may still not be preferable if its performance in other areas (e.g. runtime, covering array size) lacks far behind the others. Note that our benchmark would not render other comparison frameworks obsolete, but rather serve as a complement to them.

In this paper we present our endeavor towards creating a benchmark for evaluating the fault detection capabilities of SPL testing approaches. Due to the lack of availability of SPLs with executable tests, we devised a process to generate tests automatically. Furthermore, we implemented an algorithm to introduce mutations into SPL code, in order to simulate faults. Our final goal is to create a benchmark consisting of SPL code including tests, a coherent variability model, and a set of faults to be detected. This would allow researchers to evaluate their SPL testing approaches more thoroughly and more convincingly. With our process we can incrementally expand the benchmark with more SPLs over time.

2 BACKGROUND

In this section we provide the background and basic terminology required to describe our work, along with a running example.

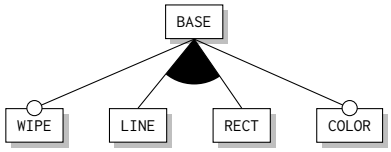


Figure 1: Feature Model for DPL

SAC'18, April 2018, Pau, France

ACM ISBN 978-1-4503-5191-1/18/04…$15.00
https://doi.org/10.475/123_4

Running Example. Our running example is the *Draw Product Line (DPL)*, an SPL of drawing applications, which contains five features. Feature BASE is the basic framework of the SPL that all variants have in common. Features LINE and RECT are responsible for drawing lines and rectangles respectively. Feature COLOR allows choosing a color to draw in and feature WIPE enables a user to wipe the drawing area clean. The variability model of our example is illustrated in Figure 1.

The most common form of a variability model is the *feature model* which is a tree-like structure where the nodes describe the features and the edges denote the different forms of relations among features [5]. In our running example, for sake of simplicity, we use two types of feature relations. The first type is *optional features* which may or may not be selected in a variant when their parent feature is selected. Examples are features COLOR and WIPE. A second type is inclusive-or groups, where at least one member of a group of features must be selected when their parent is selected. An example are features LINE and RECT.

Figure 2 shows snippets of the annotated source code of our running example. We can see that the interface of class Canvas changes depending on which features are selected. For instance, the method wipe in Line 9 is only present in variants that contain feature WIPE. Moreover, in class Rect we can see that the constructor in Line 15 changes depending if feature COLOR is selected or not.

```
1   class Canvas {
2     #ifdef $LINE
3     List<Line> lines;
4     #end
5     #ifdef $RECT
6     List<Rect> rects;
7     #end
8     #ifdef $WIPE
9     void wipe() {...}
10    #end
11    ...
12  }
13  #ifdef $RECT
14  class Rect {
15    Rect(
16      #ifdef $COLOR
17      Color c,
18      #end
19      int x, int y) {...}
20
21    int perimeter(){
22      return 2 * width + 2 * height;
23    }
24    ...
25  }
26  #end
27  ...
```

Figure 2: Code example of preprocessor annotations in DPL

Features and Feature Interactions. Within the realm of variable software there are many definitions or interpretations for the concept of features [6]. We regard features as increments in program functionality, that can be activated (i.e. selected) or deactivated (i.e. unselected). Each possible configuration of activated or deactivated features constitutes a variant of the SPL. Similarly, there are also multiple conceptions and interpretations of the concept of feature interactions [2]. Broadly speaking, a feature interaction occurs when the behavior of one feature changes depending on the presence or absence of another feature or set of features [3]. Therefore, SPL testing has to select variants that best reflect these potential *feature interactions*.

SPL Testing. Over the years, many different approaches for testing SPLs have been devised. A common thread among these approaches is the task to select variants based on a *covering criterion*. The most prominent among them are approaches based on *Combinatorial Interaction Testing* (CIT) [13, 18, 20, 21]. CIT techniques when applied to SPLs commonly use a variability model from which they calculate all valid t-wise feature-combinations which they use for computing covering arrays of a strength t. For instance for $t = 2$, also known as pairwise testing, a CIT algorithm has to find a set of variants (i.e. *covering array*) to cover all combinations of two features selected and not selected, that are allowed by the variability model. In our example this means a pairwise CIT approach has to cover 31 pairs of features, e.g. features LINE and RECT both selected, feature LINE selected and feature RECT not selected, and so on.

However, many approaches relying on other covering criteria have been proposed. To name some, Henard et al. maximize dissimilarity between randomly generated variants to approximate combinatorial coverage [14]. Javeed et al. proposed a covering criterion *decision coverage* that includes all possible branches in source code annotations [15]. For instance, for class Rect in our example, *decision coverage* would cover all possible configurations with feature RECT not selected (i.e. removing the entire class), feature RECT selected and feature COLOR not selected, and both features RECT and COLOR selected.

