Measuring Effectiveness of Sample-Based Product-Line Testing

Sebastian Ruland
Real-Time Systems Lab
TU Darmstadt
Germany
sebastian.ruland@es.tu-darmstadt.de

Lars Luthmann
Real-Time Systems Lab
TU Darmstadt
Germany
lars.luthmann@es.tu-darmstadt.de

Johannes Bürdek
Real-Time Systems Lab
TU Darmstadt
Germany
johannes.buerdek@es.tu-darmstadt.de

Sascha Lity
Institute of Software Engineering and Automotive Informatics
TU Braunschweig
Germany
lity@isf.cs.tu-bs.de

Thomas Thüm
Institute of Software Engineering and Automotive Informatics
TU Braunschweig
Germany
t.thuem@tu-braunschweig.de

Malte Lochau
Real-Time Systems Lab
TU Darmstadt
Germany
malte.lochau@es.tu-darmstadt.de

Márcio Ribeiro
Computing Institute
Federal University of Alagoas
Brazil
marcio@ic.ufal.br

Abstract

Recent research on quality assurance (QA) of configurable software systems (e. g., software product lines) proposes different analysis strategies to cope with the inherent complexity caused by the well-known combinatorial-explosion problem. Those strategies aim at improving efficiency of QA techniques like software testing as compared to brute-force configuration-by-configuration analysis. Sampling constitutes one of the most established strategies, defining criteria for selecting a drastically reduced, yet sufficiently diverse subset of software configurations considered during QA. However, finding generally accepted measures for assessing the impact of sample-based analysis on the effectiveness of QA techniques is still an open issue. We address this problem by lifting concepts from single-software mutation testing to configurable software. Our framework incorporates a rich collection of mutation operators for product lines implemented in C to measure mutation scores of samples, including a novel family-based technique for product-line mutation detection. Our experimental results gained from applying our tool implementation to a collection of subject systems confirms the widely-accepted assumption that pairwise sampling constitutes the most reasonable efficiency/effectiveness trade-off for sample-based product-line testing.

CCS Concepts • **General and reference → Measurement**; *Evaluation*; • **Software and its engineering → Software product lines**; *Feature interaction*; *Software testing and debugging*;

Keywords Software Product Lines, Sample-Based Testing, Mutation Testing

ACM Reference Format:
Sebastian Ruland, Lars Luthmann, Johannes Bürdek, Sascha Lity, Thomas Thüm, Malte Lochau, and Márcio Ribeiro. 2018. Measuring Effectiveness of Sample-Based Product-Line Testing. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '18), November 5–6, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3278122.3278130

ACM ISBN 978-1-4503-6045-6/18/11…$15.00
https://doi.org/10.1145/3278122.3278130

1 Introduction

Nowaday, software becomes increasingly adaptable to the ever-growing diversity of requirements, customers' needs, and application platforms [17, 60]. Software product-line engineering [60] (SPLE) is a promising paradigm for developing highly-configurable software in an efficient way. SPLE proliferates systematic *reuse* of shared development artifacts among similar software variants, based on a common core platform. A product-line (PL) architecture consists of three essential parts [17, 60]: (1) a configuration model specifying software variability within the problem space in terms of valid *product configurations*, (2) a common code base within the solution space being customizable by *variable implementation artifacts*, and (3) a mapping between both spaces.

Besides their inherent *variability*, modern software systems are no longer isolated entities but rather embedded into larger technical—and often also safety- and security-critical—environments. Thus, quality assurance (QA), like software testing, has to be conducted with special care to reduce the risk of fatal software errors [8, 51]. However, performing exhaustive QA for a whole PL would require a complete product-by-product analysis which is highly inefficient due to the many redundant checks of shared software artifacts [18, 25, 27, 46, 51, 56]. In case of realistic PLs with thousands of configuration options, QA of all products becomes even literally impossible, due to the combinatorial-explosion problem. Hence, recent research is concerned with developing *efficient* PL analysis strategies to avoid (or reduce) computational effort of product-by-product analysis while ensuring relatively stable *effectiveness* [18, 25, 27, 46, 56, 66]. Sample-based PL testing [25, 47, 66] constitutes one of the most prominent and mature PL QA strategies, aiming at selecting a drastically reduced, yet sufficiently diverse subset of *test configurations* to be considered during QA. For instance, established (black-box) sampling heuristics apply pairwise combinatorial feature-interaction testing [25, 47] which relies on the assumption that a high fraction of failures in PL architectures is caused by erroneous interactions among at most two configuration options (so-called *features*) [44]. However, defining appropriate metrics for measuring the (potential) impact of sampling strategies on the effectiveness of analysis techniques (e. g., in terms of fault-detection probability) as compared to complete product-by-product analysis remains an open question.

Mutation testing [8, 38, 54] has recently gained growing attention in defining realistic effectiveness metrics by measuring the ability of a test suite to detect artificially seeded software faults (*mutations*) at unit-code level. However, adapting mutation-testing concepts from single-software systems to PLs is not straight-forward due to additional complexity caused by the inherent variability within PL architectures. Although preliminary work has been done in this area [4, 6, 9, 15, 20, 24, 34, 35, 45, 61], a comprehensive notion of PL mutation testing is still an open issue.

Contributions. We propose a mutation-based framework for assessing effectiveness of PL analysis strategies in general, with a special focus on sample-based testing in particular. In particular, we present an in-depth elaboration of RIPR criteria [8] and the equivalent-mutant problem [53], being one of the major pitfalls in mutation testing, in the context of sample-based PL testing. In addition, we present a novel family-based technique for PL mutation detection to avoid costly effectiveness evaluation for a given sample in a product-by-product way. Our framework facilitates the comparison of different sampling strategies w. r. t. efficiency/-effectiveness trade-offs in a fully-automated way. Although we focus on PLs implemented in C and sample-based PL testing, our methodology can be generalized to other kinds of PL implementations and analysis strategies, respectively. To summarize, we make the following contributions.

- **Concept.** We propose a mutation-based framework for family-based effectiveness measurement of PL analysis strategies.
- **Tool Support.** We present a tool implementation including a rich collection of PL mutation operators for PLs implemented in C.
- **Experiments.** We present experimental evaluation results gained from applying our tool to different subject systems to measure and compare mutation-detection scores of state-of-the-art sampling strategies. Our results confirm the generally accepted assumption that pairwise combinatorial feature-interaction testing yields the best efficiency/effectiveness trade-off for sample-based PL testing.

Verifiability. To make our results reproducible, we provide the tool implementation and all experimental results as well as raw data on a supplementary web page[1].

2 Background and Motivation

We first recapitulate basic concepts and notions of mutation testing, PL engineering and PL analysis as used throughout this paper.

2.1 Mutation Testing

Consider the C program in Fig. 1 (where the *original program* comes without lines 4 and 6). Function find_last takes as inputs an integer array x and an integer value y, and returns the position i in x of the last occurrence of y in x. If y is not contained in x, then -1 is returned.

In the following, we refer to programs by p and consider unit testing as quality-assurance (QA) technique [8]. A (unit) *test case* for a program (unit) p is given as a pair $t = (i, o)$, consisting of a *test input* i (or input-value vector) and an *expected output* o (test oracle). By $exec(t, p) = o'$ we denote the result of *executing* test case t on program p (i. e., the output o' returned by p for test input i). Program p *passes* test case t if $o = o'$ and p *fails* t, otherwise.

Example 2.1. A test case for the original program in Fig. 1 may be given as $t_1 = ((x := [0, 8, 15], y := 8), 1)$ as value 8 is located at index 1 in array x. □

[1] https://www.es.tu-darmstadt.de/gpce18/

```
1   int find_last (int[] x, int y) {
2     int pos = 0;
3     for (int i = x.length - 1; i >= 0; i--) {//original
4     for (int i = x.length - 1; i > 0; i--) { //mutant m_p
5       if (x[i] == y) { //original
6       if (x[i] <= y) { //mutant m'_p
7         pos = i;
8         return pos;
9       }
10    }
11    return -1;
12  }
```

(a) Original Program p

Figure 1. Original C Program and Two Faulty Versions obtained by replacing Line 3 with Line 4 or Line 5 with Line 6, respectively

By T_p, we denote the (potentially infinite) set of all possible test cases of program p. For non-trivial programs p with potentially a-priori unbounded input-values domains (e. g., arrays of arbitrary length), exhaustive testing is, in general, not possible as *test suites* $T = \{t_1, \ldots, t_n\} \subseteq T_p$ to be executed on p are naturally limited to finite sets of n test cases. Hence, *testing efficiency* (i. e., the effort for executing test suite T) may be measured as $\mathit{efficiency}(T) = |T|$. In contrast to testing efficiency, there is no obvious *a-priori measure* for *testing effectiveness* achievable by a given test suite T (i. e., a quantification of the ability of test cases in T to detect faults in program p). In practice, *code-coverage criteria* [8] are used to approximate effectiveness of test suites.

Example 2.2. Statement coverage is a basic, yet widely used code-coverage criterion requiring each program statement to be reached by at least one test case of a test suite T. Here, the original program in Fig. 1 requires at least two test cases to achieve statement coverage, (e. g., $t_1 = (([0, 8, 15], 8), 1)$ (covering lines 1 to 8), $t_2 = (([], 8), -1)$ (covering line 11)). However, if applied to the faulty version m_p of **`find_last`** in Fig. 1 where line 3 is replaced by 4 (containing the wrong condition **`i > 0`** instead of **`i >= 0`**), neither of the two test cases is able to reveal this error. This error can only be detected by a test case in which the index of the last position of the element to be found equals 0 (e. g., $t_3 = (([0, 8, 15], 0), 0)$). ◻

Mutation testing [8, 38, 54] has been proposed to overcome the weaknesses of purely coverage-based estimation of test-suite effectiveness [36, 42]. Mutations are supposed to mimic frequent faults of programmers (cf., e. g., Example 2.2). Particularly, mutation testing is based on the *coupling hypothesis* [19], stating that every serious fault can be emulated through a small number of locally restricted syntactic changes. Hence, a *mutation* constitutes a small transformation of an original program, usually applied at the level of single statements or expressions, resulting in a valid (i. e., compilable), yet slightly different program (*mutant*).

