Mutation Operators for Feature-Oriented Software Product Lines

Jacob Krüger[1,2*], Mustafa Al-Hajjaji[1*], Thomas Leich[2,3], and Gunter Saake[1]

[1] *Otto-von-Guericke University, Magdeburg, Germany*
[2] *Harz University of Applied Sciences, Wernigerode, Germany*
[3] *METOP GmbH, Magdeburg, Germany*

SUMMARY

Mutation testing is an approach to assess the quality of test cases. Mutants are modified versions of a system that ideally comprise faulty behavior. Test cases for a system are effective if they kill these mutants. For software product lines, several works have addressed mutation testing to inject variability faults, which may only exist in some variants. These works focus on variability models or specific implementation techniques. In contrast, feature-oriented programming has been rarely investigated, wherefore we *i)* derive corresponding mutation operators, *ii)* investigate the feasibility of our proposed and conventional operators on four software product lines, and *iii)* discuss open challenges in mutation testing of software product lines. The results show that our proposed operators are suitable to cause variability faults and extend the capabilities of conventional operators. Nonetheless, mutation testing of software product lines is comparably expensive, due to a high number of variants and mutants—resulting in equivalence and redundancy. Copyright © 2018 John Wiley & Sons, Ltd.

Received ...

KEY WORDS: Mutation testing; mutation operator; software product line; composition; unit test; feature-oriented programming

1. INTRODUCTION

Mutation testing is an approach to assess the quality of a test suite [1, 2, 3]. A system is mutated with mutation operators that potentially mimic developers' mistakes. For example, operators replace logical operations, remove a statement, or change increments [3, 4, 5, 6]. The generated mutants are then executed on the system's test suite, which is effective if it kills these mutants. However, the effectiveness of mutation testing depends heavily on the operators' quality.

Mutation testing has shown its value for assessing systems and their test suites in several domains [3]. As a result, this approach has also been adopted in the context of software-product-line engineering [7, 8, 9], to which we refer to as *variability-aware mutation testing* in this article. Software product lines enable developers to systematically reuse *features*, which represent a user-visible functionality, and customize similar variants of a system [10, 11, 12]. Due to the changed process of deriving a variant – namely modeling, configuring, and instantiating a variant based on the software product line's features – as well as the differing programming paradigm, adapted mutation operators are necessary. In particular, these operators are required to cause variability faults. Consequently, recent works have proposed new operators to introduce faults in variability-models, variability-mapping, and domain-artifacts (i.e., the actual source code) [8].

*Mustafa Al-Hajjaji works now for pure-systems GmbH, Magdeburg, Germany.
*Correspondence to: Otto-von-Guericke University, Universitätsplatz 2, 39106 Magdeburg, Germany. E-mail: jkrueger@ovgu.de

For this reason, we distinguish two types of mutation operators in this article: *Conventional* and *variability-aware* operators. Conventional operators are not specifically adapted to resemble faults that may occur due to variability. In contrast, variability-aware operators are designed for the purpose of mutating software product lines. Thus, they aim to resemble faults that are related to variability. A main challenge in variability-aware mutation testing is the large number of variants that can be instantiated from a software product line [9]. Each variant includes a different set of features, which is why variants have to be tested separately. As the same feature can be reused in different variants and may not be affected by variability, several *equivalent* and *redundant* mutants [13, 14, 15, 16] may be created that are either identical to the original variants or to each other, respectively. This increases the costs and execution time significantly, hampering the practical applicability of mutation testing for software product lines. To overcome this challenge, adapted cost reduction techniques and variability-aware mutation operators are necessary [9, 17, 18, 19].

In this article, we investigate mutation testing of composition-based software product lines – implemented with feature-oriented programming [20] – and corresponding challenges, building on our previous works on mutation testing of preprocessor-based systems [8, 9]. We first propose a set of variability-aware mutation operators that potentially introduce variability faults for feature-oriented programming. Furthermore, we provide a study on conventional and variability-aware mutation operators to investigate the usability of our proposed operators. We discuss open challenges in variability-aware mutation testing and composition in particular. More precisely, we contribute the following:

- We propose a set of mutation operators for feature-oriented programming and exemplify them. Thus, we enable variability-aware mutation testing of such software product lines.
- We report a study on conventional and our proposed mutation operators. This way, we aim to illustrate the necessity for variability-aware mutation operators. We also consider equivalent and redundant (i.e., duplicated) mutants, which are additionally challenging and costly in mutation testing of software product lines.
- We discuss open challenges in mutation testing of software product lines. These challenges are based on our study and aim to open further research directions.

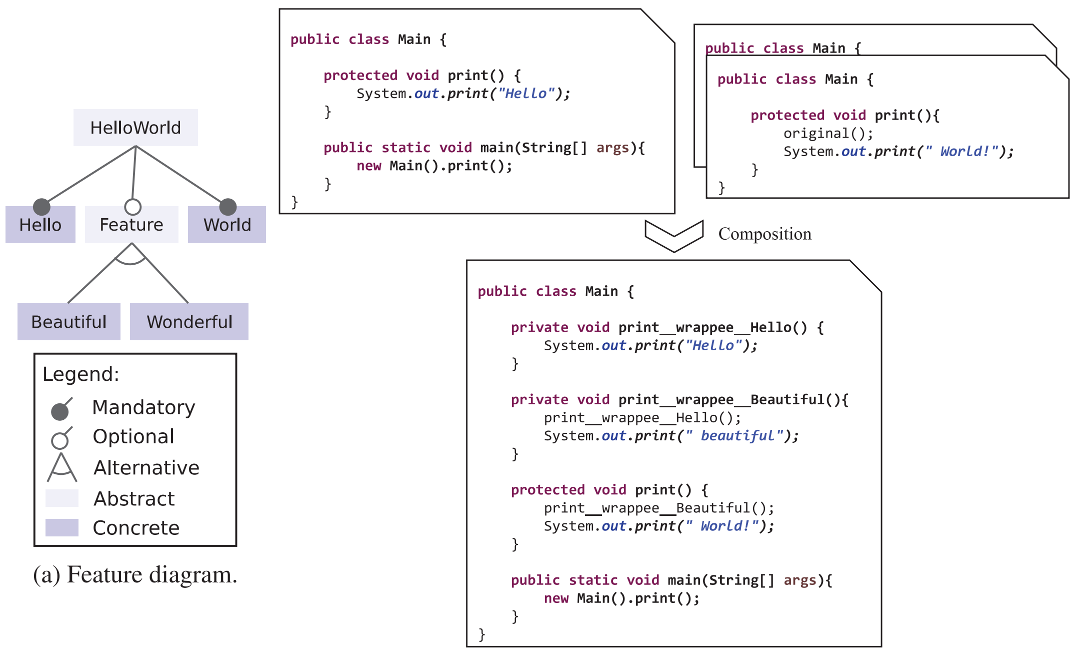
The remaining article is structured as follows: In Section 2, we provide background on feature-oriented programming, mutation testing, and variability faults. We describe and exemplify our proposed mutation operators in Section 3. Then, we describe the design of our case study and evaluation in Section 4. In Section 5, we report and discuss the results of the study and follow up with a discussion of the identified challenges in Section 6. We discuss potential threats to validity in Section 7. Finally, we describe related work in Section 8 and conclude in Section 9.

2. BACKGROUND

There exist several approaches to implement software product lines [12, 21]. In the context of this article, we focus on composition-based approaches and feature-oriented programming [20] in particular. Nonetheless, the proposed mutation operators should partly be applicable to other composition-based approaches, such as aspect-oriented programming [22]. In this section, we introduce *feature-oriented programming* and *mutation testing*. We then describe the relation between *variability faults* that can be introduced by mutation operators and feature-oriented programming.

2.1. Feature-Oriented Programming

A software product line is defined by its features that describe common and variable functionalities of a system [10, 11, 12]. In feature-oriented programming, features are implemented (as so called *domain artifacts*) in physically separated modules. Each feature can refine classes and methods of another feature. To derive a variant, the features' code is merged based on their structure, the composition mechanism [12, 23]. In feature-oriented programming, the `original` method is used as *refinement call* to define if and at which point feature refinements are composed. If this method is



```
public class Main {

    protected void print() {
        System.out.print("Hello");
    }

    public static void main(String[] args){
        new Main().print();
    }
}
```

```
public class Main {
```

```
public class Main {

    protected void print(){
        original();
        System.out.print(" World!");
    }
}
```

```
public class Main {

    private void print__wrappee__Hello() {
        System.out.print("Hello");
    }

    private void print__wrappee__Beautiful(){
        print__wrappee__Hello();
        System.out.print(" beautiful");
    }

    protected void print() {
        print__wrappee__Beautiful();
        System.out.print(" World!");
    }

    public static void main(String[] args){
        new Main().print();
    }
}
```

(a) Feature diagram.

(b) Implemented feature modules.