Mutation Testing. *Mutation testing* is generally used to evaluate the adequacy of test suites to detect faults in the code. The underlying principle of mutation testing is to introduce small changes into the program that simulate common faults [16]. These changes are introduced into the source code and each change creates a so called *mutant*. If the test results, of any test case, differ from testing the original program, the mutant has been detected (a.k.a. killed). The result of mutation testing is the *mutation score*, which is the ratio of killed mutants over the total number of mutants seeded.

Comparing Approaches. With many different SPL testing approaches available, it is important to know how well they perform compared to each other. Therefore a common benchmark that enables researchers to straightforwardly evaluate their testing approach is needed. Lopez-Herrejon et al. proposed a benchmark for comparing different CIT approaches that compute the same t-wise covering criteria using simple metrics like performance or number of variants to test [19]. Nevertheless, when comparing different covering criteria we need to be able to assess their capability to detect faults.

Ideally we would have a benchmark of SPLs with code including real tests, a variability model, and real-life faults that can be tested against. Unfortunately, publicly available SPL case studies that meet these requirements out-of-the-box are hard (if not impossible) to come by. The majority of available SPLs do not contain tests or real life faults. Therefore, in our work we attempt to automatically generate tests and faults for SPLs, that can then be used in a benchmark for comparing SPL testing approaches.

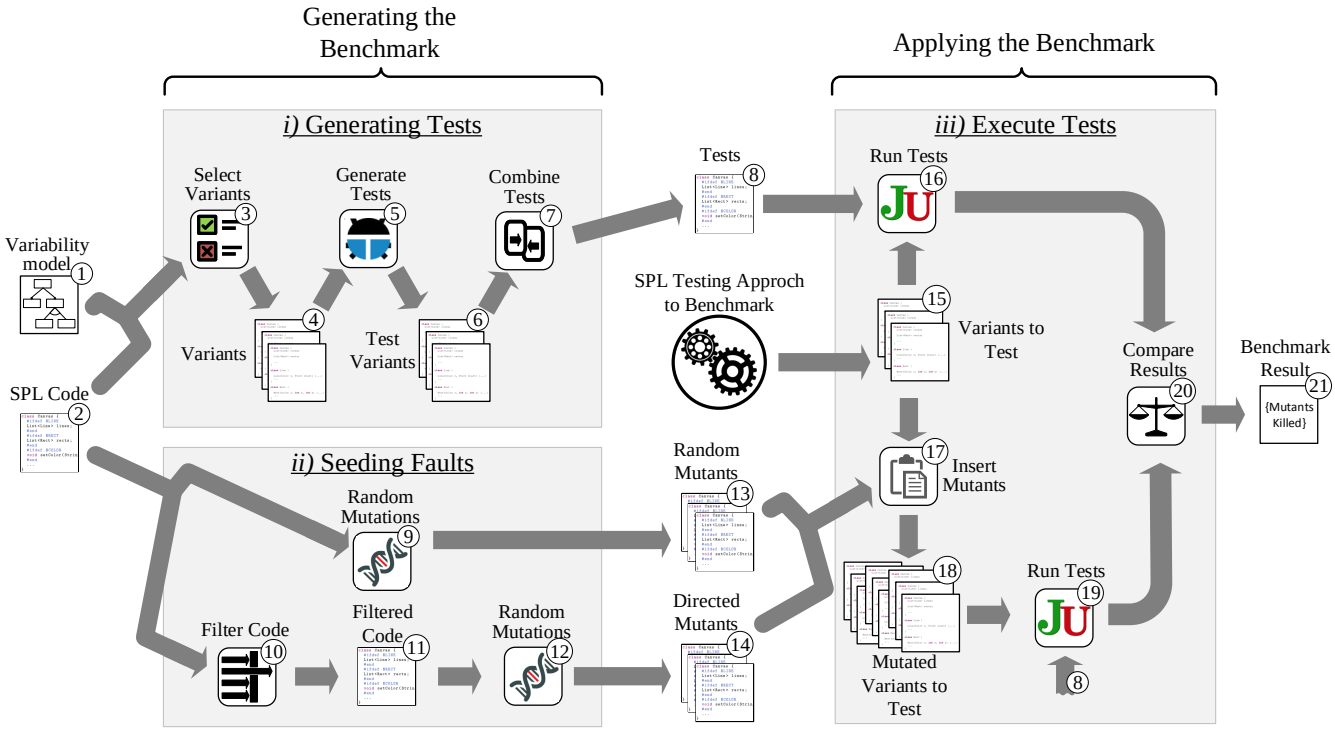


Figure 3: Process overview

3 METHODOLOGY

In this section we describe the methodology we follow for creating our benchmark and how we use it to evaluate the fault detection capabilities when testing Java SPLs. Our proposed process includes the following steps: *i)* If the SPL does not include any tests, we automatically generate them. *ii)* Subsequently, we seed faults, by mutating the SPL code. *iii)* Finally, we execute the tests on mutated SPL variants. Figure 3 outlines all the parts of our methodology. We will now go into more detail about the three components of our process.