Let m_p denote a mutant derived from a given program p. Test case $t = (i, o)$ *detects* (or *kills*) m_p if $\mathit{exec}(t, p) \neq \mathit{exec}(t, m_p)$. In this regard, original program p serves as (correct) *specification* (i. e., $\mathit{exec}(t, p) = o$) and mutant m_p as (error-prone) implementation (i. e., $\mathit{exec}(t, m_p) \neq o$) under test. Thus, test suite T *detects* a mutant m_p if T contains at least one test case $t \in T$ detecting m_p. To mimic a wide range of possible faults, a large set M_p of mutants is usually derived from p, by applying a collection of different *mutation operators* to all locations in p to which the respective operators are applicable. A comprehensive list of mutation operators for C programs has been collected by Agrawal et al. [2]. Most of these operators change one occurrence of a particular relational, logic or arithmetic operator within a single statement or expression of p.

Example 2.3. For instance, **>=** may be replaced by **>** to mutate the original C program in Fig. 1 into the faulty program m_p. Alternatively, we may change **==** in line 5 to **<=**, resulting in the mutant m'_p having line 6 instead of 5. ◻

Hence, *effectiveness* of test suite T may be measured as *mutation-detection rate* (*mutation score* [8, 38, 54]):

$$\mathit{effectiveness}(T) = \frac{|\{m_p \in M_p \mid T \text{ detects } m_p\}|}{|M_p|}.$$

Example 2.4. Let set M_p for program p in Fig. 1 to simply contain m_p, test suite $T = \{t_1, t_2, t_3\}$ from Example 2.2 has effectiveness 1 as t_3 is able to detect m_p (i. e., $\mathit{exec}(t_3, p) = 0 \neq \mathit{exec}(t_3, m_p) = -1$). However, if M_p further comprises m'_p, effectiveness of T is 0.5 as m'_p is not detected by T. ◻

More precisely, test case $t \in T_p$ is able to detect mutant m_p if it satisfies the **RIPR** criteria [8].

- **Reachability (R):** The mutated program location is reached during execution of t on m_p.
- **Infection (I):** Reaching the mutated program location in m_p leads to an incorrect program state.
- **Propagation (P):** The incorrect program state affects the further execution of t until termination.
- **Reveal (R):** The propagated incorrect program state leads to an observable erroneous output.

Example 2.5. Consider the original program p in Fig. 1 and mutant m_p. Test case $t_2 = (([], 8), -1)$ does not satisfy **Reachability** as the statements within the loop are never executed. In contrast, $t_1 = (([0, 8, 15], 8), 1)$ satisfies **Reachability**, but not **Infection** as the first element in the array is not accessed. Test case $t_4 = (([0, 8, 15], 42), -1)$ satisfies **Reachability** and **Infection** but not **Propagation**, whereas $t_3 = (([0, 8, 15], 0), 0)$ is a **RIPR** test case as it produces the erroneous output -1 instead of 1. ◻

One essential drawback of applying *mutation-detection rate* as effectiveness measure is the *equivalent-mutant problem* [53]. A mutant m_p is *equivalent* to original program p, if no **RIPR** test case $t \in T_p$ exists for detecting m_p and should therefore

```
int find_configurable (int[] x, int y) {
  int pos = 0;
  #ifdef ISEMPTY
  if(sizeof(x) / sizeof(int) == 0)
    return -1;
  #endif
  #ifdef COUNT
  int c = 0;
  #endif
  #ifdef FIRST || COUNT
  for (int i = 0; i <= x.length - 1; i++) {
  #elif LAST
  for (int i = x.length - 1; i >= 0; i--) {
  #endif
    if (x[i] == y) {
      #ifdef COUNT
      c++;
      #else
      pos = i;
      return pos;
      #endif
    }
  }
  #ifdef COUNT
  if(c > 0)
    return c;
  else
    return 0;
  #else
    return -1;
  #endif
}
```

Figure 2. PL Implementation in C

be left out of set M_p. Unfortunately, it is, in general, undecidable whether two programs (here: p and m_p) have equivalent input-output behaviors (i. e., whether $T_p = T_{m_p}$ holds).

Example 2.6. Let us mutate line 2 in the original program in Fig. 1 to `pos = 42`. This results in an equivalent mutant as the value of `pos` is overwritten in line 7 such that the faulty value is never propagated. □

However, as mutation testing is usually used for *comparing* effectiveness of different testing techniques against each other, the resulting uncertainty of mutation scores is often negligible [15, 38, 48, 53, 54, 58].

2.2 Product Line Engineering and Analysis

The notions discussed so far are only concerned with testing programs p existing in singleton variants. In contrast, PL implementations consist of a set of similar, yet well-distinguished *program configurations*.

2.2.1 Product Line Implementations

Consider the program in Fig. 2 extending the original program from Fig. 1 by additionally configurable functionality. Compile-time variability [68] in terms of `#ifdef` directives for C preprocessor is used to associate variable code artifacts with (combinations of) configuration options. In the context of PL engineering, the configuration options are referred to as *features*. The configurable program in Fig. 2 thus further implements an optional emptiness check on input array x (feature `ISEMPTY`). In addition, the alternative features `FIRST` and `LAST` allow to configure whether the input array x should be traversed from the first to the last element or vice versa. As a third option, if feature `COUNT` is selected, the function will return the number of occurrences of y in array x instead of the index of its first occurrence.

Constraints among features (e. g., mutual exclusions of `FIRST`, `LAST` and `COUNT`) are usually specified in a *feature model* [12, 43], e. g., represented by a propositional formula:

$$\mathtt{FIND} \land (\mathtt{ISEMPTY} \lor \neg\mathtt{ISEMPTY}) \land ((\mathtt{FIRST} \land \neg\mathtt{LAST} \land \neg\mathtt{COUNT})$$
$$\lor(\neg\mathtt{FIRST} \land \mathtt{LAST} \land \neg\mathtt{COUNT}) \lor (\neg\mathtt{FIRST} \land \neg\mathtt{LAST} \land \mathtt{COUNT})).$$

Here, features *FIRST*, *LAST* and *COUNT* are *alternatives* from which exactly one feature has to be selected. In contrast, *ISEMPTY* is an *optional feature* which may be selected independently from the other features. In the following, we refer to the set of valid *product configurations* of PL *spl* over a set of features F as $C_{spl} \subseteq 2^F$. Hence, a valid *product configuration* $c \in C_{spl}$ corresponds to those subsets of features satisfying to the constraints of the feature model. For instance, the above feature model is defined on a set $F = \{\mathtt{FIND}, \mathtt{FIRST}, \mathtt{LAST}, \mathtt{COUNT}, \mathtt{ISEMPTY}\}$ of five features and defines a set C_{spl} of six valid configurations: $c_1 = \{\mathtt{FIND}, \mathtt{FIRST}\}$, $c_2 = \{\mathtt{FIND}, \mathtt{LAST}\}$, $c_3 = \{\mathtt{FIND}, \mathtt{COUNT}\}$, $c_4 = \{\mathtt{FIND}, \mathtt{FIRST}, \mathtt{ISEMPTY}\}$, $c_5 = \{\mathtt{FIND}, \mathtt{LAST}, \mathtt{ISEMPTY}\}$, and $c_6 = \{\mathtt{FIND}, \mathtt{COUNT}, \mathtt{ISEMPTY}\}$. The PL implementation p_{spl} in Fig. 2 allows to derive *program configurations* $p_c = [\![p_{spl}, c]\!]$ for product configurations $c \in C$, containing those program artifacts from p_{spl} whose *presence conditions* (i. e., `#ifdef` conditions over features) are satisfied by c. For instance, the original program in Fig. 1 corresponds to the program configuration of the PL in Fig. 2 obtained for configuration c_2.

2.2.2 Sample-Based Product Line Analysis

Exhaustive QA of a PL implementation p_{spl} by unit testing would require to execute on *every* program configuration $p_c = [\![p_{spl}, c]\!]$ corresponding to a valid product configuration $c \in C_{spl}$ a (configuration-specific) test suite $T_c \subseteq T_{p_c}$. Such a *product-by-product* QA strategy is not feasible in practice [18, 25, 27, 46, 56, 66], mostly for two reasons.

- The high amount of similarity among different program configurations potentially leads to a high number of *redundant test-case executions* (i. e., given two configurations c, c', the number of common test cases in the intersection $T_{p_c} \cap T_{p'_c}$ is presumably very high).
- The number of possible product configurations C_{spl} grows exponentially in the number $|F|$ of features thus leading to the well-known *combinatorial-explosion problem* (i. e., every additional optional feature in F may double the number of configurations in C_{spl}).

PL analysis strategies [18, 27, 56, 66] aim at counteracting those problems by improving efficiency of PL QA while keeping effectiveness relatively stable. *Sample-based analysis* [25, 47, 66] constitutes one of the most established approaches in practice. A *sample* $S \subseteq C_{spl}$ is a preferably small, yet sufficiently diverse subset of *test configurations* to be analyzed in a product-by-product manner instead of the entire set C_{spl}. A widely used sampling criterion is τ-wise combinatorial feature-interaction coverage [25, 47] requiring every valid combination of selections and deselections of τ features to be contained in at least one test configuration $c \in S$.