Figure 1. Feature-oriented programming using FeatureHouse.

missing, the refining code (e.g., a method) completely replaces the existing one. The features and their dependencies are modeled with *variability models* [24, 25, 26]. To display the variability of a software product line, we use feature diagrams [12, 26, 27], which are graphical representations of the commonly used *feature models* [25, 28]. Based on the defined dependencies, developers can derive a valid configuration to scope and compose a variant.

One of the main advantages of composition-based approaches is the physical separation of features. This facilitates traceability and maintenance [12, 29, 30], but also poses new challenges. For example, analyzing and testing become more challenging, due to the complexity and separation of code [12, 31]. Existing composition-based approaches rely on the same concepts, but differ in implementation details compared to feature-oriented programming.

In Figure 1, we show an example for feature-oriented programming using FeatureHouse [23, 32], a tool to compose the corresponding source code. In Figure 1a, we illustrate the feature diagram for a Hello-World software product line. We see that there are 6 different features, namely, `HelloWorld`, `Hello`, `Feature`, `Beautiful`, `Wonderful`, and `World`. Of these, two (i.e., `HelloWorld` and `Feature`) are abstract, meaning that they do not have an implementation and are only used to structure the software product line. `Hello` and `World` are mandatory features and, thus, are required for every variant. In contrast, `Beautiful` and `Wonderful` are alternatives, wherefore only one of them can be selected at the same time.

We exemplify the implementation and composition in Figure 1b. Here, the base code is part of the `Hello` feature (top left) and is refined with two other features: `Beautiful` and `World` (top right). We remark that the base does not need to implement all core functionality, but can also be extended by mandatory features (i.e., `Hello` on its own is not a valid configuration according to the feature model). Overall, in our example the selected configuration comprises `Hello`, `Beautiful`, and `World`. As this is a valid configuration according to the feature model, the variant is composed (bottom) and can be used afterwards. In Figure 1b, we also show a characteristic of most composition-based approaches: They require a defined order in which features are composed [12, 20, 33, 34]. Consequently, the outcome depends on whether we first include `Beautiful` and then `World`, or the other way around – which would exchange the order of the console output.

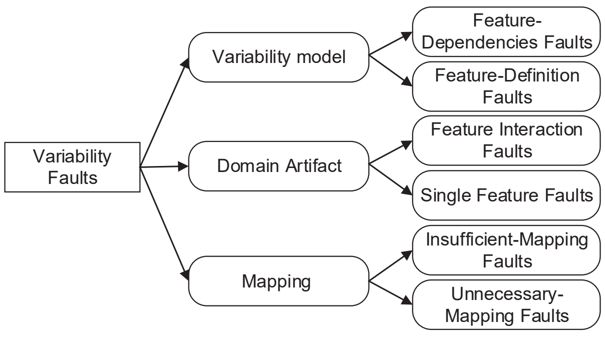

Variability Faults
Variability model
Domain Artifact
Mapping
Feature-Dependencies Faults
Feature-Definition Faults
Feature Interaction Faults
Single Feature Faults
Insufficient-Mapping Faults
Unnecessary-Mapping Faults

Figure 2. Variability-based faults [8].

Besides composition-based approaches, such as feature-oriented programming, several other implementation techniques for software product lines exist [12, 21]. In practice, annotation-based approaches, mainly the C preprocessor (CPP) [35], are widely used [12, 36, 37]. Here, features are implemented in a single code base and marked with annotations, for instance, `#ifdef` in the CPP. Based on a valid configuration, the variable code is then removed instead of composed.

2.2. *Mutation Testing*

Mutation testing aims to evaluate the effectiveness of test cases for a software system [3]. To achieve this, mutation operators automatically inject changes into the system, potentially simulating developers' mistakes. The resulting mutants are then tested against the test cases. These tests either fail in their execution (because an error occurs) or run successfully (because they do not run into an error). In the first scenario, the mutant is *killed*, which means that the test cases detect the injected mutation. In the second scenario, the test cases do not reveal the injected mutation, wherefore the mutant is *alive*. The effectiveness of test cases, and the test suite, refers to their ability to kill mutants.

Mutation testing is used in several domains and has shown its value to evaluate test cases [3]. It has been used to test different aspects of a system. For instance, several operators aim to mutate the source code of different programming languages, such as, C [4], Java [38], and AspectJ [39], while others aim to test specifications and models, such as, state charts [40] and state machines [41].

While seen as an effective means, mutation testing is also considered to be expensive [3, 42]. The costs result from executing a huge number of mutants against the test cases. To tackle this problem, several cost reduction techniques have been proposed, which can be clustered into three groups [5]:

- *Do fewer* approaches aim to reduce the number of mutants.
- *Do faster* approaches minimize the execution time of mutation testing.
- *Do smarter* approaches optimize the generation and execution of mutants.

In the context of software-product-line engineering, mutation testing becomes even more expensive, due to the large number of variants that can be derived and have to be mutated [9, 43]. Consequently, applying a mutation operator on a feature does not result in a single mutant. Instead, a set of variants including this feature – and the mutation – can be instantiated.

2.3. *Variability Faults*

For the purpose of mutation testing, mutation operators should ideally mimic real faults. In the context of software-product-line engineering, three types of variability faults occur in practice [8]. We display these in Figure 2 and exemplify them in the following:

1. **Variability Model Faults:** Faults in the variability model comprise faults in either the feature definitions or feature dependencies. Feature definition faults describe situations in which the defined set of features differs from the intended one. For instance, some features may be wanted

but are missing. Feature dependency faults describe inconsistencies in the variability model. To exemplify this, we consider the feature model we show in Figure 1a. As the feature `World` is intended to be mandatory, setting it optional would be a faulty feature-dependency.

2. **Domain Artifact Faults:** Faults in the domain artifacts comprise faults that occur in the implementation of a single feature or in feature interactions. The first case resembles faults in single system development and can rely on existing mutation operators. In contrast, interaction faults only appear in a specific selection of features that results in unintended behavior [12]. For example, we could assume that in Figure 1b the used Strings are stored in variables. If two features use the same variable, one of them could potentially overwrite the stored String before the other feature was able to read it. Thus, the result would be faulty because the interaction of both features does not behave as intended.
3. **Variability Mapping Faults:** The last type of faults appears in the *mapping* between the model and the domain artifacts, which defines the artifacts (code) that belong to a feature. Only with this mapping, a configuration can be composed. Here, code can either be mapped insufficiently, for instance, not all domain artifacts are mapped to the correct features, or unnecessarily, for instance, additional domain artifacts are mapped to features for which they are not required. For our example in Figure 1, we can imagine that a domain artifact of feature `Wonderful` is not mapped to this feature (insufficient) or is additionally mapped to another one (unnecessary; e.g., `Beautiful`).

While several mutation operators for variability models have been proposed [7, 44, 45, 46], the mapping and domain artifacts are addressed less frequently [8, 39]. In particular, we are only aware of few mutation operators for the variability mechanisms of composition-based approaches. Those address specific problems of the underlying implementation technique, such as the *fragile point-cut problem* in aspect-oriented programming [39, 47, 48].

3. MUTATION OPERATORS FOR FEATURE-ORIENTED PROGRAMMING

In this section, we propose a set of seven mutation operators that apply mutations in feature-oriented programming to potentially inject variability faults and illustrate them on simple examples resembling Figure 1. We derive these operators from existing works that discuss problems in composing systems. Additionally, we adapt operators for preprocessor-based or other composition-based software product lines to resemble variability faults in feature-oriented programming. For clarity, we use the following terms:

- **Base Method** refers to the method that is refined by a feature. If we consider Figure 1, such a base method would be `print` in the `Hello` feature (top left in Figure 1b). The other features refine this method by adding further String outputs to the console.
- **Feature Method** refers to a refining method. In Figure 1b, this is illustrated in the top right. Here, the `print` method of feature `World` refines the corresponding base method by adding the String `World!` to the console output.
- **Refinement Call** refers to the call that specifies how base and feature method are composed – the `original` method in feature-oriented programming. For example, we display this in Figure 1b: As the refinement call appears before the refinement itself, the code is appended to the base code and its previous refinements. Thus, the call order represented at the bottom of Figure 1b is created. If the refinement call would be after the other statement, `World!` would appear first in the console.

In the following, we propose mutation operators for *domain artifacts* and *mapping* in feature-oriented programming. Still, adaptations of these operators are also applicable for other composition-based approaches. We remark that these mutation operators can also result in invalid code, due to syntactical errors that are created during the composition of a variant. For each operator, we briefly describe why this can happen, but discuss this issue more in details in Section 6.

3.1. Domain Artifact Operators

In the following, we propose three operators that change the source code of a feature. The goal of these operators is to mimic faults in implementing variability. To this end, we adapt previously proposed mutation operators for preprocessor-based software product lines [8, 49]. While these follow a different mechanism, by excluding code from the base instead of composing modules, the base concepts of injecting faults are still applicable.