3.1 Generating Tests

Publicly available SPLs with actual test code are rare and hard to find. Therefore, we devised a process to generate tests and use them for testing SPLs. For this test generation we use *EvoSuite*, a tool that generates tests for Java code and optimizes the whole test suites towards satisfying a code coverage criterion [12]. However, *EvoSuite* is not designed to take variability into account. Therefore, we need to generate tests for individual variants and apply them to the entire SPL, in a way that reflects the variability of the SPL (i.e. taking into account different combinations of selected and un-selected features).

The process for generating these SPL tests is illustrated in Figure 3, in the top left segment. First we need to select a subset of variants to generate tests for (3). We use the variability model (1) to compute a pairwise covering array using an algorithm similar to AETG [7]. Then, we use the SPL source code (2) for the selection of variants (3) that yields the code artifacts for the variants to generate tests for (4). Next we use *EvoSuite* (5) to generate tests for each Java class of each of these variants. This gives us tests for each individual variant (6). Figure 4 depicts two tests generated for different variants of class `Rect` of our running example, one with the feature `COLOR` selected and one without.

```
1  class Rect_ESTest {
2    test0(){
3      new Rect(color, x, y);
4      ...
5    }
6    ...
7  }
```

```
1  class Rect_ESTest {
2    test0(){
3      new Rect(x, y);
4      ...
5    }
6    ...
7  }
```

Figure 4: Test Code for Variants of Class Rect

In order to create one coherent test suite for the SPL we automatically combine all generated tests together (7). That means all test classes with equal name (i.e. testing the same source code class) are combined into one test class, in which all tests are contained. Therefore, we have to rename test methods and make sure that equal tests (i.e. containing the exact same statement in the same order) are contained only once. To account for variability we next annotate this test code, to end up with one test suite for the SPL (8). We do

this by annotating entire test methods, according to the annotations of the code they use. As an example consider Figure 5. Initially the tests are not annotated. The first test `test0` calls the constructor in class `Rect` in Line 4. Class `Rect` is only present if feature `RECT` is selected (see Line 15 in Figure 2), therefore the entire test method is annotated with feature `RECT`. Because the constructor of class `Rect` changes its signature depending if feature `COLOR` is present or not, we also have to consider the annotation in Line 16 of Figure 2. In Figure 5, the constructor call in `test0` in Line 4 contains the `color` parameter. Therefore the test also needs to be annotated with feature `COLOR` in Line 2. On the other hand, `test1` is annotated with negated feature `COLOR`, because the constructor call in Line 10 does not contain the `color` parameter.

```
1   class Rect_ESTest {
2     #ifdef $RECT && $COLOR
3     test0(){
4       new Rect(color, x, y);
5       ...
6     }
7     #end
8     #ifdef $RECT && !$COLOR
9     test1(){
10      new Rect(x, y);
11      ...
12    }
13    #end
14    ...
15  }
```

Figure 5: Annotated Test Code for Class Rect

3.2 Seeding Faults

To use the generated tests to detect faults, we need to create some faults in the SPL. We do this by introducing mutations into the code of the SPL. For this we use mutation operators that have been developed for the mutation testing tool μJava [23]. The process for this is outline in Figure 3, in the bottom left segment.

We generate two types of mutants: *i) random mutants*, and *ii) directed mutants*.

Random mutants: First we introduce random mutants (9) at arbitrary positions in the SPL source code (2). This creates random faults (13) that may get some test cases to fail.

Directed mutants: Next we introduce faults at directed positions. These faults are introduced in order to make sure variability is considered in our mutations. We filter the source code for parts that are annotated by a specific number of t features (10). For instance, we can filter for code that is annotated with two features, and find Line 17 in Figure 2. We introduce random mutants (12) at arbitrary positions in these filtered source code parts (11). As an outcome we get a set of mutated source code classes (14).

However, just like for EvoSuite, μJava does not consider variability. Nor does it allow us to control, below file level, where it will introduce mutations. This would not be a problem if we just want to generate some *random mutants*, but if we want to insert *directed mutations* it is not enough. Therefore, we implemented all method-level mutation operators, using the descriptions provided by the authors of μJava. Moreover, we implemented the required filters to identify source code that is annotated with a specific number of features.

These mutations are very simple changes to the source code. For instance, in our example a mutation would be changing the operators in the expression in method `perimeter` in class `Rect`. Figure 6 shows one such possible mutant. In Line 5 the + operator was changed to a $*$, as we highlighted and underlined. All three operators can be changed to any other arithmetic operator, and each of these changes constitutes a separate mutant. For more details on the mutation operators we refer to the μJava site[1] for further documentation.

```
1   #ifdef $RECT
2   class Rect {
3     ...
4     int perimeter(){
5       return 2 * width * 2 * height;
6     }
7     ...
8   }
9   #end
```