Example 2.7. The sample $S = \{c_1, c_2, c_6\}$ satisfies 1-wise combinatorial feature-interaction coverage on our running example as all valid selection- and deselection-decisions for every single feature are covered by at least one configuration in S. Hence, sample S is able to find the fault m_p shown in Fig. 1 as a RIPR test case for the respective mutation revealing this fault can be executed on test configuration p_{c_2} with $c_2 \in S$. However, if we insert **pos++** between line 4 and 5, the resulting mutant is not detectable by test configurations in S as line 5 contains a return statement not using **pos**. Instead, a sample S' satisfying 2-wise (pairwise) coverage is guaranteed to contain at least one test configuration on which this mutant is detectable (i. e., c_4 or c_5). □

As demonstrated by the previous example, the effectiveness of a sample S not only depends on the value chosen for τ, but also on the different possible kinds of faults potentially emerging in a PL implementation thus making any a-priori assessment of the expected effectiveness of different sampling criteria a challenging task.

3 Product-Line Mutation Testing

We now present our PL mutation-testing framework together with a family-based mutation-detection technique for measuring the effectiveness of sample-based PL testing in an automated way.

3.1 Overview

Figure 3 provides a conceptual overview of our PL mutation-testing framework. Given a PL consisting of a feature model and a PL implementation p_{spl}, the set of valid product configurations (here: $C_{spl} = \{c_1, c_2, c_3, c_4\}$) corresponds to a set of program configurations (here: $p_{c_1}, p_{c_2}, p_{c_3}$, and p_{c_4}, respectively) as described above. By applying *PL mutation operators* to p_{spl}, a set of PL mutants (here $M_{spl} = \{m^1_{spl}, m^2_{spl}, m^3_{spl}\}$) is derived from p_{spl} for measuring effectiveness of a given sample (here: $S = \{c_2, c_3\}$). In a product-based approach, those program configurations corresponding to product configurations contained in a given sample are derived, one after another, from each generated PL mutant. If the program configuration derived from the mutated PL implementation is *not* (semantically) equivalent to the program configuration derived from the original PL implementation for the same configuration, then this particular configuration in the sample is capable of detecting this mutation (denoted

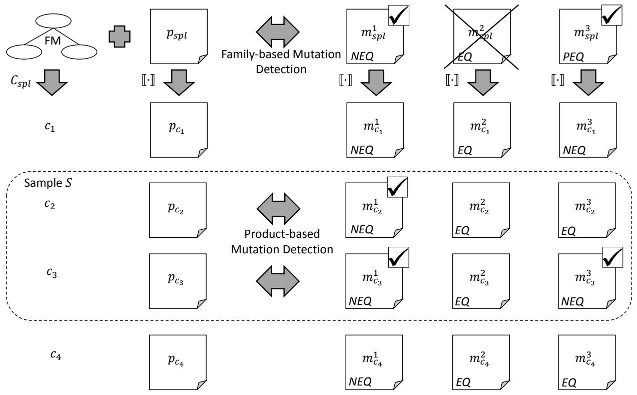


Figure 3. PL Mutation-Testing Framework

as a check-mark in Fig. 3). To generalize, we consider three different kinds of PL mutants.

- A PL mutant is *non-equivalent* (*NEQ*) if every configuration derived from the mutated PL implementation is non-equivalent (e. g., m^1_{spl} in Fig. 3). Hence, every non-empty sample may detect non-equivalent mutants.
- A PL mutant is *equivalent* (*EQ*) if no program configuration derived from the mutated PL implementation is equivalent (e. g., m^2_{spl} in Fig. 3). Hence, no sample is able to detect equivalent mutants.
- A PL mutant is *partially-equivalent* (*PEQ*) if it is neither *NEQ* nor *EQ* (e. g., m^3_{spl} in Fig. 3). Hence, a sample may detect a partially-equivalent mutant if it contains at least one non-equivalent test configuration.

Consequently, only PL mutants being *PEQ* provide a meaningful basis for measuring effectiveness of sample-based PL testing, whereas PL mutants being *EQ* and *NEQ* are negligible. For instance, if sample S in Fig. 3 contains c_4 instead of c_3, then S would fail to detect mutant m^3_{spl}.

We next describe our framework in more detail together with a family-based algorithm for detecting PL mutants.

3.2 Product-Line Mutation Operators

We first describe a collection of mutation operators for PL implementations in C, categorized into operators for common code mutation and PL-specific operators for mutating feature-to-code mappings. Note that we do not consider operators for feature-model mutation [9, 34, 45, 61], but instead refer to recent works in this area (cf. Sect. 5). Alternatively, we may also simply embed all feature-model constraints as compound propositional formula [11] into the PL implementation which allows us to utilize mapping-mutation operators also for feature-model mutation.

As depicted in Fig. 3, given an original PL implementation p_{spl}, we refer to a *PL mutant* resulting from applying a mutation operator to p_{spl} as m_{spl}. Correspondingly, we denote by M_{spl} a set of PL mutants resulting from applying a set of mutation operators to every code artifact of p_{spl} to which the respective operators are applicable.

Code-Mutation Operators. For the mutation of source-code artifacts, we consider established single-software mutation operators working at the level of conditional expressions and assignment statements [2, 8, 38]. In particular, our framework incorporates 77 C-code mutation operators as defined by Agrawal et al. [2] and implemented in the C-code mutation tool MiLu [37], which we use as basis for our tool (cf. Sect. 4). These standard C mutation operators range from replacing memory allocation calls (e. g., replacing occurrences of `malloc` by `alloca`), over replacing logic-, arithmetic-, and relational-operators (e. g., replacing occurrences of `+` by `/`), to insertions of pre-/postfix in-/decrements (e. g., changing occurrences of variables `i` to `i++` or `--i`). However, those operators are not directly applicable to also mutate feature-to-code mappings by means of presence conditions. To this end, our framework incorporates further mapping-mutation operators particularly tailored to handle feature-based variability encoded by presence conditions within C code.

Mapping-Mutation Operators. We adopt existing mutation operators, originally being applicable for replacement- and insertion-transformations of conditional expressions, for mutating presence conditions which results in five additional *mapping-mutation operators*. All these operators mimic different kinds of variability errors in terms of erroneous feature-mappings onto conditional code. Table 1 provides an overview on the different categories of mutation operators included in our framework, grouped into *replacement*, *insertion*, and *other*.

Feature Replacement: Replacement of an occurrence of a feature-variable name $f \in F$ in a presence condition by another feature-variable name $f' \in F$ from the set F of all feature names defined in the feature model.

Feature Deletion: Deletion of an occurrence of a feature variable from a presence condition by replacing the variable by either the constant `TRUE` or `FALSE`.

Logical Operator Replacement: Replacement of an occurrence of a binary logical operator in a presence condition by another binary logical operator. As presence conditions are, in most cases, given in CNF or DNF format, we can limit our considerations to replacing && by || and vice versa.

Presence Condition Negation, Feature Negation: Insertion and deletion of occurrences of negation operators in presence conditions, where either the entire condition is affected (e. g., `f∧f'` becomes `!(f∧f')` or vice versa) or one particular feature variable (e. g., `f∧f'` becomes `!f∧f'`).

Our framework is currently limited to variability by means of Boolean features and corresponding presence conditions denoted as propositional formulae over Boolean feature variables. In contrast, extended feature models [12] further support the definition of non-Boolean attributes for features which may also occur in presence conditions. However, our framework may be extended, accordingly, by adopting further mutation operators including relational- and arithmetic-operators as summarized in Table 1.

3.3 Family-Based Product-Line Mutant Detection

Based on the foundations of mutation testing and PL engineering as described in Sect. 2, we are now able to lift the respective notions of mutation detection to sets of mutations M_{spl} derived from a PL implementation p_{spl} using the mutation operators as described before. A PL mutant $m_{spl} \in M_{spl}$ of p_{spl} is *detectable* if at least one configuration $c \in C_{spl}$ exists such that the program configuration $p_c = [\![p_{spl}, c]\!]$ derived from p_{spl} for configuration c enables the detection of the respective mutant $m_c = [\![m_{spl}, c]\!]$ derived from m_{spl} for the same configuration c.

Definition 3.1 (Detectable Mutant)**.** A PL mutant $m_{spl} \in M_{spl}$ is *detectable* if there exists a configuration $c \in C_{spl}$ such that from $p_c = [\![p_{spl}, c]\!]$ and $m_c = [\![m_{spl}, c]\!]$ it follows that $T_{p_c} \neq T_{m_c}$ holds.

Hence, as a necessary criterion for a PL mutant m_{spl} to be detectable by configuration $c \in C$ it must hold that $p_c \neq m_c$ (i. e., the program part changed in the PL implementation p_{spl} to obtain PL mutant m_{spl} is syntactically contained in m_c). Furthermore, there must exist at least one test case $t = (i, o) \in T_{p_c}$ for the original program configuration p_c of configuration c not being valid for the mutated program configuration m_c (i. e., $t \notin T_{m_c}$ and, therefore, $T_{p_c} \neq T_{m_c}$). Hence, according to mutation testing of single programs [8, 38], we can directly apply the notion of RIPR criteria [8] to PL mutation testing with respect to configurations c.

- **Reachability (R):** The mutated program location is reachable by some test case $t \in T_{m_c}$.
- **Infection (I):** Reaching the mutated program location in m_c leads to an incorrect program state.
- **Propagation (P):** The incorrect program state affects the further execution of t until termination.
- **Reveal (R):** The propagated incorrect program state leads to an observable erroneous output.