DFM - Delete Feature Method The DFM operator is an adapted version of *Removing Complete ifdef Block* [8]. An entire feature method is removed, resulting in missing code for corresponding configurations. Due to this removed source code, variable definitions, refined method behavior, or references can be missing and result in faults. We remark that the whole method, not only its content, has to be removed to prevent overriding. Otherwise, the empty method may completely replace the base method because of a missing refinement call. This can also result in faults, but not the ones we intend to inject with DFM.

We exemplify the DFM operator in Figure 3. If we remove this feature method, the code will not appear in the final variant. Considering our example in Figure 1, we can apply this operator to the `World` feature displayed on the top right in Figure 1b. Removing the `print` method there would mean that the corresponding String would not be printed in any variant.

```
...
void print(String msg) {
  original(msg);
  System.out.print(msg);
}
...
```

(a) Original code.

```
...




...
```

(b) DFM mutant.

Figure 3. DFM mutation operator.

MRC - Move Refinement Call For preprocessor-based systems, the *Moving Code around ifdef Blocks* operator aims to inject faulty placed variability [8]. As a result, corresponding variants may behave differently or are even unusable, due to additional conditions. Based on this mutation operator, we derive our MRC operator for feature-oriented programming that moves the refinement call to a different position.

We display an example of this operator in Figure 4, in which we move the `original` method. Consequently, operations on the global variable `i` that are done in the refinement, may not be performed correctly as the corresponding `original` call is wrongly placed. Considering the example from Figure 1, the MRC operator changes the order in which the text is printed out. Especially, this can be problematic if several features interact and refine the same base method; thus, relying on the same structure [9, 49, 50].

```
int i;
...
void print(String msg) {
  original(msg);
  System.out.print(msg + i);
}
...
```

(a) Original code.

```
int i;
...
void print(String msg) {
  System.out.print(msg + i);
  original(msg);
}
...
```

(b) MRC mutant.

Figure 4. MRC mutation operator.

UO - Unintended Overwrite In feature-oriented programming, features refine or overwrite an existing code base. While this is the desired behavior, it can result in unintended overwrites of methods (similar to the fragile point-cut problem, cf. RBM operator) and variables, which are addressed with the UO operator. Overwriting values of variables may be unnoticed by developers because the features are physically separated into different modules, wherefore other changes to the variable are not obvious to spot [51, 52].

We display an example of the UO operator in Figure 5, where we set an integer value in a feature method. This can result in two different faults: Firstly, syntactical errors could appear if the value is not instantiated beforehand. Secondly, faulty behavior may be injected as the replacing value could distort the intended execution.

```
...
void print(String msg) {

  original(msg);
  System.out.print(msg);
}
...
```

(a) Original code.

```
...
void print(String msg) {
  i = 0;
  original(msg);
  System.out.print(msg);
}
...
```

(b) UO mutant.

Figure 5. UO mutation operator.

3.2. *Variability-Mapping Operators*

With preprocessors, features are mapped to the domain artifacts by directly annotating the corresponding source code [8, 12]. As a result, mappings can be mutated by editing these annotations on source-code level. In contrast, composition-based approaches map features and their code implicitly, relying on the artifacts' structure and call references [12]. For instance, in aspect-oriented programming [12, 22] join points and point-cuts are used to define variable behavior. A point-cut defines the variable behavior (e.g., a feature) that is executed at specific join points if the configuration includes this variability. Feature-oriented programming relies on refinements based on the `original` method and the program structure (i.e., classes and methods), utilizing a similar concept as inheritance [12, 20]. Variability-aware mutation operators are necessary to simulate potential faults in this implicit mapping, wherefore we propose the following four.

ARC - Alternate Refinement Call For feature-oriented programming, it is necessary to specify how a feature is composed into the base code. The code of a base method is used at the position of the refinement call in the feature method. In contrast, if a refinement call is missing, the base method is completely replaced (overwriting). We propose the ARC operator to remove – similar to *Statement Deletion* [4] – or add such a statement in feature methods. Thus, this operator changes how a variant is composed, potentially injecting faults into a variant's behavior.

We show a corresponding example in Figure 6. Here, the feature on the left side would include the base code of an existing `print` method at the position of the `original` method. In contrast, the feature on the right side would completely replace an existing `print` method. We remark that including a refinement call (mutating Figure 6b to Figure 6a) can also result in syntactical invalid variants if there exists no base method `print` that can be refined.

CFO - Change Feature Order The *feature order* defines the sequence in which features are composed. Because refinements change the existing code, editing this order may result in a different behavior for some variants [12, 20, 33, 34]. Thus, CFO can cause faults as classes or methods are overwritten or refined incorrectly.

In Figure 7, we illustrate an example based on Figure 1 for a configuration with feature-oriented programming. The order in Figure 7a is the correct version and would be used to compose the code

```
...
void print(String msg) {
  original(msg);
  System.out.print(msg);
}
...
```

(a) ARC mutant refining a method.

```
...
void print(String msg) {

  System.out.print(msg);
}
...
```

(b) ARC mutant overwriting a method.

Figure 6. ARC mutation operator. In Figure 6a, the `original` method is removed, resulting in the faulty mapping in Figure 6b. The other way around, the `original` method is added.

we show in Figure 1. In contrast, in Figure 7b any variant including `Beautiful` would be incorrect, as the order of Strings in the console output would be wrong. Furthermore, CFO can cause syntactical errors resulting in broken variants, for example, if base methods are composed at last. For instance, we could assume that in Figure 7b the feature `Hello` is at the position of `World`. Any variant including `Wonderful` would then be faulty, because the refinement call (`original`) would refer to a non-existing base method, which is only composed afterwards. Depending on the used composer, this may result in missing code or syntactical errors, depending on whether the composer handles such issues.

```
Hello
Wonderful
Beautiful
World
```

(a) Original feature order.

```
Hello
Wonderful
World
Beautiful
```

(b) CFO mutant.

Figure 7. CFO mutation operator.

RBM - Rename Base Method The *fragile point-cut problem* is commonly known in aspect-oriented programming and tested with mutants [39, 47, 48, 53, 54]. It includes situations in which a base method is renamed but not the corresponding feature methods, wherefore the mapping between them is lost. With the RBM operator, we propose to adapt this scenario for feature-oriented programming by exchanging the names of base methods with identical signatures or by renaming a base method. Thus, features are incorrectly mapped and will result in mutated variants.

We illustrate an example for this operator in Figure 8. The method `print` is a base method and, due to the changed signature in Figure 8b, feature methods that refine `println` would be composed here. In contrast, feature methods originally mapped to the `print` method would be additionally included instead of refining this base method. Again, the operator may lead to syntactical errors, due to refinement calls that are not mapped to any method.

```
...
void print(String msg) {
  System.out.print(msg);
}
...
```

(a) Original code.

```
...
void println(String msg) {
  System.out.print(msg);
}
...
```

(b) RBM mutant.

Figure 8. RBM mutation operator.

RFM - Rename Feature Method The fragile point-cut problem also refers to situations in which new methods are introduced that share their name with existing ones, potentially resulting in unintended refinements. With our RFM operator, we propose to also adapt this situation by renaming a feature method. Thus, the feature method will not refine the intended base method.

Considering the mutant we show in Figure 9b, the `println` method would now refine another one. Thus, two faults can appear: As intended, the correct method is not refined (`print`), but also a wrong method (`println`) may be refined. If no method `println` exists in the base code, the resulting variant may be syntactically incorrect. Again, if the composer handles such issues, the feature method may be just included in the code.

```
...
void print(String msg) {
  original(msg);
  System.out.print(msg);
}
...
```

(a) Original code.

```
...
void println(String msg) {
  original(msg);
  System.out.print(msg);
}
...
```

(b) RFM mutant.

Figure 9. RFM mutation operator.

3.3. Summary

The described operators mutate either the domain artifacts directly or how features in the variability model are mapped to these domain artifacts. As a result, the variants that can be generated with each mutant may be composed differently, potentially injecting faults into their behavior. However, syntactically incorrect variants can also be generated, a problem we further discuss in Section 6. With the proposed mutation operators, we aim to inject variability faults into systems that have been implemented with feature-oriented programming. We derived them from known problems of composition and from operators for preprocessor-based variability. Thus, we argue that they represent a reasonable set of mutation operators to introduce variability faults, potentially reflecting those made by developers.

4. STUDY DESIGN

In this section, we report the design of our study, in which we apply mutation testing on software product lines implemented with feature-oriented programming. We describe our *subject systems*, *tool setup*, *research objectives*, and *methodology*.

4.1. Subject Systems

Unfortunately, real-world systems for testing software product lines are rare [43, 55], often due to missing test cases. For this reason, we select four systems from a set of available open-source case studies[‡] and examples provided in FeatureIDE [56]. All these software product lines are implemented with feature-oriented programming, using Java and FeatureHouse [23, 32]. In Table I we provide an overview of the software product lines' characteristics. We argue that these four software product lines are a reasonable selection to gain insights into mutation testing for feature-oriented programming and our proposed operators.