Figure 6: Code example for a mutations

3.3 Executing Tests

Applying our benchmark requires executing the tests. For this execution the SPL testing approach under evaluation must select a set of variants to test (see (15) in Figure 3). However, the execution of the tests (8) to detect the introduced mutants has to consider variability, because it can influence test outcomes. For instance if inside a method, that is called by the test, the implementation changes depending on the presence or absence of a feature, the result of the method call can change as well. Therefore assertions in the unit test might not hold in certain variants, and we would wrongly assume that the mutation was detected. To eliminate this we first test the un-mutated variants (16) to create a baseline for the test results. Next we insert the mutants (13) & (14) into the variants (17), by replacing the un-mutated class with the mutated one. With this, we end up with a mutated version of the selected variants for each mutant (18). Each of these mutated versions of the selected variants is then tested (19) and the results of the tests (8) are compared (20) against the test results of testing the un-mutated variant. Therefore, each variant has to be tested with each mutation inserted into it. If we discover differences in the results we consider the mutant as detected. For instance, in our running example, a test calling method `perimeter` (see Line 21 in Figure 2) does not have any problem when this method's code is not mutated. However, if the code of method `perimeter` is mutated, like for instance in Figure 6, the same test will fail, because the method call will not produce the expected result. Because, the test did not fail in the un-mutated variant, we can exclude variability as the cause for

[1] http://cs.gmu.edu/~offutt/mujava/

the test case failing, and therefore conclude that the test detected the mutant correctly. The result for our benchmark (21) is a set of detected mutants for a given SPL.

4 FIRST RESULTS

Here we present the first results of our benchmark. To date, we have found four case studies to integrate in this benchmark as shown in Table 1. These SPLs are implemented in Java, their source code and their variability model are available and are consistent.

4.1 Systems

Let us now briefly describe each system included in our benchmark so far. *DPL* is the full implementation of our running example, the Draw Product Line. *GPL* is short for the Graph Product Line and is a configurable framework of basic graph algorithms [17]. *Notepad* is a Java Swing application with functionalities similar to Windows Notepad. *VOD*[2] is a product line for video-on-demand streaming applications [22].

Table 1 lists for each of these systems the number of features, the number of possible variants according to the feature model, and the lines of code in the SPL according to the CLOC tool[3]. Moreover, we listed the number of variants used for generating tests, the number of test cases generated, and the average execution time of the tests for a variant. The experiments have been conducted on a system with an Intel Core i7-4770@3.40GHz processor, 16GB of memory, and a 64Bit environment.

We generated tests for all four systems and introduced mutations into them. For each system 30 *random* mutants and 10 *directed* mutants were introduced. *Directed* mutants were introduced to source code positions that were annotated by at least two features. DPL is an exception to this, because the source code of this system is annotated so that there was no single *directed* mutant possible with the filter used. Therefore, for DPL we only used the 30 *random* mutants.

4.2 Experiments

In order to demonstrate the usage of our benchmark, we selected variants using two different covering criteria. First, the same pairwise coverage, as we used for generating tests [7]. Second, the *decision coverage*, mentioned earlier in the paper, as proposed by Javeed et al. [15]. We performed ten individual runs for each covering criterion to account for non-determinism, and make sure we do not only get one lucky or unlucky result.

Furthermore, we wanted to check how much the selection of variants for test generation affects the quality of the generated tests. Therefore, we generated tests for all possible variants of the systems DPL and VOD, and executed the benchmark. The number of tests generated using all variants are in parentheses in Table 1 in column $|Tests|$. We selected these two systems because for them it was feasible to generate all variants, since they have a relatively low number of possible variants. Note, we computed the covering arrays once, and tested the variants with both test suites individually.

[2]http://peace.snu.ac.kr/dhkim/java/MPEG/
[3]http://cloc.sourceforge.net/

4.3 Metrics

We measured the number of mutants that were detected for each system, with each covering criterion, and with each test suite (important in the case of DPL and VOD). To check if the differences in results between different covering criteria (or different test suites) are statistically significant, we apply the Wilcoxon-Rank-sum test [26]. We used a type-one error of $\alpha = 5\%$ in the tests for significance. When comparing the benchmark results $R(X)$ for two different covering criteria (or different test suites) A and B the null hypothesis H_0 and the alternative hypothesis H_1 are:

$$H_0 : R(A) = R(B) \mid H_1 : R(A) \neq R(B)$$

Moreover, we computed the non-parametric effect size measure $\hat{A}_{12}$ as proposed by Vargha and Delaney in [28]. This $\hat{A}_{12}$ value measures the probability that the results for a metric $R(X)$ are higher for one covering criterion A versus another B, as a value between zero and one. For instance, $\hat{A}_{12} = 0.3$ means that covering criterion A results in higher values for our metric $R(X)$, 30% of the times. If the results of the two covering criteria are equal, then $\hat{A}_{12} = 0.5$. In general, if $\hat{A}_{12} < 0.5$ $R(A)$ is worse, if $\hat{A}_{12} > 0.5$ $R(B)$ is worse. The closer, $\hat{A}_{12}$ is to 0.5 the more similar the results are.

4.4 Results

Figure 7 shows the number of mutants detected by each covering criterion. We distinguish between *random* and *directed* mutants.