Conversely, mutant $m_{spl} \in M_{spl}$ is an *equivalent PL mutant* if no product configuration exists for which the RIPR criteria hold (i. e., for all configurations $c \in C_{spl}$ either $p_c = m_c$ or $T_{p_c} = T_{m_c}$ holds). In addition, a PL mutant $m_{spl} \in M_{spl}$ is *partially equivalent* if it is detectable by at least one configuration $c \in C_{spl}$, but there also exists at least one other configuration $c' \in C_{spl}$ with $p_c = m_c$ or $T_{p_c} = T_{m_c}$. Consequently, we call mutant $m_{spl} \in M_{spl}$ *non-equivalent* if it is neither equivalent, nor partially equivalent (i. e., it is detectable on all configurations).

Example 3.2. Let us again consider the PL implementation in Fig. 2. First, assume the relational operator "`==`" in line 15 to be mutated to "`<`". This mutation affects a program part contained in all six program configurations. For instance, test

Table 1. Overview Mutation Operators Applicable for Mutating PL Mappings

	Replacement					Insertion		Other	
	Constant/Variable	Arithmetic Op.	Logic Op.	Relation Op.	In-/Decrement	Absolute Value	In-/Decrement	Delete	Negate
Condition	x	x	x	x	x	x	x		x
Assignment	x	x	x	x	x	x	x		
Presence Condition	x		x					x	x

case $t = (([0, 8, 0], 8), 1)$ for the original PL implementation detects this mutant, as the faulty output is 0 for all mutated test configurations having feature FIRST and 2 for mutated test configuration with feature COUNT or LAST. Hence, this mutant is *detectable* on all six product configurations and is therefore a non-equivalent mutant. Next, assume the expression "`i >= 0`" in line 13 to be mutated to "`i > 0`" which affects a program location being contained in only two program configurations of the PL (c_2, c_5). For instance, test case $t' = ([0, 8, 15], 0), 0)$ for the original implementation returns the faulty value -1 instead of 0 if applied to any mutated program configuration having feature LAST. However, the mutant is not detectable on program configurations without feature LAST thus constituting a *partially-equivalent mutant*. Next, mutating "`>`" in line 25 to "`>=`" affects all program configurations with feature COUNT but yields a mutant which is not detectable on any of those configurations thus constituting an *equivalent mutant*. Furthermore, the mapping mutation replacing feature variable COUNT by TRUE in line 7 results in an equivalent mutant as this presence condition is only relevant for initializing `c` in the subsequent line. Additionally, replacing FIRST and/or COUNT in line 10 by either FALSE, TRUE, another feature variable, or by negating the features results in partially-equivalent mutants. Finally, replacing "||" by "&&" in line 10 results in line 11 becoming dead code as FIRST and COUNT are conflicting features (cf. feature model in Sect. 2). □

Hence, for sample S to be able to detect a partially-equivalent mutant m_{spl}, it is insufficient that there exists a configuration $c \in S$ with $p_c \neq m_c$. In addition, we require $T_{p_c} \neq T_{m_c}$ in order for S to be able to detect m_{spl}.

However, checking the entire set C_{spl} to identify the subset $C_{m_{spl}} \subseteq C_{spl}$ of configurations detecting m_{spl} in a product-by-product manner is infeasible due to the reasons mentioned before. Instead, we propose a family-based approach for computing set $C_{m_{spl}}$ by adapting a recent technique for family-based test-suite generation for whole PL coverage [13]. In particular, Bürdek et al. [13] recently presented a family-based PL test-suite generation methodology which enables complete test-coverage of a given PL implementation without considering every possible configuration one-by-one separately. Given a test goal ℓ in p_{spl} (i. e., a program location to be covered by a test case), the procedure

$$(t, C_t) := \text{SPLTestGen}(p_{spl}, \ell, C_{spl})$$

Algorithm 1 Family-based Mutant Detection

```
Input:  Original $p_{spl}$, Mutant $m_{spl}$, Configurations $C_{spl}$
Output: Mutant-Detection Test Configurations $C_m$
 1: procedure COMPARATOR($p_{spl}$, $m_{spl}$, $input$)
 2:     $output := p_{spl}(input)$
 3:     $output' := m_{spl}(input)$
 4:     if $output \neq output'$
 5:         $\ell$ : break
 6: procedure MAIN
 7:     $C_m := \emptyset$
 8:     $C := C_{spl}$
 9:     $p :=$ Comparator($p_{spl}$, $m_{spl}$, $input$)
10:     do
11:         $(t, C_t) :=$ SPLTESTGEN($p$, $\ell$, $C$)
12:         $C := C \setminus C_t$
13:         $C_m := C_m \cup C_t$
14:     while $C_t \neq \emptyset$
```

generates a *PL test case* (t, C_t) consisting of a test case t that reaches ℓ in all configurations $C_t \subseteq C_{spl}$ (i. e., for each $c \in C_t$ it holds that $t \in T_{p_c}$). The procedure returns $C_t = \emptyset$ only if no such test case can be found on any configuration in C_{spl}. Hence, a further call

$$(t', C_t') := \text{SPLTestGen}(p_{spl}, \ell, C_{spl} \setminus C_t)$$

can be invoked to produce a further PL test case (t', C_t') for covering ℓ on the remaining set $C_{spl} \setminus C_t$. This technique is called *blocking-clause method* and is repeated until no further test case exists on the remaining set of configurations.

Example 3.3. Considering the example in Fig. 2, a test case detecting the mutation in line 13 (cf. Example 3.2) may be given as $t_{spl} = ((([0, 8, 15], 0), 0), \{c_3, c_6\})$. □

We utilize the procedure SPLTestGen in Algorithm 1 for family-based mutant detection as follows. The algorithm receives as input the original PL implementation p_{spl} and PL mutant m_{spl} and returns as output the subset $C_m \subseteq C_{spl}$ of configurations on which m_{spl} is detectable. In procedure Main, set C_m is initialized by the empty set. In addition, set C is initialized with the set of all configurations C_{spl} thus denoting the set of configurations not yet analyzed whether being able to detect m_{spl}. The procedure template Comparator is instantiated to program p by referencing to p_{spl} as well as an *input* parameter according to the types of input value of p_{spl} and m_{spl} (line 9). The call

$$(t, C_t) := \text{SPLTestGen}(p_{spl}, \ell, C_{spl} \setminus C)$$

in line 11 utilizes the PL test-case generator as describe before to search for *input* values reaching line 5 in p. The input of

```
1  int func (int x, int y, int z) {
2    int a;
3    #ifdef f
4    a = x;
5    #elif
6    a = y;
7    #endif
8    z = z - a;
9    return z;
10 }
```

Figure 4. Example for Algorithm 1 Requiring More Than One Test Case (Adapted from Bürdek et al. [13])

the resulting test case t therefore yields different outputs if applied to p_{spl} and m_{spl}, thus t is able to detect mutant m_{spl} on all configurations C_t. Set C_t is then excluded from set C (line 12) and added to set C_m (line 13) and the search is repeated until no more configurations can be found (line 14).

Example 3.4. Let the example in Fig. 2 to be p_{spl} and the mutation of line 17 to `c--` to yield a mutant m_{spl}. The first call of SPLTestGen may, for instance, return a test case $t_{spl} = (((([0, 8, 15], 0), 1), \{c_3, c_6\})$ where the expected output is 1 whereas the mutations in c_3 and c_6 would return the value -1. As a consequence, we have $C = \{c_1, c_2, c_4, c_5\}$ and $C_m = \{c_3, c_6\}$. However, the mutant cannot be detected in any configuration of C (as the respective part of the code is not part of these configurations) such that Algorithm 1 terminates with sample C_m as output. As a further example, consider the simple configurable program depicted in Fig. 4 with two configurations $c_1 = \{f\}$ and $c_2 = \emptyset$ This function either subtracts the value of variable `x` or `y` from variable `z` depending on feature f. If we assume a mutation of line 8 to `z = z + a`, Algorithm 1 may first produce a test case $t_{f1} = (((1, 0, 2), 1), \{c_1\})$ for detecting this mutant on configuration c_1 thus yielding $C = \{c_2\}$ and $C_m = \{c_1\}$. As the mutant is also detectable on configuration c_2, an additional test case, e. g., $t_{f2} = (((0, 1, 2), 1), \{c_2\})$ is generated in the next iteration thus leading to $C = \emptyset$ and $C_m = \{c_1, c_2\}$ such that the mutant is covered in all configurations by at least one test case. □

Note that we use set notation over features for better readability. In practice, an equivalent representation in terms of propositional formulae over Boolean feature variables may be used instead [11] to facilitate utilization of SAT-solvers for different analysis steps. Furthermore, note that mutation detection in general—and therefore the presented algorithm in particular—is inherently incomplete due to undecidability of program equivalence. Consequently, the algorithm is not guaranteed to terminate thus potentially returning *unknown* as a result which may be treated as a special case of equivalent mutant in practice.

3.4 Measuring Effectiveness of Product-Line Samples

Using the family-based mutation-detection technique described in the previous subsection, we are able to compute effectiveness of a sample S with respect to a given set M_{spl}:

$$effectiveness(S) = \frac{|\{m_{spl} \in M_{spl} \mid S \cap C_m \neq \emptyset|}{|M_{spl}|} \quad (1)$$

Example 3.5. Considering set M_{spl} to only contain the first mutant described in Example 3.4, we have $effectiveness(S) = 1$ for sample $S = \{c_3, c_6\}$. However, further adding to set M_{spl} the partially-equivalent mutant obtained by transforming line 30 of the example (cf. Fig. 2) results in $effectiveness(S) = 0.5$ as line 30 is not part of any configuration in S. □

4 Experimental Evaluation

The family-based mutant-detection methodology introduced in Sect. 3.3 allows for measuring the effectiveness of different kinds of PL analysis strategies in general, and sample-based PL testing in particular. To this end, we first derive a set of mutants from a PL implementation (cf. Sect. 3.2) using code-mutation operators and mapping-mutation operators . To measure effectiveness for a given sample, we investigate how many of the generated mutants are detectable by at least one test configuration in the sample. As outlined in Algorithm 1, we generate an (abstract) test suite for each mutant containing a set of test cases detecting the mutant. Those test cases are equipped with presence conditions over features denoting the set of product configurations on which the test case is able to detect the mutant. After successful termination, our algorithm guarantees that for every product configuration on which a mutant is detectable, at least one respective test case is generated. Hence, this additional information can be used for precisely measuring the mutation-detection score of a given sample.