Hello-World SPL One of the systems we use for our evaluation is the introduced Hello-World software product line (Hello-World SPL). We showed the basic implementation in Section 2, comprising the feature diagram and code. For our evaluation, we use this example version from FeatureIDE [56] that includes 4 test cases. These tests assess whether the String defined in each

[‡]http://spl2go.cs.ovgu.de/

Table I. Considered software product lines (SPL).

Name	Features	SLOC	Variants	Test Cases
Hello-World SPL	4	24	3	4
Union-Find SPL	8	147	6	5
Prop4j SPL	14	2,051	31	63
Graph SPL	27	2,851	157	-

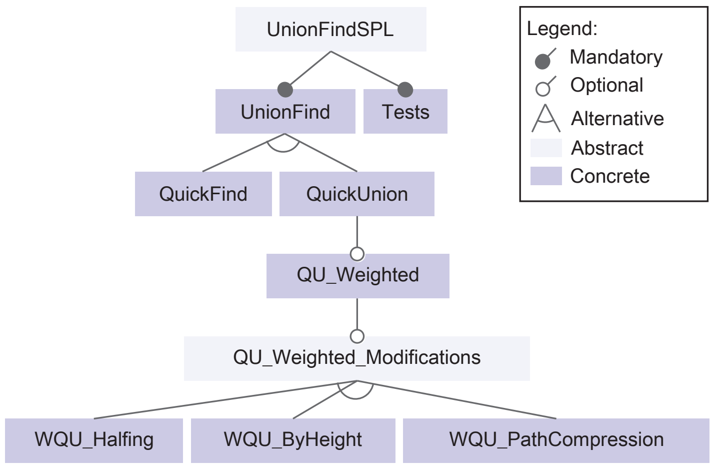


Figure 10. Feature diagram for the Union-Find SPL.

feature is part of the actually derived variant. Overall, we can utilize 3 variants: Including the feature `Wonderful`, the feature `Beautiful`, or none of both.

Union-Find SPL The Union-Find software product line (Union-Find SPL) implements union-find algorithms, as described by Sedgewick [57]. It contains 8 concrete features that we display in the feature diagram in Figure 10. The Union-Find SPL contains a base algorithm (i.e., `UnionFind`) and test cases that must always be selected. Its base algorithm can be refined in multiple steps to utilize different computations. Overall, we can compose 6 unique variants, each changing the source code. Here, we use the selected feature that is furthest down in the diagram as name for a variant: `QuickFind`, `QuickUnion`, `QU_Weighted`, `WQU_Halfing`, `WQU_ByHeight`, and `WQU_PathCompression`. We select this software product line because of its limited size of 147 SLOC, enabling us to provide detailed examples. As a result, it is well suited to comprehensively illustrate mutation testing og feature-oriented programming.

Furthermore, five different test cases are implemented for the Union-Find SPL. These unit tests assess the following aspects of any algorithm, independent of the selected features:

- No entries are connected after initializing the union-find structure.
- Entries are connected with the corresponding method.
- Each entry is reflexive.
- All connections are symmetric.
- Connections are transitive.

These test cases are not designed to evaluate specific features or their interactions. Instead, they test the actual outcome of the algorithms, which should not differ. Consequently, they are not ideal to test refinements that, for example, improve the performance or add attributes to variants. Still, this means that we can apply a mutation testing tool without further adaptations, as all unit tests have to be executable. In contrast, if a test assesses a specific feature that is selected only in some configurations, we would need to make the test also configurable and address potential dependencies.

Prop4j SPL The Prop4j software product line (Prop4j SPL) is a variable re-implementation of arbitrary propositional formulas. Overall, this system contains 2,051 SLOC in 14 features, based on which 31 different variants can be instantiated. Moreover, 63 working unit tests, which also evaluate different features, are provided for this software product line. Two limitations of these test cases is that they (i) rely on the existence of specific features and (ii) depend on each other: If we do not select all features, we have to remove test cases. As a result, we have to remove further test cases that rely on the former ones. Unfortunately, we find that – due to these dependencies – only the full configuration (all features are selected) allows us to run the provided test cases. Overall, the Prop4j SPL enables us to assess more variability, corresponding interactions, and a larger test suite, but we are limited to a single variant.

Graph SPL Finally, we use an artificial software product line developed for evaluation purposes [12, 58]. The Graph software product line (Graph SPL) contains 27 concrete features that allow 157 different variants. While the Graph SPL comprises 2,851 SLOC, it does not include any test cases. To use this system for our experimentation, we automatically generate unit tests, as we describe in Section 4.4

4.2. Tool Setup

For our study, we rely on FeatureIDE [56], an Eclipse[§] plug-in for software-product-line engineering. FeatureIDE provides all functionalities that we require to integrate, configure, and investigate our subject systems. More precisely, this plug-in allows us to import the software product lines into FeatureHouse projects to then analyze and derive variants from them.

We use two generators to automatically create test cases for the Graph SPL: Firstly, we apply Randoop [59], which is a state-of-the-art tool to create random test cases [60]. Secondly, we apply EvoSuite [61], a test case generator that achieved the highest scores in several competitions [62, 63]. Thus, we argue that we use suitable tools to generate unbiased test cases for our evaluation.

To apply conventional mutation testing, we use the Eclipse plug-in Pitclipse[¶]. It integrates the PIT system[‖] [64, 65] into the Eclipse IDE. PIT provides a set of conventional operators, can automatically create mutants, and run unit tests for Java programs. In addition, it already contains optimization strategies that limit the number of necessary test runs. For the conventional mutation operators, we rely on those that are activated by default in PIT:

- *Conditionals Boundary* replaces relational operators with their boundary counterpart (e.g., <= becomes <).
- *Increments* changes increment commands to decrements and vice versa on local variables (e.g., `i++` becomes `i--`).
- *Invert Negatives* removes negations of integers and floating numbers (e.g., `-i` becomes `i`).
- *Math* replaces binary arithmetic operations with another based on a predefined set (e.g., + becomes − and vice versa).
- *Negate Conditionals* changes conditional statements to their counterparts (e.g., == becomes `!=`).
- *Return Values* switches the returned value of a method depending on the return type (e.g., any `Object` becomes `null`).
- *Void Method Call* removes any call to a void method.

These operators are not designed to be applied in a variability-aware context, but may still introduce corresponding faults [8, 9]. We remark that we manually apply our proposed mutation operators, due to a lack of mature tooling.

[§] https://eclipse.org/
[¶] https://marketplace.eclipse.org/content/pitclipse
[‖] http://pitest.org/

4.3. Research Objectives

Mutation operators aim to introduce faults into a system that are then exposed by the test suite. In order to investigate mutation testing for feature-oriented programming, we use conventional and our proposed mutation operators. For this purpose, we investigate the following four research objectives:

RO_1 *Assessing whether conventional and our proposed operators inject faults into variable code.*
With this research objective, we aim to identify whether conventional and our proposed variability-aware mutation operators inject faults into variable code. This way, software product line specific variability faults can be resembled during mutation testing. We investigate this objective as an exploratory study in which we analyze the source code of the Union-Find SPL and Prop4j SPL to exemplify the applied mutations. We use these two systems, due to their size and number of variants that allow a detailed manual analysis. The results we describe in Section 5 provide first insights into the usability of the different operators and their behavior.

RO_2 *Assessing equivalent and redundant mutants of our proposed mutation operators.*
Equivalent mutants are those that comprise identical behavior compared to the original system, despite a mutation being applied. In contrast, redundant mutants are either (a) equal to each other (*duplicated* [16]) or (b) are always jointly killed (*subsumed* [66]). Within this article, we consider redundancy in the context of duplication. Equivalent as well as redundant mutants result in additional and unnecessary test runs, increasing the costs of mutation testing [3, 14, 67]. With this research objective, we aim to assess to which extent the mutants we generate with our proposed operators are equivalent or redundant. In an initial empirical study, Carvalho et al. [15] show that almost 40% of the mutants for preprocessor-based mutation operators are equivalent. As we expect, we also find that the ratio of equivalent and redundant mutants for our proposed mutation operators is quite high and demands for cost reduction techniques.

RO_3 *Comparing conventional and variability-aware mutation operators.*
Our operators we proposed in Section 3 are designed to inject variability faults. The question that arises is, whether these injected faults differ from those of conventional operators and, thus, result in different performances for the considered test cases. For this research objective, we adapt the approach of Laurent et al. [60] and design an experiment to compare the effectiveness of tests for both types of operators. The results we show in Section 5 illustrate the applicability of our proposed operators to resemble variability faults.