Random Mutants: Looking at Figure 7a and Figure 7b we can see that the number of *random mutants* that were detected in our benchmark did not change significantly with the two different covering criteria. Table 2 lists the results for the $\hat{A}_{12}$ measure, for the number of mutants detected with *pairwise* vs. *decision coverage*. For DPL the exact same mutants were detected for both covering criteria, but with on average 4 variants for *decision coverage* instead of an average of 5.9 variants for *pairwise coverage*. Similarly, our benchmark achieved the same results for *random mutants* for GPL and Notepad for both covering criteria. For GPL, one pairwise run detected an additional mutant (i.e. $\hat{A}_{12} = 0.55$ in Table 2) and for Notepad two runs detected one and two additional mutants respectively (i.e. $\hat{A}_{12} = 0.6$ in Table 2). However, the Wilcoxon-Rank-sum test shows no statistically significant difference. Moreover, the number of variants was also decreased for *decision coverage* for both systems (GPL 7.3 vs. 15.2, Notepad 4.2 vs. 13.5). For VOD we found a small advantage for using *pairwise coverage* over *decision coverage*. Here six out of the ten runs detected 14 mutants, instead of all runs with *decision coverage* only detecting 13. This is statistically significant according to the Wilcoxon-Rank-sum test ($p - value = 0.005016$). Moreover, the $\hat{A}_{12}$ value in Table 2 suggests that *pairwise coverage* yields better results in 80% of the cases. However, in the case of using the test suite that was generated from all variants, that difference becomes smaller, as seen in $\hat{A}_{12} = 0.7$ in Table 2, and therefore according to the Wilcoxon-Rank-sum test not statistically significant any more. The average number of variants for VOD are 2 for *decision coverage* vs. 7 for *pairwise coverage*. Considering that on average fewer variants are required for the *decision coverage* criterion, and that in most cases there is no significant differences in number of faults detected, it appears that the less

System	$\|F\|$	$\|V\|$	$\#LoC$	$\|V\|$ Test gen.	$\|Tests\|$	$\overline{Exec.Time}$
DPL	5	12	274	5	159 (380)	1.2s (1.6s)
GPL	28	273	922	15	584	4.7s
Notepad	25	82627	963	13	168	3.8s
VOD	11	32	4401	7	1909 (7092)	85.8s (236.2s)

$|F|$: Number of Features, $|V|$: Number of possible Variants, $\#LoC$: Number of Lines of Code,
$|V|$ Test gen.: Number of variants selected for test generation, $|Tests|$: Number of test cases generated,
$\overline{Exec.Time}$: Average execution time of the test suite for one variant

Table 1: SPL benchmark Overview

expensive *decision coverage* is preferable to *pairwise coverage* for our systems.

Directed Mutants: For GPL, Notepad, and VOD the results for *directed* mutants are exactly the same. However, when applying the test suite generated using all variants to VOD, *pairwise coverage* detects one mutant in two separate runs. This suggests that one of the *directed* mutants is only detected by a certain combination of features and by using the test suite generated from all variants. Nevertheless, the Wilcoxon-Rank-sum test reveals that the difference is not statistically significant. So again, there is no statistically significant difference detectable in the fault detection capabilities of the two covering criteria, although *decision coverage* requires significantly fewer variants to test.

Test suites: Next we present the results for comparing the different test suites generated for DPL and VOD. The results with the test suite generated using all variants are denoted with the system name with the word "FULL" added to them, in Figure 7 and in Table 2. Table 3 lists the $\hat{A}_{12}$ values for comparing the number of mutants detected for the *pairwise* generated test suite versus the test suite generated using all variants (i.e. "FULL"). We discovered that for DPL there was absolutely no difference in the results for either covering criterion. However, for VOD the Wilcoxon-Rank-sum test does suggest statistically significant differences when detecting *random* mutants using the "FULL" test suite for both covering criteria ($p - value = 0.006653$ for *pairwise coverage*, and $p - value = 1.594e^{-05}$ for *decision coverage*). This can also be observed in Figure 7a and Table 3. Nevertheless, this was not found for detecting *directed* mutants, where we detected no statistically significant difference. These results corroborate our assumption that the selection of variants for test generation, can influence the quality of the test suite.

Moreover, the execution of these "FULL" test suites takes significantly longer than the *pairwise* generated test suites, as can be seen in Table 1 in column $\overline{Exec.Time}$. This is because of the larger size of the "FULL" test suites.