In this section, we describe experimental evaluation results gained from applying this technique to a collection of subject systems. The goal of our evaluation is twofold: first, we show the general applicability of the novel technique and, second, it provides a fresh assessment of measuring effectiveness of τ-wise sample-based PL testing.

4.1 Research Questions

(RQ1) How does the value of τ impact effectiveness of τ-wise sample-based PL testing strategies?

(RQ2) What is the impact of partially-equivalent mutants in measuring τ-wise sample-based PL testing strategies?

4.2 Methods and Experimental Design

In order to keep the amount of experimental data graspable, we limit our considerations to $\tau \in \{1, 2, 3\}$, being the most relevant sampling criteria in practice. We divide **RQ1** into the following sub-questions.

(RQ1.1) Does increasing $\tau = 2$ to $\tau = 3$ in τ-wise sampling lead to a significant improvement of effectiveness, as compared to increasing $\tau = 1$ to $\tau = 2$?
(RQ1.2) Does decreasing $\tau = 2$ to $\tau = 1$ in τ-wise sampling lead to a significant improvement of efficiency, as compared to decreasing $\tau = 3$ to $\tau = 2$?
(RQ1.3) Does 2-wise sampling constitute the best trade-off between efficiency and effectiveness, as compared to 1-wise sampling and 3-wise sampling?

Furthermore, we divide **RQ2** into the following sub-questions.
(RQ2.1) How many partially-equivalent mutants occur on average, as compared to the total number of mutants?
(RQ2.2) How many different test configurations are capable on average to detect a partially-equivalent mutant?
(RQ2.3) To what extent does the detection of a mutant depend on selection/deselection of particular sets of features?

For both **RQ1** and **RQ2**, we apply mutation operators (cf. Sect. 3.2) to all subject systems under consideration and generate τ-wise samples for $\tau \in \{1, 2, 3\}$ for those subject systems. For each generated mutant, we apply our family-based mutant-detection technique (cf. Sect. 3.3) to compute the effectiveness of the generated samples.

Subject Systems. Our selection of subject systems consists of the following real-world PLs implemented in C.

- **VIM**: VIM (v8.1) is a new implementation of the UNIX editor VI. Many of the classes and functions in VIM are highly-configurable by a variety of different features.
- **BusyBox**: BusyBox (v1.24.2) is a reimplementation of standard Unix tools written in C for systems with limited resources. Many of the tools implemented in BusyBox consist of families of program variants, which are configurable by selection of features.

As our family-based mutation-detection technique is based on the generation of mutation-detecting test cases at unit level, we consider in our experiments a suitable selection of functions extracted from the respective subject systems. In particular, we selected only those functions depending on at least four features and having a return type other than **`void`**.

In order to systematically investigate in more detail the impact of PL characteristics on the evaluation results, we further consider **synthetically** generated PL implementations in our experiments. We therefore utilize the tool SiMPOSE[2] for reverse-engineering PL implementations from a set of program variants randomly generated from an existing non-configurable single program using mutation operators. The tool SiMPOSE takes as input N program variants and applies N-way merging to integrate all given variants into one PL implementation using superimposition of control-flow representations. The overall procedure is shown in Fig. 5 and consists of the following steps.

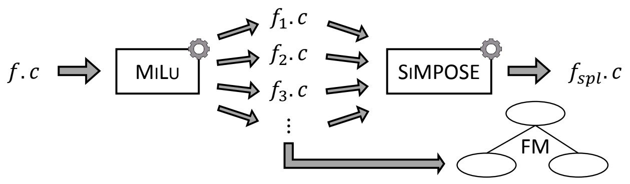


Figure 5. Synthetic SPL Generation

Table 2. Functions Extracted from the Subject Systems

Function	Subject System	Synthetic	#F	LOC
01: calculate	–	✓	5	78
02: is_method_1_triggered	BusyBox	✓	5	86
03: lcm	–	✓	5	105
04: rand_int	–	✓	5	49
05: passwd_main	BusyBox	✗	4	84
06: deluser_main	BusyBox	✗	6	476
07: minimized_get_varp	VIM	✗	7	40
08: gui_get_base_height	VIM	✗	7	35
09: gui_init_check	VIM	✗	9	107
10: insecure_flag	VIM	✗	6	45

1. We take an arbitrary existing C function $f.c$ which originally contains no variability.
2. We apply the tool MiLu to $f.c$ to generate a set of mutants $f_1.c, f_2.c, \ldots$ from $f.c$.
3. We apply SiMPOSE to integrate all generated mutants as variants into one PL implementation $f_{spl}.c$.
4. We create a feature model for the generated PL implementation where each mutated code artifact corresponds to one feature. The structure of the feature model ensures that every derivable program configuration is compilable. We further equip the feature model with randomly generated cross-tree constraints.

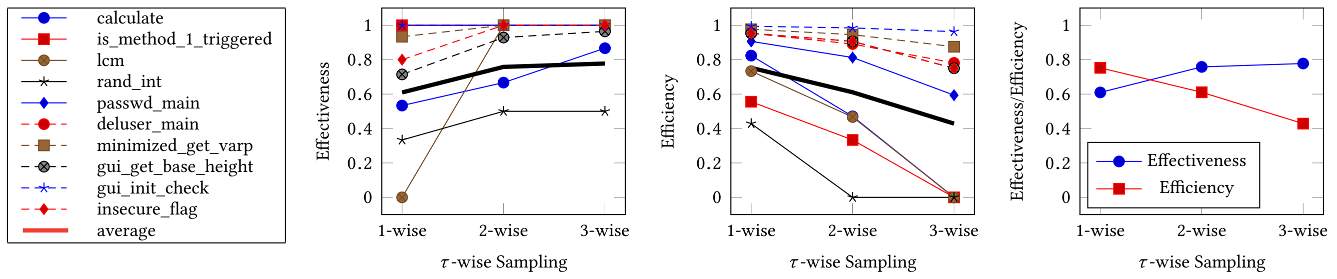
The overall number of 10 functions extracted from the subject systems for our experiments are listed in Table 2, as well as the number of features *#F* and lines of code *LOC*.

Data Collection. We collected the following data to answer our research questions. Regarding **RQ1**, we generated different τ-wise samples and measured their effectiveness and efficiency for comparison. We calculated effectiveness using Formula 1 in Sect. 3.4 and efficiency as the sample size as compared to the number of possible configurations:

$$efficiency(S) = 1 - \frac{|S|}{|C_{spl}|}$$

For analyzing the trade-off, we calculated the average values of effectiveness and efficiency. For **RQ2**, we (randomly) generated 30, 50 and 100 mutants, respectively. We chose these numbers of mutants as for many of the functions in our evaluation, no more than 100 mutants could be generated. To answer **RQ2.1**, we compare the number of partially-equivalent mutants to the total number of mutants. To answer **RQ2.2**, we compare the number of configurations being able to detect a mutant to the total number of valid configurations. To

[2]http://pi.informatik.uni-siegen.de/Projekte/variance/tosem18/



(a) Legend for Figures 6b, 6c **(b)** Effectiveness of Samples **(c)** Efficiency of Samples **(d)** Trade-off

Figure 6. Results for **RQ1.1**, **RQ1.2** and **RQ1.3**

answer **RQ2.3**, we measure the number of features which must be selected/deselected, respectively, in order to detect a particular mutant.

Tool Support. We implemented Algorithm 1 in a tool for conducting our experiments. We extended MiLu [37], an existing tool for mutant generation from C programs. We utilized CPAchecker [13] (branch *TigerIntegration*, rev. 28637), a model checker for C programs, to generate PL test cases being able to detect mutants. Specifically, we used the CPATiger algorithm, which is part of CPAchecker. Furthermore, we applied FeatureIDE [66] (v3.5.0) to generate τ-wise samples using *ICPL* [41] for which we measure effectiveness and efficiency. The required CPU time *ICPL* was negligibly short for our subject systems (see Johansen et al. [41] for more details). Finally, we utilized SiMPOSE (v1.0.0) to generate synthetic case studies. The tool implementation, the experimental results and raw data are provided on a supplementary web page[3]. As CPAchecker does not support #ifdefs, we encoded presence conditions as load-time variability [68].

Measurement Setup. Our evaluation has been executed on a Windows 10 machine, equipped with an Intel Core i7-7500 CPU with a 2.7-3.5GHz clock rate and 16GB of RAM. For each detection of a single mutant, we selected a timeout value of 10 minutes (600s). We executed our evaluation with Java 1.8.0_-161 and limited the Java heap to 6GB. For generating mutants with MiLu, we used the Linux subsystem on Windows 10.