RO_4 *Identifying challenges in mutation testing of software product lines.*
Applying mutation testing in variable systems results in new challenges, for example, increasing costs due to testing the same code structures multiple times [9]. Still, testing variability is essential for software product lines, as feature interactions are a potential source for faults [9, 49, 50]. Because testing all variants is unfeasible, several approaches, for instance, combinatorial interaction testing [68], have been proposed to reduce the number of variants while covering all interactions to a certain degree. For preprocessor-based systems, sampling and static analysis are promising approaches to reduce costs of mutation testing [9]. Based on our findings for the aforementioned objectives, we discuss open challenges in the context of mutation testing for software product lines and especially feature-oriented programming in Section 6.

By analyzing these research objectives, we aim to provide detailed insights on our proposed mutation operators. Furthermore, we discuss open challenges that we identify during our study.

4.4. Methodology

During our study, we utilize the aforementioned subject systems as follows: Firstly, we investigate the impact of conventional and variability-aware mutation operators in the context of variability (RO_1). Therefore, we analyze the Union-Find SPL and Prop4j SPL, which contain several variable features, but can still be manually analyzed with reasonable effort. Using PIT, we automatically mutate and test all possible variants with conventional operators. Additionally, we manually apply our proposed mutation operators. We assess the results based on the test cases and by manually investigating the injected mutations. The purpose of this procedure is to illustrate that the operators actually inject variability faults that may not be killed by tests. For instance, the CFO operator does change the source code of some variants, but the test cases do not fail.

Secondly, to investigate equivalent and redundant mutants (RO_2), we analyze variants of the Graph SPL that we create by mutating three features. To this end, we manually compare each mutated variant with its original version, using a similar approach as Carvalho et al. [15]—who mutate and analyze a set of configurable files in a system. Here, we aim to identify the number of equivalent and redundant mutants that our proposed operators create. This helps us to reason about the costs associated with our proposed mutation operators and their applicability in real-world scenarios.

Thirdly, we conduct an experiment following the approach of Laurent et al. [60] to assess the quality of our proposed operators (RO_3). Laurent et al. compare two sets of mutation operators by identifying test cases that kill mutants created with one set and applying these tests on the mutants of the other set, and vice versa. For this purpose, we use the Hello-World SPL, for which we apply all operators in all variants on the same tests, and the Graph SPL, for which we analyze the same three features as for our second research objective. We use the Hello-World SPL only to exemplify the differences between both types of operators, as it provides manually created tests that kill all conventional mutation operators. In contrast, for the Graph SPL we have no set of tests that kills all conventional operators, but can generate test cases that are unbiased by humans.

For both systems, we apply the following approach. At the beginning, we identify two test sets: The first set (T_{sub}) is a subset of all tests, including only those that killed mutants created by conventional operators. The second set (T_{all}) includes all test cases. We apply both sets on our proposed variability-aware mutation operators. This way, we aim to show that the mutants are killed by different test cases. Thus, the proposed operators would inject other faults and would be a reasonable addition to the conventional ones.

Finally, we describe problems we identify while setting up and conducting our study (RO_4) in Section 6. Based on these problems, we discuss challenges of effective and efficient mutation testing of software product lines, especially based on feature-oriented programming. More precisely, we focus on equivalent and redundant mutants, cost reduction techniques, usability of mutation operators, tooling, as well as test case design.

5. RESULTS AND EVALUATION

In this section, we report and discuss the results of our study. We present the outcome of applying conventional and our variability-aware mutation operators to assess our first three research objectives.

5.1. Assessing whether conventional and our proposed operators inject faults into variable code.

For our first research objective (RO_1), we assess how conventional and our proposed mutation operators perform on software product lines. To this end, we use the Union-Find SPL and Prop4j SPL. We analyze whether the injected mutations are actually killed and represent variability faults.

Results for Conventional Operators In Table II, we present the results of applying conventional mutation operators on all variants of the Union-Find SPL and the full configuration of the Prop4j SPL. For the Union-Find SPL, we see that for each variant at least 90% of the source code is covered by the test cases. Depending on the size of each variant, also the number of mutants and tests increases. However, while more mutants are created, only few more are killed by the test suite. In both software product lines, approximately 67.9% of mutants are killed on average among all variants. While the test cases cover most lines (except for the Prop4j SPL), they do not kill many mutants, especially when these are introduced by refinements.

For a detailed analysis, we display the source code of the `QuickUnion` variant and its injected mutations in Listing 1. We show the applied mutation, a corresponding identifier, and the used operator. Triangles filled with black indicate mutants that are killed, while empty triangles indicate survivors. We can see that the source code is mutated in every method and especially at loops. The two surviving mutants result from the fact that no test covers the variable `count`.

Table II. Results for mutation testing with conventional operators.

System	**Variant**	**LC**		**MC**		**# Tests**
		#	**%**	**#**	**%**	
Union-Find SPL	QuickFind	13/14	93	11/13	85	15
	QuickUnion	16/17	94	9/11	82	17
	QU_Weighted	20/21	95	11/18	61	33
	WQU_Halfing	21/22	95	11/18	61	33
	WQU_ByHeight	28/31	90	13/23	57	43
	WQU_PathCompression	25/26	96	12/19	63	31
Prop4j SPL	Full	490/691	71	337/510	66	778

LC: Line Coverage; MC: Mutation Coverage

Listing 1. QuickUnion and the injected faults.

```
public class UnionFind {
  private int[] id;
  private int count;

  public UnionFind(int N) {
    count = N;
    id = new int[N];
    for (int i = 0; i < N; i++)
        ▶ for (int i = 0; i >= N; i++)        (1) Negative Conditional
        ▶ for (int i = 0; i <= N; i++)        (2) Conditionals Boundary
        ▶ for (int i = 0; i < N; i--)         (3) Increments
      id[i] = i;
  }

  public int count() {
    return count;
        ▷ return 0;                           (4) Return Values
  }

  public boolean connected(int p, int q) {
    return find(p) == find(q);
        ▶ return find(p) == 0;                (5) Return Values
        ▶ return 0 == find(q);                (6) Return Values
        ▶ return find(p) != find(q);          (7) Negative Conditional
  }

  public void union (int p, int q) {
    int i = find(p);
    int j = find(q);
    if (i == j) return;
        ▶ if (i != j) return;                 (8) Negated Conditional
    id[i] = j;
    count--;
        ▷ count++;                            (9) Increments
  }

  public int find(int p) {
    while (p != id[p])
        ▶ while (p != id[p])                  (10) Negative Conditional
      p = id[p];
    return p;
        ▶ return 0;                           (11) Return Values
  }
}
```

We compare the mutations illustrated in Listing 1 to all other variants and display the results in Table III. Note that, in this case, we consider a mutant to be redundant if it was structurally at the same position and identical on lexical level (i.e., a *type-1 code clone* [69]). To this end, we only

Table III. Distribution of conventional mutants in the Union-Find SPL. The identifiers refer to Listing 1.

Variant	Mutant										
	1	2	3	4	5	6	7	8	9	10	11
QuickFind	✓	✓	✓	✗	-	-	-	-	✗	-	-
QuickUnion	✓	✓	✓	✗	✓	✓	✓	✓	✗	✓	✓
QU_Weighted	✓	✓	✓	✗	✓	✓	✓	✓	✗	✓	✓
WQU_Halfing	✓	✓	✓	✗	✓	✓	✓	✓	✗	✓	✓
WQU_ByHeight	✗	✓	✓	✗	✓	✓	✓	✓	✗	✓	✓
WQU_PathCompression	✓	✓	✓	✗	✓	✓	✓	✓	✗	-	-

✓ Killed; ✗ Survived; - Not Existing

Table IV. Number of test cases that failed when applied on the proposed mutation operators.

System	Variant	Tests	Failed Test Cases					
			DFM	MRC	UO	ARC	CFO	RBM	RFM
Union-Find SPL	WQU_ByHeight	5	3	0	3	0	0	3	3
Prop4j SPL	Complete	63	27	-	-	-	0	20	23

aim to identify the operators that result in redundant mutations but not equivalent mutants that are identical to the original system (cf. RO_2).

We can see that all variants that are refinements of `QuickUnion` (only `QuickFind` is not a refinement, see Section 4) share almost all these mutants. In all variants, the results for conventional mutation operators differ only for three cases: One mutant is not killed in `WQU_ByHeight` and two are not created in `WQU_PathCompression`. In all other cases, the applied mutation operators as well as the results of the test suites are identical. As a comparison with Table II shows, several additional mutants in refinements are created but rarely killed, because the test cases' mutation coverage in the corresponding variants rarely includes more than the 11 mutants we display.

Discussion Considering the results we show in Table II and Table III, we find that conventional mutation operators can inject faults into variable parts. Thus, they can be used in feature-oriented programming to potentially also test variability faults. For example, applying the Statement Deletion [4] operator on domain artifacts could result in our proposed ARC operator. However, they do not inject many faults that are connected to the underlying concept of variability (i.e., the composition mechanism of feature-oriented programming). Overall, conventional operators can resemble variability faults, but are considerably limited in this regard. This is rarely surprising, as conventional mutation operators are not designed to inject this kind of faults.