System	Random Mutants	Directed Mutants
DPL	0.5	-
DPL FULL	0.5	-
GPL	0.55	0.5
Notepad	0.6	0.5
VOD	0.8	0.5
VOD FULL	0.7	0.6

Table 2: $\hat{A}_{12}$ values for number of mutants detected with *pairwise* vs. *decision coverage*

System	Random Mutants	Directed Mutants
DPL	0.5	-
VOD pairwise	0.18	0.4
VOD decision	0	0.5

Table 3: $\hat{A}_{12}$ values for number of mutants detected with *pairwise* generated tests vs. *FULL* generated tests

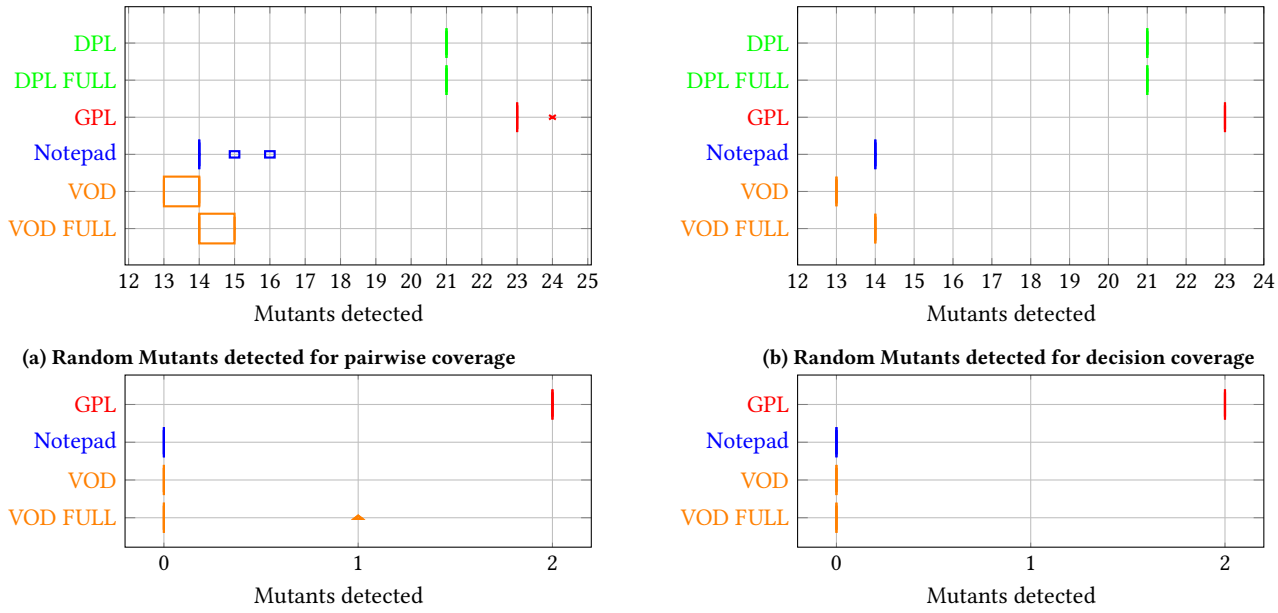
5 LIMITATIONS

During the work on creating this benchmark, we recognized some limitations and issues that should be considered when evaluating an SPL testing approach with this benchmark.

The first limitation is that we use automatically generated tests. Of course manually developed tests by domain expert would be preferable, but as we have mentioned before systems with such tests are hard to come by. We have plans in our future work, to evaluate our generated test suites, by comparing their results against manually developed tests.

The next limitation is regarding the mutations that we used in order to simulate faults. Again, real life faults would be preferable, but are hard to come by. Moreover, it can happen that some mutants may be undetectable by the tests in the SPL. On the other hand, some mutants may be killed by every possible variant in the SPL, and will therefore inflate the numbers of faults detected. Therefore, the metric of detected mutants is only expressive when comparing different approaches.

Another problem we have encountered lies in the selection of variants for generating tests (see (3) in Figure 3). We use an approach to select variants for testing to generate a test suite that is then used to evaluate an approach with the same goal of selecting variants for testing. The quality of the generated test suite will depend, at least in part, on the variants we select for the generation. We attempted to evaluate the impact the selection of variants has on the generated test suite, by devising test suites using every variant of two SPLs. For DPL selecting all the variants did not make a difference in the benchmark results. However, for VOD we did detect differences when using the two different test suites. This indicates that the selected variants for which the tests are generated



(a) Random Mutants detected for pairwise coverage

(b) Random Mutants detected for decision coverage

(c) Directed Mutants detected for pairwise coverage

(d) Directed Mutants detected for decision coverage

Figure 7: Mutants detected

can impact the benchmark results. This is an issue that we want to further investigate as part of our future work.

6 RELATED WORK

As we have mentioned before Lopez-Herrejon et al. proposed a benchmark for comparing different CIT approaches to one another [19]. Their benchmark uses metrics like performance, and the number of variants that need to be tested, to compare different approaches. These metrics are useful to assess the effort applying a testing approach, and are therefore important when comparing different approaches. The difference to our benchmark is that we want the evaluate the actual capabilities for detecting faults. Therefore, our benchmark allows researchers to compare different covering criteria to one another, in a more expressive way.