4.3 Results

The results of our experiments addressing **RQ1** are shown in Fig. 6, while results for **RQ2** are presented in Fig. 7.
RQ1.1 (Effectiveness). As shown in Fig. 6b, effectiveness of samples generated by τ-wise sampling increases by a large margin when stepping from $\tau = 1$ to $\tau = 2$. For most generated samples, effectiveness increases by a much smaller margin when stepping from $\tau = 2$ to $\tau = 3$.
RQ1.2 (Efficiency). Figure 6c depicts efficiency measures for the different τ-wise sampling strategies. For most functions, sample efficiency decreases more from $\tau = 2$ to $\tau = 3$ than from $\tau = 1$ to $\tau = 2$.
RQ1.3 (Trade-off). Figure 6d shows a comparison of the average effectiveness and efficiency.
RQ2.1 (Partially-Equivalent Mutants). Figure 7a depicts the number of partially-equivalent mutants divided by the total number of mutants. Similar results can be observed for 30, 50 or 100 mutants.
RQ2.2 (Test Configurations per Mutant). Figure 7b shows the number of test configurations being able to detect a partially-equivalent or non-equivalent mutant compared to the number of valid product configurations. Again, similar results can be observed for 30, 50 or 100 mutants.
RQ2.3 (Feature Dependencies). As shown in Fig. 7c, the number of features to be selected/deselected for detecting a partially-equivalent mutant is between 0.78 and 3.96.

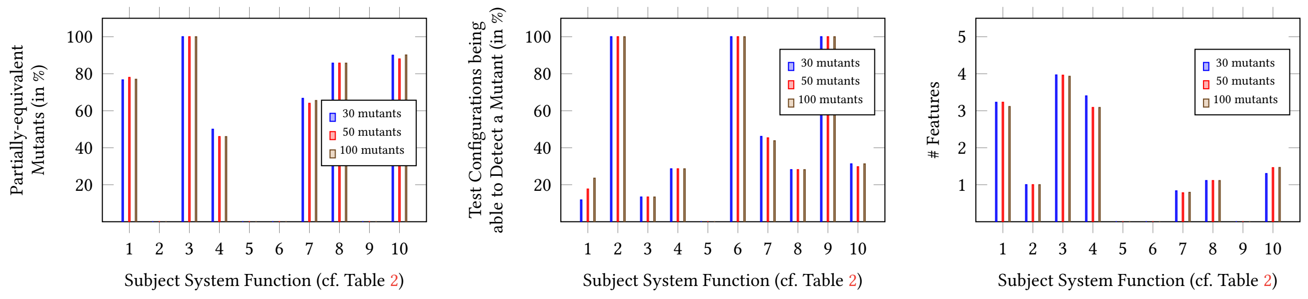
4.4 Discussion and Summary

First, we discuss the results related to **RQ1**. **Effectiveness** of most generated τ-wise samples increases when stepping $\tau = 2$ to $\tau = 3$, but by a much smaller margin than from $\tau = 1$ to $\tau = 2$. However, there are a few exceptional cases. The generation of mutants for the functions *passwd_main*, *is_method1_triggered*, *deluser_main* and *gui_init_check* only generate non-equivalent mutants, i. e., all samples are able to detect every generated mutant. As a consequence, effectiveness of all samples for these functions is constantly 1. For function *insecure_flag*, 2-wise sampling generates samples being able to detect all mutants thus also having effectiveness of 1. For mutants generated for function *calculate*, the increase of effectiveness from $\tau = 2$ to $\tau = 3$ is even higher as compared to the increase from $\tau = 1$ to $\tau = 2$.

For most functions, **efficiency** of samples decreases more from $\tau = 2$ to $\tau = 3$ than from $\tau = 1$ to $\tau = 2$. Function *rand_int* is the only exception as 2-wise samples already contains all product configuration thus having efficiency of 0. Therefore, decrease of efficiency from $\tau = 2$ to $\tau = 3$ is, on average, much higher than from $\tau = 1$ to $\tau = 2$.

Concluding, Fig. 6d shows an intersection at ~1.51 indicating that 2-wise sampling provides the best trade-off between

[3]https://www.es.tu-darmstadt.de/gpce18/



(a) Percentage of Partially-equivalent Mutants **(b)** Percentage of Test Configurations being able to Detect a Mutant **(c)** # Features Necessary to Detect a Mutant

Figure 7. Results for **RQ2.1**, **RQ2.2** and **RQ2.3**

effectiveness and efficiency of sampling for the functions considered in our evaluation.

Next, we discuss the results for **RQ2**. We were able to generate **partially-equivalent mutants** for most functions in our evaluation. Their number is between 40% to 80%. However, we identify functions for which all generated mutants are non-equivalent.

The number of **test configurations being able to detect a mutant** varies depending on the particular function. The mutants generated for the functions *passwd_main*, *is_-method1_triggered*, *deluser_main* and *gui_init_check* were all non-equivalent. Therefore, every possible product configuration is able to detect these mutants, leading to $|T| = |C_{spl}|$ for these functions.

For our last research question concerning **feature dependency**, we calculated the number of features that have to be selected or deselected in every test case for detecting a given mutant. We observed the number of features to be between 0.78 and 3.96 for partially-equivalent mutants.

4.5 Threats to Validity

A threat to **internal validity** might arise from the applied mutation operators. However, by randomly selecting mutants in our evaluation, we hope to avoid possibly favorable operators. In addition, we are currently limited to programs implemented in C. Nevertheless, our approach should, in principle, be applicable to any programming language based on an imperative core language and one may expect similar results. A threat to **external validity** may be the lack of comparison to other approaches rather than our own baseline. However, this is due to the novelty of our approach. To the best of our knowledge, no similar technique has been described so far for measuring effectiveness of sample-based PL testing. Furthermore, the selection of subject systems (including synthetic functions and the random generation of constraints for these synthetic functions) may constitute a threat to validity. While mutants of VIM behave quite similar to mutants generated from synthetic PLs in terms of effectiveness, mutants generated for BusyBox are often non-equivalent. This leads to different results concerning **RQ1.1**, depending on the subject systems, due to the different number of features required for detecting a mutant. For other research questions, the different subject systems lead to similar results. Although the number of systems is small and the results slightly differ, these PLs have, in our opinion, reasonable size and complexity. Additionally, the real-world examples (e. g., VIM) constitute a reliable reference.

5 Related Work

We discuss related work on (1) how effectiveness of samples can be measured, and (2) how mutation-based testing is applied to PLs so far. We further refer to dedicated surveys on PL testing [18, 25, 27, 46, 56] and sample-based PL testing [3, 47, 67] for a broader overview.

Sample-based PL testing is typically evaluated w. r. t. three main criteria, i. e., sampling efficiency, testing efficiency, and effectiveness [67]. While sampling efficiency indicates the effort to compute a sample, testing efficiency refers to the effort for testing such a sample.

Effectiveness is mainly assessed by counting the number of covered, potential (e. g., pairwise) feature interactions. Even though numerous algorithms guarantee the coverage of feature interactions up-to a certain degree τ [5, 16, 31, 33, 40, 41, 49, 55, 59], it is still possible to assess how many interactions of a degree higher than τ are covered. While this effectiveness metric can be easily computed given the feature model, it is a rather abstract metric and does not incorporate any properties of the PL implementation. In contrast, Devroey et al. [22, 23] evaluated the effectiveness in terms of feature and behavioral model coverage.

Tartler et al. [65] assess the effectiveness by incorporating the PL implementation. They focus on PLs implemented with a preprocessor and measure the coverage of `#ifdef` blocks, i. e., if it is visible to the compiler in some configuration and, thus, can potentially be executed. While this metric includes the feature-to-code mapping, it is still oblivious to interactions caused by several #ifdef blocks and to the source code aside of #ifdef statements.

Recently, researchers started to evaluate samples by means of real faults [1, 9, 30, 32, 52, 62–64]. Sánchez et al. [62, 63] introduced the Drupal case study for measuring the effectiveness of samples. Fischer et al. evaluated not only real faults of the Drupal case study, but also covered feature interactions [30]. Similarly, Tartler et al. evaluated real faults together with the coverage of #ifdef blocks [64]. Medeiros et al. [52] assessed the effectiveness based on configuration-related faults in open-source C systems, e. g., BusyBox and VIM. Halin et al. [32] investigated exhaustive testing of the industrial-size system JHipster and found a set of faults used afterwards for the effectiveness evaluation. Real faults are a good source for evaluation, but their number is typically too low and they are only known for some PLs.

Besides those three strategies for measuring effectiveness, it is also possible to apply mutation-based testing. While mutations can be applied to the feature model, the mapping from features to source code, and to the source code itself [4], we start with work on feature-model mutations.

Feature-model mutation is realized either by mutating the feature diagram [9, 26, 28, 29, 50, 61] or by mutating a propositional formula [34, 35]. Resulting mutants have been used for evaluation of sampling effectiveness [26, 29, 34, 35, 50, 61], and for sampling computation [9, 29, 34, 50, 61]. Lackner and Schmidt [45] use feature-model mutants to assess the effectiveness of model-based PL testing techniques. Al-Hajjaji et al. [4] mutate Kconfig variability models, where feature-definition and -dependency faults are simulated. In this paper, we abstract from feature-model mutation, but our framework facilitates its application by embedding all feature-model constraints as compound propositional formula [11] into the implementation such that we may utilize our mapping-mutation operators also for feature-model mutation.

For feature-mapping mutation, Al-Hajjaji et al. [4] also came up with a distinct set of operators defined for #ifdef directives, e. g., #ifdef is changed to #ifndef. Lackner and Schmidt [45] defined operators for mutating feature mappings in annotative UML state machines. Although differently defined, the result of the mapping mutations are similar to mutations in our framework, e. g., variable code is added to configurations it was not intended for. To the best of our knowledge, there is no work that applies feature-mapping mutations for the assessment of the effectiveness of samples.