Results for Variability-Aware Operators Considering our proposed mutation operators, we find that most created mutants are not killed by the test cases. Still, we apply them in each software product line and investigate the changes that are injected into the source code. We illustrate exemplary results for a single application of each operator in one variant of each software product line in Table IV. Dashes indicate that we found no code structure on which we can apply the mutation operator.

For four of our mutation operators, test cases fail when we inject the corresponding mutation:

DFM	We remove a feature method in the Union-Find SPL, wherefore 3 test cases fail. This is due to missing refinements that ensure a proper functionality of the union-find structure. In the Prop4j SPL, we delete a feature method that leads to 27 test cases failing.
UO	In the `WQU_ByHeight` feature of the Union-Find SPL, we change one integer initialization – that uses a method call to do so – to a fixed value (i.e., 0). Consequently, 3 test cases are failing and, thus, exploiting the fault.

Listing 2. Mutation of the CFO operator in the Union-Find SPL (original code in comments).

```
public void union  (int p, int q) {
  int i = find(p);
  int j = find(q);
  if (i == j) return;

  /* if      (ht[i] < ht[j]) id[i] = j;
  else if (ht[i] > ht[j]) id[j] = i;
  else    { id[j] = i; ht[i]++; } */
  if (sz[i] < sz[j]) { id[i] = j; sz[j] += sz[i]; }
  else { id[j] = i; sz[i] += sz[j]; }

  count--;
}
```

RBM We rename a constructor in the Union-Find SPL that is refined by features. As a result, objects are created faulty because variables that are required for further calculations are not available. Similarly, renaming a base method in Prop4j SPL results in 20 failing test cases.

RFM In both, the Union-Find SPL and Prop4j SPL, we rename a method (for the Union-Find SPL the same as for DFM) so that its composition into the base is faulty. Thus, 3 test cases of the Union-Find SPL (identical to DFM) and 23 test cases of the Prop4j SPL fail.

Contrary to these operators, the remaining three operators (i.e., MRC, ARC, and CFO) do not cause faulty behavior in the Union-Find SPL and Prop4j SPL that is killed by the test cases. However, we analyze the resulting source code and find that each of the operators mutates the structure [9] of variants. For example, in Listing 2, we apply the CFO operator and mutate the Union-Find SPL so that the feature `WQU_ByHeight` is composed first (instead of last). In the resulting mutant, 3 SLOC are replaced by 2 others. Still, the test cases do not kill this mutant with wrongly used variables, because the outcome of the method remains the same – which is not the case for all variants and changed feature orders. We can explain this as the feature `WQU_ByHeight` does introduce a more efficient algorithm, but the outcome should of course not change. Instead, the basic version is used, meaning that the mutated variant may behave identical, while maybe not fulfilling performance requirements. We could exploit this issue with test cases by testing for the existence and values of the newly introduced variables (i.e., the `ht` array).

Discussion Regarding our proposed variability-aware mutation operators, we find that they apply reasonable and variability-related mutations into the subject systems. For instance, they can change the structure and behavior of source code, due to wrongly composed variants. As we show in Listing 2, the resulting code resembles variability faults. Still, similar to the conventional operators, our manual analysis shows that several mutants are redundant. This is due to the variants containing the same mutated features and because some of our operators can result in the same changes.

Furthermore, equivalent mutants can be a challenging issue as we find while assessing, for example, the CFO operator. Here, the program is often composed correctly as long as all depending methods are in the correct order (i.e., base methods beforehand and feature methods afterwards). Thus, mutants that are equivalent to the original variant are composed multiple times, an issue we investigate in detail within our next research objective (RO_2).

5.2. Assessing equivalent and redundant mutants of our proposed mutation operators.

Our previous results already indicate that we face redundant and equivalent mutants that can drastically increase the costs of mutation testing. In software-product-line engineering, this issue is arguably even more problematic, due to the variability that allows to instantiate multiple variants [9, 15]. A recent study indicates that variability-aware mutation operators can easily result in almost 40% equivalent mutants [15]. As this may distort our results, we investigate this issue with our second research objective (RO_2): To this end, we manually compare one original variant of the

Table V. Applying our proposed mutation operators on the Graph SPL.

Feature	**RO**	**Mutants**		**Mutation Operators**							**Total**	
				DFM	**MRC**	**UO**	**ARC**	**CFO**	**RBM**	**RFM**	**#**	**%**
		Created		5	2	2	4	26	5	5	49	100.0
		Errors		0	0	0	0	0	3	2	5	10.2
		Applicable		5	2	2	4	26	2	3	44	89.8
WeightedWithEdges	RO$_2$	Equivalent		0	0	0	2	22	0	0	24	49.0
		Redundant		0	1	0	0	3	0	0	4	8.2
		Tested		5	1	2	2	1	2	3	16	32.7
	RO$_3$	Killed	T$_{sub}$	4	0	1	1	1	0	3	10	20.4
			T$_{all}$	4	0	2	1	1	0	4	12	24.5
		Created		3	3	2	2	26	3	5	44	100.0
		Errors		1	0	0	0	0	3	3	7	15.9
		Applicable		2	3	2	2	26	0	2	37	84.1
Connected	RO$_2$	Equivalent		0	0	1	0	2	0	0	3	6.8
		Redundant		0	0	0	0	20	0	0	20	45.5
		Tested		2	3	1	2	4	0	2	14	31.8
	RO$_3$	Killed	T$_{sub}$	0	0	0	1	0	0	0	1	2.3
			T$_{all}$	0	0	0	1	0	0	0	1	2.3
		Created		2	4	2	2	26	3	3	42	100.0
		Errors		0	0	0	0	0	2	3	5	11.9
		Applicable		2	4	2	2	26	1	0	37	88.1
MSTKruskal	RO$_2$	Equivalent		0	0	2	0	2	0	0	4	9.5
		Redundant		0	1	0	0	19	0	0	20	47.6
		Tested		2	3	0	2	5	1	0	13	31.0
	RO$_3$	Killed	T$_{sub}$	0	0	0	0	0	0	0	0	0.0
			T$_{all}$	1	0	0	1	1	1	0	4	9.5

Graph SPL to all mutated variants we can derive by mutating three of its features. We assess if these mutants are equivalent or redundant in behavior. This way, we aim to improve our comparison to conventional operators (cf. RO$_3$) and illustrate the necessity for cost reduction techniques.

Results We show our results for equivalent and redundant mutants in Table V (marked with RO$_2$). Here, we can see that the ratio of both types is quite different among our mutation operators. For example, we already expected that CFO will result in several equivalent and redundant mutants, as changing the feature order is possible for all features (i.e., 26), but will only cause structural changes in some orders. Consequently, for feature `Connected` there are 20 redundant mutants, but they belong to 4 distinct sets: These mutants are only redundant within these sets, not among them. Unsurprisingly, we find fewer equivalent and redundant mutants for our remaining mutation operators. Overall, we find that – besides an average error rate of 12.7% – more than half of all our created mutants is either equivalent or redundant for each feature. Again, this is mainly due to CFO allowing for most mutations while also causing most of the equivalence and redundancy. Thus, this result strongly depends on the actual software product line: More and larger feature methods allow to apply all other operators more often—and fewer features limit CFO in its applicability.

Discussion Our results show that equivalent and redundant mutants can be an issue for our proposed operators. This seems to mainly account for CFO and only occasionally to the others. Thus, we argue that our operators actually mutate the behavior of feature-oriented software product lines, at least considering our subject systems. Nonetheless, we need cost reduction techniques, considering that the high overall ratio of equivalent and redundant mutants drastically increases as a software product line comprises multiple variants. Consequently, instead of mutating a single system, a set of systems has to be tested.

To exemplify the complexity of mutation testing of software product lines, we can assume that each feature in the Graph SPL allows to create 35 mutants (which is even below the number for each of the three features in Table V). We could then inject 945 mutations in the 27 features alone, considering only our proposed operators. If we have to test all 157 variants of the Graph SPL – which only comprises 2,851 SLOC – without cost reduction techniques, we would have 148,365 mutants.

In comparison, using PIT, we can create 142 mutants for one variant of the Graph SPL (cf. Table VI). Assuming this as average, PIT may create 22,294 mutants for the whole Graph SPL. While this is a considerably smaller number (approximately 15%), PIT already includes cost reduction techniques. Consequently, considering the number of syntactical errors as well as equivalent and redundant mutants, our variability-aware mutation operators call for cost reduction techniques – which may decrease the number of mutants to around 30% (cf. Table V). Otherwise, it seems that applying mutation testing on real-world software product lines may be too expensive to be adopted. To this end, different approaches may be used, which we discuss in Section 6. Similar observations are also made in our previous initial analysis on this topic [9] and an empirical study by Carvalho et al. [15] for preprocessor-based software product lines.