An alternative method to evaluate the fault detection capabilities of SPL testing approaches is using real reported faults. For instance, Sánchez et al. mined a list of faults from the open source web content management framework *Drupal*, using information from their issue tracking system [25]. They managed to identify 3392 faults, and report the features and interactions that are associated with these faults. Similarly, Abal et al. identified bugs in Linux, by mining bug-fixing commits to the Linux kernel repository [1]. They have since expanded their database of faults in Linux and added other systems as well. Our benchmark is different because it executes test on the systems, instead of simply checking if certain interactions, that have reportedly caused faults, are covered. We encountered some problems with using these fault databases in our work, which encouraged us to create our own benchmark in the first place. The main problem with using the database form Abal et al. for us was, that the reported faults span over many different code versions of the Linux kernel with changes in the variability model between them. This is a problem for evaluating covering arrays, because, for instance in the case of CIT, one would have to generate a covering array for each version of the variability model and could in the worst case only evaluate one fault per covering array. The reported faults for the *Drupal* system do not have this problem. However, we did not include this work in our benchmark for now, because *Drupal* is implemented in php, which made it difficult for us to apply covering criteria that analyze the source code, like the *decision coverage* criterion used in our experiments. Nevertheless, it may be valuable to include these fault databases into our benchmark in the future.

7 CONCLUSIONS AND FUTURE WORK

In this paper we presented our work for creating a benchmark for evaluating the fault detection capabilities of SPL testing approaches. Because of the lack of publicly available SPL tests and real SPL faults we devised a process to automatically generate tests and faults. We presented our preliminary results, consisting of a set of Java SPLs for which we generated tests and introduced mutations to simulate faults. Finally, we discussed some limitations we have recognized in our current benchmark, and we plan to address them in our future work.

The first item for our future work is to evaluate the quality of our generated test suites. Our plan is to expand our efforts to find SPLs that contain tests and then also generate tests for these SPL according to our described process. This would allow us to directly compare the benchmark results of our generated test suite to manually implemented tests. To this date we have only identified one SPL that includes tests, ArgoUML an open source project that has been made into a product line of UML modeling tools. Due to time constraints we were not able to include this system into our

results yet. Alternatively, we will have to derive our own manual tests to evaluate against, or find other test generation tools and compare the results.

The next item for our future work is regarding the generated mutants. We plan to introduce further steps into our process to detect and exclude equivalent mutants. As of this time we only exclude mutants if they produce the exact same source code. Furthermore, we plan to determine the quality of the generated mutants, and only include mutants that best represent real-life faults. Next we plan to to identify mutants that are either killed by every variant or by no variant at all. Because, much like equivalent mutants, these mutants only inflate the number of mutants that were discovered, or lead to interpreting undetected mutants as shortcomings in the evaluated approach respectively.

To complete our benchmark we need to include more SPLs. Therefore, one of the most important items for our future work will be to identify more SPLs and also extend our process to work for other programing languages, in oder to include SPLs implemented in languages like C/C++.

Finally, we are interested in identifying feature interactions. As we mentioned in the paper, variability can lead to test cases failing for certain variants. We plan to investigate these failing tests and find out if we can use them to identify feature interactions in the executed code parts. Knowing these interactions can be a second way to use our benchmark. Researchers could simply check if their covering arrays contain feature combinations that reveal these interactions (similar as one would use a fault database with associated interactions).

ACKNOWLEDGMENTS

We thank Gordon Fraser for his help clarifying questions about EvoSuite. Stefan Fischer is a recipient of a DOC Fellowship of the Austrian Academy of Sciences at the Institute for Software Systems Engineering. This research was partially funded by the Natural Sciences and Engineering Research Council of Canada (NSERC) grant RGPIN-2017-05421.

REFERENCES

[1] Iago Abal, Claus Brabrand, and Andrzej Wasowski. 2014. 42 variability bugs in the linux kernel: a qualitative analysis. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, Ivica Crnkovic, Marsha Chechik, and Paul Grünbacher (Eds.). ACM, 421–432.

[2] Sven Apel, Joanne M. Atlee, Luciano Baresi, and Pamela Zave. 2014. Feature Interactions: The Next Generation (Dagstuhl Seminar 14281). *Dagstuhl Reports* 4, 7 (2014), 1–24.

[3] Sven Apel, Sergiy S. Kolesnikov, Norbert Siegmund, Christian Kästner, and Brady Garvin. 2013. Exploring feature interactions in the wild: the new feature-interaction challenge. In *5th International Workshop on Feature-Oriented Software Development, FOSD '13, Indianapolis, IN, USA, October 26, 2013*. 1–8.

[4] Don S. Batory, Jacob Neal Sarvela, and Axel Rauschmayer. 2004. Scaling Step-Wise Refinement. *IEEE Trans. Software Eng.* 30, 6 (2004), 355–371.

[5] David Benavides, Sergio Segura, and Antonio Ruiz Cortés. 2010. Automated analysis of feature models 20 years later: A literature review. *Inf. Syst.* 35, 6 (2010), 615–636.