Source-code mutation is the standard application of single-software mutation testing. We refer to Jia and Harman [38, 39] and Offutt [54] for an overview. Papadakis et al. [57] and Bures and Ahmed [14] also use mutation testing to assess the effectiveness of combinatorial interaction testing, where interactions of program input parameters are sampled as unit test cases. However, the authors focus on single-software testing and mutate the input model capturing all input parameters, their potential values, and contraints between them. Al-Hajjaji et al. [4] proposed the application of existing source-code mutation operators restricted to variable code blocks, i. e., #ifdef blocks. Our framework uses existing operators implemented in MiLu [37], but we do not restrict their application to solely #ifdef blocks. Lackner and Schmidt [45] mutate variable UML state machines, e. g., by removing/adding transitions. Devroey et al. [20] proposed model-based SPL mutation testing and discussed potential applications and challenges. In contrast, our framework is solely applicable to configurable systems implemented in C.

Furthermore, SPL mutation analysis can be improved by applying (1) mutant sampling [6, 24], (2) similar and equivalent mutant detection [15, 61], and (3) variability analysis [6] to reduce the number of mutants to be analyzed. In this paper, we omit the potential improvements of mutant reduction as well as early detection of equivalent or similar mutants and leave the extensions open for future work.

6 Conclusion and Future Work

In this paper, we presented a framework for measuring effectiveness of PL analysis strategies based on the principles of mutation testing. In particular, we focused our considerations on measuring effectiveness of sample-based PL testing strategies in terms of the mutation-detection score achieved by the set of test configurations contained in generated samples. We further developed a tool implementation for family-based mutant detection that allows for a fully-automated effectiveness-evaluation of sampling algorithms for PLs implemented in C. The experimental results gained from applying our tool to a collection of realistic subject systems confirms the well-known assumption that pairwise sampling constitutes the best efficiency/effectiveness trade-off for sample-based PL testing.

In future experiments, we plan to also measure effectiveness of product prioritization techniques [7, 10, 21, 63] and to measure effectiveness of alternative PL analysis strategies beyond sample-based testing (e. g., family-based PL model-checking [66]). In addition, we plan to utilize our approach to not only measure the effectiveness of existing samples, but also to pro-actively generate samples achieving a predefined degree of effectiveness on a given set of PL mutants. Furthermore, we plan to consider larger functions and whole c-programs as subject systems.

Acknowledgments

This work was funded by the Hessian LOEWE initiative within the Software-Factory 4.0 project as well as by the German Research Foundation (DFG) in the Priority Programme SPP 1593: Design For Future – Managed Software Evolution (LO 2198/2-1, SCHA 1635/10-1), by CAPES/PROCAD (175956), by CAPES/PGCI (117875), by CNPq (307190/2015-3), and by FAPEAL PPGs (14/2016 60030 000435/2017).

References

[1] Iago Abal, Jean Melo, Stefan Stănciulescu, Claus Brabrand, Márcio Ribeiro, and Andrzej Wąsowski. 2018. Variability Bugs in Highly Configurable Systems: A Qualitative Analysis. *ACM Trans. Softw. Eng. Methodol.* 26, 3, Article 10 (Jan. 2018), 34 pages. https://doi.org/10.1145/3149119

[2] Hiralal Agrawal, Richard A. DeMillo, Bob Hathaway, William Hsu, Wynne Hsu, E. W. Krauser, R. J. Martin, Aditya P. Mathur, and Eugene Spafford. 1989. Design of mutant operators for the C programming language – Technical Report. (1989).

[3] B. S. Ahmed, K. Z. Zamli, W. Afzal, and M. Bures. 2017. Constrained Interaction Testing: A Systematic Literature Study. *IEEE Access* 5 (2017), 25706–25730. https://doi.org/10.1109/ACCESS.2017.2771562

[4] Mustafa Al-Hajjaji, Fabian Benduhn, Thomas Thüm, Thomas Leich, and Gunter Saake. 2016. Mutation Operators for Preprocessor-Based Variability. In *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS '16)*. ACM, New York, NY, USA, 81–88. https://doi.org/10.1145/2866614.2866626

[5] Mustafa Al-Hajjaji, Sebastian Krieter, Thomas Thüm, Malte Lochau, and Gunter Saake. 2016. IncLing: Efficient Product-line Testing Using Incremental Pairwise Sampling. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2016)*. ACM, New York, NY, USA, 144–155. https://doi.org/10.1145/2993236.2993253

[6] M. Al-Hajjaji, J. Krüger, F. Benduhn, T. Leich, and G. Saake. 2017. Efficient Mutation Testing in Configurable Systems. In *2017 IEEE/ACM 2nd International Workshop on Variability and Complexity in Software Design (VACE)*. 2–8. https://doi.org/10.1109/VACE.2017.3

[7] Mustafa Al-Hajjaji, Thomas Thüm, Malte Lochau, Jens Meinicke, and Gunter Saake. 2016. Effective Product-Line Testing using Similarity-Based Product Prioritization. *Software & Systems Modeling* (17 Dec 2016). https://doi.org/10.1007/s10270-016-0569-2

[8] Paul Ammann and Jeff Offutt. 2016. *Introduction to Software Testing*. Cambridge University Press.

[9] P. Arcaini, A. Gargantini, and P. Vavassori. 2015. Generating Tests for Detecting Faults in Feature Models. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. 1–10. https://doi.org/10.1109/ICST.2015.7102591

[10] H. Baller, S. Lity, M. Lochau, and I. Schaefer. 2014. Multi-objective Test Suite Optimization for Incremental Product Family Testing. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. 303–312. https://doi.org/10.1109/ICST.2014.43

[11] Don Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *Software Product Lines*, Henk Obbink and Klaus Pohl (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 7–20.

[12] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems* 35, 6 (2010), 615–636.

[13] Johannes Bürdek, Malte Lochau, Stefan Bauregger, Andreas Holzer, Alexander von Rhein, Sven Apel, and Dirk Beyer. 2015. Facilitating Reuse in Multi-goal Test-Suite Generation for Software Product Lines. In *Fundamental Approaches to Software Engineering*, Alexander Egyed and Ina Schaefer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 84–99.

[14] M. Bures and B. S. Ahmed. 2017. On the Effectiveness of Combinatorial Interaction Testing: A Case Study. In *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. 69–76. https://doi.org/10.1109/QRS-C.2017.20

[15] Luiz Carvalho, Marcio Augusto Guimarães, Márcio Ribeiro, Leonardo Fernandes, Mustafa Al-Hajjaji, Rohit Gheyi, and Thomas Thüm. 2018. Equivalent Mutants in Configurable Systems: An Empirical Study. In *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS 2018)*. ACM, New York, NY, USA, 11–18. https://doi.org/10.1145/3168365.3168379

[16] Anastasia Cmyrev and Ralf Reissing. 2014. Efficient and Effective Testing of Automotive Software Product Lines. *Int'l J. Applied Science and Technology (IJAST)* 7, 2 (2014). https://doi.org/10.14416/j.ijast.2014.05.001

[17] Krzysztof Czarnecki and Ulrich W Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications*. Vol. 16. Addison Wesley Reading.

[18] 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. *Inf. Softw. Technol.* 53 (2011), 407–423. Issue 5.

[19] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* 11, 4 (April 1978), 34–41. https://doi.org/10.1109/C-M.1978.218136

[20] Xavier Devroey, Gilles Perrouin, Maxime Cordy, Mike Papadakis, Axel Legay, and Pierre-Yves Schobbens. 2014. A Variability Perspective of Mutation Analysis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 841–844. https://doi.org/10.1145/2635868.2666610

[21] Xavier Devroey, Gilles Perrouin, Maxime Cordy, Pierre-Yves Schobbens, Axel Legay, and Patrick Heymans. 2013. Towards Statistical Prioritization for Software Product Lines Testing. In *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS '14)*. ACM, New York, NY, USA, Article 10, 7 pages. https://doi.org/10.1145/2556624.2556635

[22] Xavier Devroey, Gilles Perrouin, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. 2015. Covering SPL Behaviour with Sampled Configurations: An Initial Assessment. In *Proceedings of the Ninth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS '15)*. ACM, New York, NY, USA, Article 59, 8 pages. https://doi.org/10.1145/2701319.2701325

[23] Xavier Devroey, Gilles Perrouin, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. 2016. Search-Based Similarity-Driven Behavioural SPL Testing. In *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS '16)*. ACM, New York, NY, USA, 89–96. https://doi.org/10.1145/2866614.2866627

[24] Xavier Devroey, Gilles Perrouin, Mike Papadakis, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. 2016. Featured Model-based Mutation Analysis. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 655–666. https://doi.org/10.1145/2884781.2884821

[25] Ivan do Carmo Machado, John D. McGregor, Yguarata Cerqueira Cavalcanti, and Eduardo Santana de Almeida. 2014. On Strategies for Testing Software Product Lines: A Systematic Literature Review. *Information and Software Technology* 56, 10 (2014), 1183 – 1199. https://doi.org/10.1016/j.infsof.2014.04.002

[26] T. do Nascimento Ferreira, J. N. Kuk, A. Pozo, and S. R. Vergilio. 2016. Product selection based on upper confidence bound MOEA/D-DRA for testing software product lines. In *2016 IEEE Congress on Evolutionary Computation (CEC)*. 4135–4142. https://doi.org/10.1109/CEC.2016.7744315

[27] Emelie Engström and Per Runeson. 2011. Software Product Line Testing - A Systematic Mapping Study. *Information and Software Technology* 53, 1 (2011), 2 – 13. https://doi.org/10.1016/j.infsof.2010.05.011

[28] T. N. Ferreira, J. A. P. Lima, A. Strickler, J. N. Kuk, S. R. Vergilio, and A. Pozo. 2017. Hyper-Heuristic Based Product Selection for Software Product Line Testing. *IEEE Computational Intelligence Magazine* 12, 2 (May 2017), 34–45. https://doi.org/10.1109/MCI.2017.2670461

[29] Helson L. Jakubovski Filho, Jackson A. Prado Lima, and Silvia R. Vergilio. 2017. Automatic Generation of Search-Based Algorithms Applied to the Feature Testing of Software Product Lines. In *Proceedings of*

the 31st Brazilian Symposium on Software Engineering (SBES'17). ACM, New York, NY, USA, 114–123. https://doi.org/10.1145/3131151.3131152

[30] S. Fischer, R. E. Lopez-Herrejon, R. Ramler, and A. Egyed. 2016. A Preliminary Empirical Assessment of Similarity for Combinatorial Iteraction Testing of Software Product Lines. In *2016 IEEE/ACM 9th International Workshop on Search-Based Software Testing (SBST)*. 15–18. https://doi.org/10.1109/SBST.2016.011

[31] Brady J. Garvin, Myra B. Cohen, and Matthew B. Dwyer. 2011. Evaluating Improvements to a Meta-Heuristic Search for Constrained Interaction Testing. *Empirical Software Engineering* 16, 1 (01 Feb 2011), 61–102. https://doi.org/10.1007/s10664-010-9135-7

[32] Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, and Benoit Baudry. 2017. Test Them All, Is It Worth It? Assessing Configuration Sampling on the JHipster Web Development Stack. (2017). https://doi.org/10.1007/s10664-018-9635-4 arXiv:arXiv:1710.07980 (submitted to Empirical Software Engineering).