5.3. Comparing conventional and variability-aware mutation operators.

To substantiate our previous findings and investigate our third research objective (RO_3), we conduct an experiment on a variant of the Graph SPL. As we have no test cases for this system, we automatically generate 2,263 using Randoop (2,164) [59] and EvoSuite (99) [61]. Then, we define two test sets, adapting the approach used by Laurent et al. [60]: In the first one (T_{sub}), we only include the test cases that first kill a mutant we create with conventional operators, for which we display the results in Table VI. Overall, we find that 35 unit tests kill 40% of all mutants we created with PIT's operators.

While it would be ideal to have a test suite that kills 100% of these mutants, this is hardly possible for the available software product lines. We would have to develop test cases ourselves for all different variants, which could easily be biased. Thus, we decided to use test generators, but these unfortunately do not kill all created mutants.

The second test set (T_{all}) contains all 2,263 test cases. When applying our proposed operators, different outcomes between both test sets will indicate if the operators are useful. For this purpose, we apply all our operators on three different features of the Graph SPL, namely `WeightedWithEdges`, `Connected`, and `MSTKruskal`. In every case, we create all possible mutants and instantiate only a single variant of the Graph SPL that includes all features.

To additionally validate the results, we conduct a second experiment on the small Hello-World SPL, for which the test cases kill all conventional mutants. Here, we are only concerned with gaining detailed insights into all variants of a software product line and validating our previous findings. Arguably, the small size of the Hello-World SPL does not allow for generalizations.

Results We show the results for our proposed variability-aware mutation operators for the three features of the Graph SPL in Table V (marked with RO_3). Here, we compare for each feature how often we can apply each operator, how often this results in syntactical errors, as well as how many mutants are actually tested and killed by each test set. We can see that out of the 135 mutants we create, 17 are syntactically faulty. Furthermore, only 11 mutants are killed by T_{sub} and 17 by T_{all}. While we only consider three features (out of 27) of a single variant (out of 157), we see discrepancies in the results.

For the second experiment we consider the Hello-World SPL that already includes test cases. As we show in Table VII, these tests kill all mutants created by conventional mutation operators. Due to the small size, we can apply all our proposed operators on each of the three variants and investigate the changes in more detail. In Table VII, we only display the outcome for a single variant but the numbers are equal for the other two. Despite the size of this software product line, we again see differences between conventional and variability-aware operators. In particular, 6 out of 24 mutants survive: One CFO and one MRC mutant in each variant. We remark that we are not able to apply the UO operator, due to missing variables.

Table VI. Conventional mutation testing of the Graph SPL.

Class	**LC**		**MC**		**Mutants**			
	#	%	#	%	**K**	**S**	**TO**	**NC**
CycleWorkSpace	23/23	100	5/10	50	5	5	-	-
Edge	26/26	100	10/15	71	10	4	-	-
EdgeIter	3/3	100	2/2	100	2	-	-	-
Graph	120/130	92	20/72	28	19	50	1	2
Main	13/13	100	0/7	100	-	7	-	-
NumberWorkSpace	6/6	100	0/2	0	-	2	-	-
RegionWorkSpace	9/9	100	1/1	100	1	-	-	-
Vertex	54/54	100	28/32	56	18	14	-	-
VertexIter	3/3	100	2/2	100	2	-	-	-
Total	257/267	96	57/142	40	57	82	1	2

LC: Line Coverage; MC: Mutation Coverage
K: Killed; TO: Timed Out; S: Survived; NC: No Coverage

Table VII. Mutation testing of the Hello-World SPL.

Mutant	**Mutation Operators**							
	Conventional	**DFM**	**MRC**	**UO**	**ARC**	**CFO**	**RBM**	**RFM**
Applied	3	1	1	-	2	2	1	1
Killed	3	1	0	-	2	1	0	0
Errors	-	0	0	-	0	0	1	1
Survived	0	0	1	-	0	1	0	0

Discussion As the results for the Graph SPL show, there are discrepancies when applying the two test sets on our proposed operators. In particular, none of both is capable to kill a lot of them. Still, the complete set (T_{all}) kills more mutants than the subset (T_{sub}). Especially for the feature `MSTKruskal`, the subset is not able to kill any mutant while the complete set kills 4 of them. We find such discrepancies for almost all our operators (except MRC). Thus, we argue that they are suitable and necessary to assess variability faults in feature-oriented programming.

The results for the Hello-World SPL substantiate these findings. Here, even a MRC mutant survives despite the test cases killing all conventional mutants. Due to the smaller size, we analyze the test cases and source code to identify why the mutants survive and which test could kill them. We find that the existing tests assess the existence of each substring in the final output, for example, whether the String `wonderful` is part of it if the corresponding feature is selected. For conventional operators and many of our proposed ones this is enough to kill the mutants – and the tests would be suitable if developers could not introduce variability faults. The Hello-World SPL illustrates that if we mutate the way in which features are composed with the CFO or MRC operator, the final output is wrong. For example, the CFO and MRC mutants result in the String `Hello World! wonderful`, which survives as all necessary substrings are included. The changed order and faulty result are not recognized by the test cases. Thus, a variability-aware test would have to assess not only the existence, but also correct order of the Strings, which would kill all mutants created with our proposed and conventional operators. We argue that this exemplifies the necessity for our proposed variability-aware mutation operators.

5.4. Summary

Overall, the results substantiate that our proposed operators are reasonable for mutation testing in feature-oriented programming. They inject variability faults and differ from conventional operators in the type of injected mutation. However, the high number of syntactical errors as well as equivalent and redundant mutants are an issue that has to be addressed before practical applicability seems reasonable. Further studies with improved tooling and real-world software product lines can help to improve scoping the operators and better assess their costs and usability.

6. CHALLENGES

While applying our study, we identified several challenges in the context of mutation testing in software-product-line engineering. In this section, we address our fourth research objective (RO_4) by discussing these challenges and proposing starting points to investigate them.

Equivalent and Redundant Mutants A problem in mutation testing is to decide, which mutants are equivalent to the base system [3, 67] or if mutants are redundant [13]. In the context of software-product-line engineering, both problems become more complex, due to the large set of variants that have to be tested. Because these variants partly comprise the same code, the same operators may be applied without injecting new faults – but are still tested again. Thus, the costs of mutation testing unnecessarily increase. However, it is not ensured that the same mutation will always result in an equivalent or redundant mutant, due to changing configurations. First steps in this direction are performed by Carvalho et al. [15], who illustrate the problem of defining and identifying equivalence in preprocessor-based software product lines.

Cost Reduction Due to the aforementioned problem, it is essential to improve the efficiency of mutation testing for software product lines by adapting cost reduction techniques (cf. Section 2.2) [5]. In the context of preprocessor-based systems, we proposed static analysis and sampling [9] to reduce the number of injected faults on variably code as well as the number of variants under test (do fewer approaches). For example, by using combinatorial interaction testing [68], variants can be sampled to cover each feature interaction only once. However, our proposed operators inject faults that require different analyses. To reduce their costs, do smarter approaches seem most promising. For instance, CFO changes the order in which features are composed – potentially resulting in many equivalent and redundant mutants. Here, it should be possible to use the dependencies defined in a feature model to exclude configurations that do not result in new mutants.

Usability of Operators Furthermore, the usability of our proposed mutation operators is heavily depending on how the developers utilize feature-oriented programming. For instance, in the Prop4j SPL we find no connected base and feature method, wherefore we can not apply the ARC and MRC operators. The results still show that our operators inject either structural or syntactical faults [9]. In a large-scale software product line that utilizes feature-oriented programming in more situations, our operators should be able to create more suitable mutants. To address these points, we need to conduct further experiments, ideally on real-world case studies.

Automation For efficient software-product-line testing, automation is essential [43]. While tools for conventional mutation testing exist, for instance, PIT [64, 65], we are not aware of such a tool for variability-aware mutation testing – except for some prototypical implementations [15]. Without such a tool, the practical application and assessment of mutation testing of software product lines seems unrealistic. However, there are several tasks that have to be included, such as, identifying suitable mutation operators, cost reduction techniques, and automated configuring. An integration of mutation testing into software-product-line engineering tools seems to be the most promising approach. For example, including PIT into FeatureIDE [56] and extending it with our proposed operators, as well as including cost reduction, may be a first step in this direction.

Test Cases Considering test cases, we identify two issues: Firstly, we find that some of the available test cases only exploit faults that are injected into base features, for example, in the Prop4j SPL.

However, this is not a problem of the mutation operators, but the test cases themselves. The tests only cover the general behavior of the variants and not variant-specific implementations. Thus, it is not possible to identify faults that are injected into features. A more detailed granularity of test cases seems necessary to cover, for instance, exchanged algorithms or performance optimizations.