[6] Thorsten Berger, Daniela Lettner, Julia Rubin, Paul GrÃijnbacher, Adeline Silva, Martin Becker, Marsha Chechik, and Krzysztof Czarnecki. 2015. What is a Feature? A Qualitative Study of Features in Industrial Software Product Lines. In *Int'l Software Product Line Conference (SPLC'15)*.

[7] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. 2008. Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints: A Greedy Approach. *IEEE Trans. Software Eng.* 34, 5 (2008), 633–650.

[8] Paulo Anselmo da Mota Silveira Neto, Ivan do Carmo Machado, John D. McGregor, Eduardo Santana de Almeida, and Silvio Romero de Lemos Meira. 2011. A systematic mapping study of software product lines testing. *Information & Software Technology* 53, 5 (2011), 407–423.

[9] Ivan do Carmo Machado, John D. McGregor, Yguaratã Cerqueira Cavalcanti, and Eduardo Santana de Almeida. 2014. On strategies for testing software product lines: A systematic literature review. *Information & Software Technology* 56, 10 (2014), 1183–1199.

[10] Ivan do Carmo Machado, John D. McGregor, and Eduardo Santana de Almeida. 2012. Strategies for testing products in software product lines. *ACM SIGSOFT Software Engineering Notes* 37, 6 (2012), 1–8.

[11] Emelie Engström and Per Runeson. 2011. Software product line testing - A systematic mapping study. *Information & Software Technology* 53, 1 (2011), 2–13.

[12] Gordon Fraser and Andrea Arcuri. 2016. EvoSuite at the SBST 2016 tool competition. In *Proceedings of the 9th International Workshop on Search-Based Software Testing, SBST@ICSE 2016, Austin, Texas, USA, May 14-22, 2016*. ACM, 33–36.

[13] Mark Harman, Yue Jia, Jens Krinke, W. B. Langdon, J. Petke, and Y. Zhang. 2014. Search based software engineering for software product line engineering: a survey and directions for future work. In *SPLC*.

[14] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, Patrick Heymans, and Yves Le Traon. 2014. Bypassing the Combinatorial Explosion: Using Similarity to Generate and Prioritize T-Wise Test Configurations for Software Product Lines. *IEEE Trans. Software Eng.* 40, 7 (2014), 650–670.

[15] Arsalan Javeed and Cemal Yilmaz. 2015. Combinatorial interaction testing of tangled configuration options. In *Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13-17, 2015*. IEEE Computer Society, 1–4.

[16] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Trans. Software Eng.* 37, 5 (2011), 649–678.

[17] Roberto E. Lopez-Herrejon and Don S. Batory. 2001. A Standard Problem for Evaluating Product-Line Methodologies. In *GCSE (Lecture Notes in Computer Science)*, Jan Bosch (Ed.), Vol. 2186. Springer, 10–24.

[18] Roberto E. Lopez-Herrejon, Javier Ferrer, Francisco Chicano, Alexander Egyed, and Enrique Alba. 2015. *Computational Intelligence and Quantitative Software Engineering*. Springer, Chapter Evolutionary Computation for Software Product Line Testing: An Overview and Open Challenges. Accepted for publication.

[19] Roberto E. Lopez-Herrejon, Javier Ferrer, Francisco Chicano, Evelyn Nicole Haslinger, Alexander Egyed, and Enrique Alba. 2014. Towards a Benchmark and a Comparison Framework for Combinatorial Interaction Testing of Software Product Lines. *CoRR* abs/1401.5367 (2014).

[20] Roberto Erick Lopez-Herrejon, Stefan Fischer, Rudolf Ramler, and Alexander Egyed. 2015. A first systematic mapping study on combinatorial interaction testing for software product lines. In *ICST Workshops*. 1–10.

[21] Roberto E. Lopez-Herrejon, Lukas Linsbauer, and Alexander Egyed. 2015. A Systematic Mapping Study of Search-Based Software Engineering for Software Product Lines. *Journal of Information and Software Technology* (2015).

[22] Roberto E. Lopez-Herrejon, Leticia Montalvillo-Mendizabal, and Alexander Egyed. 2011. From Requirements to Features: An Exploratory Study of Feature-Oriented Refactoring. In *Software Product Lines - 15th International Conference, SPLC*. 181–190.

[23] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. 2005. MuJava: an automated class mutation system. *Softw. Test., Verif. Reliab.* 15, 2 (2005), 97–133.

[24] K. Pohl, G. Bockle, and F. J. van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer.

[25] Ana B. Sánchez, Sergio Segura, José Antonio Parejo, and Antonio Ruiz Cortés. 2017. Variability testing in the wild: the Drupal case study. *Software and System Modeling* 16, 1 (2017), 173–194.

[26] David J. Sheskin. 2007. *Handbook of Parametric and Nonparametric Statistical Procedures* (4 ed.). Chapman & Hall/CRC.

[27] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. 2005. A taxonomy of variability realization techniques. *Softw., Pract. Exper.* 35, 8 (2005), 705–754.

[28] AndrÃąs Vargha and Harold D. Delaney. 2000. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132. arXiv:https://doi.org/10.3102/10769986025002101