[33] Evelyn Nicole Haslinger, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2013. Using Feature Model Knowledge to Speed Up the Generation of Covering Arrays. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems (VaMoS '13)*. ACM, New York, NY, USA, Article 16, 6 pages. https://doi.org/10.1145/2430502.2430524

[34] Christopher Henard, Mike Papadakis, and Yves Le Traon. 2014. Mutation-Based Generation of Software Product Line Test Configurations. In *Search-Based Software Engineering*, Claire Le Goues and Shin Yoo (Eds.). Springer International Publishing, 92–106. https://doi.org/10.1007/978-3-319-09940-8_7

[35] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. L. Traon. 2013. Assessing Software Product Line Testing Via Model-Based Mutation: An Application to Similarity Testing. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*. 188–197. https://doi.org/10.1109/ICSTW.2013.30

[36] Laura Inozemtseva and Reid Holmes. 2014. Coverage is Not Strongly Correlated with Test Suite Effectiveness. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 435–445. https://doi.org/10.1145/2568225.2568271

[37] Y. Jia and M. Harman. 2008. MILU: A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language. In *Testing: Academic Industrial Conference - Practice and Research Techniques (TAIC PART 2008)*. 94–98. https://doi.org/10.1109/TAIC-PART.2008.18

[38] Y. Jia and M. Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (2011), 649–678. https://doi.org/10.1109/TSE.2010.62

[39] Yue Jia and Mark Harman. 2014 (accessed June 6, 2018). *Mutation Testing Repository*. http://crestweb.cs.ucl.ac.uk/resources/mutation_testing_repository/

[40] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. 2011. Properties of Realistic Feature Models Make Combinatorial Testing of Product Lines Feasible. In *Model Driven Engineering Languages and Systems*, Jon Whittle, Tony Clark, and Thomas Kühne (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 638–652.

[41] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. 2012. An Algorithm for Generating T-wise Covering Arrays from Large Feature Models. In *Proceedings of the 16th International Software Product Line Conference - Volume 1 (SPLC '12)*. ACM, 46–55. https://doi.org/10.1145/2362536.2362547

[42] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are Mutants a Valid Substitute for Real Faults in Software Testing?. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 654–665. https://doi.org/10.1145/2635868.2635929

[43] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report. Carnegie-Mellon Univ., Software Engineering Inst.

[44] D. R. Kuhn, D. R. Wallace, and A. M. Gallo. 2004. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering* 30, 6 (2004), 418–421. https://doi.org/10.1109/TSE.2004.24

[45] Hartmut Lackner and Martin Schmidt. 2014. Towards the Assessment of Software Product Line Tests: A Mutation System for Variable Systems. In *Proceedings of the 18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools - Volume 2 (SPLC '14)*. ACM, 62–69. https://doi.org/10.1145/2647908.2655968

[46] Jihyun Lee, Sungwon Kang, and Danhyung Lee. 2012. A Survey on Software Product Line Testing. In *Proceedings of the 16th International Software Product Line Conference - Volume 1 (SPLC '12)*. ACM, New York, NY, USA, 31–40. https://doi.org/10.1145/2362536.2362545

[47] R. E. Lopez-Herrejon, S. Fischer, R. Ramler, and A. Egyed. 2015. A First Systematic Mapping Study on Combinatorial Interaction Testing for Software Product Lines. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 1–10. https://doi.org/10.1109/ICSTW.2015.7107435

[48] L. Madeyski, W. Orzeszyna, R. Torkar, and M. Jozala. 2014. Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation. *IEEE Transactions on Software Engineering* 40, 1 (Jan 2014), 23–42. https://doi.org/10.1109/TSE.2013.44

[49] Dusica Marijan, Arnaud Gotlieb, Sagar Sen, and Aymeric Hervieu. 2013. Practical Pairwise Testing for Software Product Lines. In *Proceedings of the 17th International Software Product Line Conference (SPLC '13)*. ACM, New York, NY, USA, 227–235. https://doi.org/10.1145/2491627.2491646

[50] Rui A. Matnei Filho and Silvia R. Vergilio. 2016. A multi-objective test data generation approach for mutation testing of feature models. *Journal of Software Engineering Research and Development* 4, 1 (26 Jul 2016), 4. https://doi.org/10.1186/s40411-016-0030-9

[51] John D. McGregor. 2010. *Testing a Software Product Line*. Springer Berlin Heidelberg, Berlin, Heidelberg, 104–140. https://doi.org/10.1007/978-3-642-14335-9_4

[52] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 643–654. https://doi.org/10.1145/2884781.2884793

[53] A. Jefferson Offutt and Jie Pan. 1998. Automatically Detecting Equivalent Mutants and Infeasible Paths. *Software Testing, Verification and Reliability* 7, 3 (1998), 165–192.

[54] Jeff Offutt. 2011. A Mutation Carol: Past, Present and Future. *Information and Software Technology* 53, 10 (2011), 1098 – 1107. https://doi.org/10.1016/j.infsof.2011.03.007 Special Section on Mutation Testing.

[55] Sebastian Oster, Florian Markert, and Philipp Ritter. 2010. Automated Incremental Pairwise Testing of Software Product Lines. In *Software Product Lines: Going Beyond*, Jan Bosch and Jaejoon Lee (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 196–210.

[56] S. Oster, A. Wübbeke, G. Engels, and A. Schürr. 2011. A Survey of Model-Based Software Product Lines Testing. In *Model-based Testing for Embedded Systems*. CRC Press, 338 – 381.

[57] M. Papadakis, C. Henard, and Y. L. Traon. 2014. Sampling Program Inputs with Mutation Analysis: Going Beyond Combinatorial Interaction Testing. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. 1–10. https://doi.org/10.1109/ICST.2014.11

[58] M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon. 2015. Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and

Effective Equivalent Mutant Detection Technique. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 936–946. https://doi.org/10.1109/ICSE.2015.103

[59] G. Perrouin, S. Sen, J. Klein, B. Baudry, and Y. l. Traon. 2010. Automated and Scalable T-wise Test Case Generation Strategies for Software Product Lines. In *2010 Third International Conference on Software Testing, Verification and Validation*. 459–468. https://doi.org/10.1109/ICST.2010.43

[60] Klaus Pohl, Günter Böckle, and Frank J van Der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer Science & Business Media.

[61] Dennis Reuling, Johannes Bürdek, Serge Rotärmel, Malte Lochau, and Udo Kelter. 2015. Fault-based Product-line Testing: Effective Sample Generation Based on Feature-diagram Mutation. In *Proceedings of the 19th International Conference on Software Product Line (SPLC '15)*. ACM, New York, NY, USA, 131–140. https://doi.org/10.1145/2791060.2791074

[62] Ana B. Sánchez, Sergio Segura, José A. Parejo, and Antonio Ruiz-Cortés. 2017. Variability Testing in the Wild: The Drupal Case Study. *Software & Systems Modeling* 16, 1 (2017), 173–194.

[63] A. B. Sánchez, S. Segura, and A. Ruiz-Cortés. 2014. A Comparison of Test Case Prioritization Criteria for Software Product Lines. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. 41–50. https://doi.org/10.1109/ICST.2014.15

[64] Reinhard Tartler, Christian Dietrich, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2014. Static Analysis of Variability in System Software: The 90,000 #Ifdefs Issue. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)*. USENIX Association, Berkeley, CA, USA, 421–432. http://dl.acm.org/citation.cfm?id=2643634.2643677

[65] Reinhard Tartler, Daniel Lohmann, Christian Dietrich, Christoph Egger, and Julio Sincero. 2012. Configuration Coverage in the Analysis of Large-scale System Software. *SIGOPS Oper. Syst. Rev.* 45, 3 (Jan. 2012), 10–14. https://doi.org/10.1145/2094091.2094095

[66] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.* 47, 1, Article 6 (June 2014), 45 pages. https://doi.org/10.1145/2580950

[67] Mahsa Varshosaz, Mustafa Al-Hajjaji, Thomas Thüm, Tobias Runge, Mohammadreza Mousavi, and Ina Schaefer. 2018. A Classification of Product Sampling for Software Product Lines. To appear.

[68] Alexander von Rhein, Thomas Thüm, Ina Schaefer, Jörg Liebig, and Sven Apel. 2016. Variability Encoding: From Compile-Time to Load-Time Variability. *Journal of Logical and Algebraic Methods in Programming* 85, 1, Part 2 (2016), 125 – 145. https://doi.org/10.1016/j.jlamp.2015.06.007 Formal Methods for Software Product Line Engineering.