Secondly, considering further variants of the Prop4j SPL, we also find that the test cases are fixed and depending on each other. To facilitate mutation testing of software product lines, the test cases should also be variable and, thus, configured according to the generated variant. For this, the test cases have to be self-contained for different configurations [70], as they are in the Hello-World SPL. This is hardly a problem that only applies in the context of mutation testing, but seems to account for testing software product lines in general.

Summary Overall, we identify five major challenges in mutation testing of software product lines. In particular, the main reason for these challenges is the increasing complexity when combining both approaches: Mutation testing and software-product-line engineering. Each on its own already requires lots of effort, due to the high numbers of derivable system – shortly existing mutants that are only tested and actual variants that are distributed to customers.

7. THREATS TO VALIDITY

A threat to the external validity of our evaluation are the small subject systems. However, all software product lines that we are aware of are not well-suited, because they are either small, provide no test cases, or are closed source. This is a common problem in evaluating software-product-line testing, resulting in significant costs and efforts [43, 55]. While a more extensive evaluation may provide further insights, we were able to show that our operators inject faults and that mutation testing of software product lines poses several challenges. Thus, we argue that this threat does not undermine our results.

A threat to the internal validity are the used mutation operators. Other sets of mutation operators may inject more faults that would be identified by the test suite. For conventional operators, we used the default set of PIT [64, 65]. We derived our proposed operators from known problems of composition, from operators for preprocessor-based software product lines, and the corresponding literature. Thus, we argue that we used suitable mutation operators that do not threaten the results of our evaluation.

Another threat to the internal validity of our work is our manual analysis process. We are not aware of any tool that allows to apply mutation testing on feature-oriented programming. Consequently, we had to rely on this manual process. To avoid faults, we carefully replicated our analysis and checked the results multiple times.

Considering the reliability, we argue that any researcher can replicate this study on their own. We provided all details that are necessary for this and all our tools as well as subject systems are publicly available. There may occur discrepancies if different versions are used, for instance, when PIT is further improved with cost reduction techniques. As this study is – to the best of our knowledge – one of the first on mutation testing of software product lines, we encourage the research community to investigate other systems and consolidate the results.

8. RELATED WORK

In this section, we discuss related works with focus on mutation testing of software product lines and approaches to improve their efficiency. Previously, we [8] proposed mutation operators that are designed to inject variability faults into preprocessor-based software product lines. These operators cover the three layers, model, domain artifacts, and the mapping between them. We partly based on this work, as we use the defined layers and derive some of our operators from those proposed in this work. However, in this article we propose new mutation operators for feature-oriented programming and provide a detailed evaluation. Furthermore, we [9] also investigated how to minimize the costs apply mutation testing on variable systems by reducing the number of mutants. To achieve this goal, we proposed to apply sampling and static variability analysis techniques. While these are also applicable for feature-oriented programming, with respect to static analysis techniques, a suitable adaptation is required.

Carvalho et al. [15] are to our knowledge the first to also investigate the same types of mutation operators. They focus on empirically analyzing equivalent mutants in preprocessor-based software product lines. The results indicate that almost 40% of the generated mutants are equivalent, which aligns to our own findings. In addition, the authors are developing #MUTAF, a tool to apply mutation testing on their subject systems. While this study is closely related to ours, Carvalho et al. focus on a different implementation technique, investigate few files from open-source projects, and do not propose new mutation operators.

Arcaini et al. [45] propose to mutate feature models in order to find faults and define corresponding operators. Applying these operators can mimic mistakes developers make during the design of feature models. In addition, Henard et al. [44] propose to mutate the propositional formula of a feature model in order to inject faults. Lackner et al. [7] measure test case capability to detect faults for software product lines. To this end, they propose model-based mutation operators. Reuling et al. [46] propose mutation operators to evaluate the effectiveness of mutation testing on variants of a software product line. However, the aforementioned approaches focus only on mutating feature models. Thus, these works are complementary to ours as they focus on a different layer, but can be applied to composition-based software product lines. In this article, we propose special mutation operators for feature-oriented programming, focusing on domain artifacts and their mapping to the feature model Furthermore, we investigate the feasibility of conventional operators in software product lines.

Devroey et al. [19, 71] mutate models of a software product line based on feature transition systems. In their approach, they aim to limit the number of test cases to reduce costs. This work is related to ours, as it focuses on reducing testing costs and can complement our approach. However, in contrast to mutating transition systems, we focus on the implementation and mapping layers directly.

For aspect-oriented programming, several works [39, 53, 54] propose operators and techniques for mutation testing. These works focus on testing point-cuts by strengthening or weakening them. Consequently, the authors address variability mapping faults in particular. These works are closely related to ours as they propose mutation operators for another composition-based approach. However, they limit their investigations on one detail (point-cuts), wherefore we complement these works by considering a different implementation technique and additional types of faults.

Regarding cost reduction of mutation testing, several approaches have been proposed to limit the number of mutants or optimize the execution [3]. To reduce the number of mutants, Budd [72] and Zhang et al. [73] sample subsets of these mutants and show that small samples (10% and 5% respectively) are sufficient to achieve high accuracy with the full mutation score. Selecting a sample of mutations has been further investigated in other works [17, 74, 75, 76, 77] and some of these approaches may be used to reduce testing costs in software product lines. In contrast, we focused on mutation operators and whether they can exploit variability faults, but less on their efficiency.

9. CONCLUSIONS

Mutation testing is an established approach to evaluate the quality of test cases by assessing their ability to detect faults. The effectiveness of mutation testing depends mainly on the mutation operators that are used to automatically generate mutants. Ideally, these mutants should contain faults similar to those caused by mistakes of developers. For software product lines, most existing approaches focus only on mutating variability models and ignore the source code, which is the main source of faults.

In this article, we proposed mutation operators that are especially designed for feature-oriented programming. These inject changes that are connected to the mapping of variability model to the source code and the source code itself. We evaluated the proposed and conventional operators using four open-source software product lines and discussed open challenges. The results show that:

- Conventional mutation operators inject faulty behavior into features and, thus, are suitable to exploit variability faults. However, they are limited in this regard, as they are not designed to mutate variability mechanisms and their specific characteristics.
- Our proposed variability-aware operators do inject mutations that are connected to the variability of feature-oriented programming. Thus, they create mutants that are suitable to assess test cases on their ability to exploit variability faults.
- Equivalent and redundant mutants are an important concern in mutation testing that can drastically increase in software-product-line engineering. In particular, we find that some types of operators (i.e., CFO) may drastically increase the costs of mutation testing, due to an increasing complexity.
- Comparing both types of operators, we find that our variability-aware operators are more suitable to mutate the composition mechanism and variable behavior of such software product lines. Thus, both types are reasonable to apply in feature-oriented programming and our proposed operators fulfill their goal.
- In the context of variability-aware mutation testing, adapted approaches for automation, cost reduction, and test case design as well as further studies are necessary.

In future work, we aim to focus on extending the available tooling for mutation testing of software product lines. Furthermore, we will investigate how variability analysis and sampling can be used to adapt existing cost reduction techniques. Finally, we aim to conduct further experiments to verify the reported results. To this end, especially the availability of suitable subject systems is a challenge.

ACKNOWLEDGMENTS

This work is supported by the German Research Foundation (LE 3382/2-1, SA 465/49-1) and Volkswagen Financial Services AG.

REFERENCES

1. Hamlet RG. Testing Programs with the Aid of a Compiler. *IEEE Transactions on Software Engineering* 1977; **SE-3**(4):279–290, doi:10.1109/TSE.1977.231145.
2. DeMillo RA, Lipton RJ, Sayward FG. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* 1978; **11**(4):34–41, doi:10.1109/C-M.1978.218136.
3. Jia Y, Harman M. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 2011; **37**(5):649–678, doi:10.1109/TSE.2010.62.
4. Agrawal H, DeMillo R, Hathaway R, Hsu W, Hsu W, Krauser E, Martin RJ, Mathur A, Spafford E. Design of Mutant Operators for the C Programming Language. *Technical Report SERC-TR-41-P*, Software Engineering Research Center, Purdue University, Purdue, IN, USA 1989.
5. Offutt AJ, Untch RH. Mutation 2000: Uniting the Orthogonal. *Mutation Testing for the New Century*. Advances in Database Systems, Springer: Boston, MA, USA, 2001; 34–44, doi:10.1007/978-1-4757-5939-6_7.
6. Ma YS, Kwon YR, Offutt J. Inter-Class Mutation Operators for Java. *International Symposium on Software Reliability Engineering*. ISSRE, IEEE: Annapolis, MD, USA, 2002; 352–363, doi:10.1109/ISSRE.2002.1173287.
7. Lackner H, Schmidt M. Towards the Assessment of Software Product Line Tests: A Mutation System for Variable Systems. *International Systems and Software Product Line Conference*. SPLC, ACM: Florence, Italy, 2014; 62–69, doi:10.1145/2647908.2655968.
8. Al-Hajjaji M, Benduhn F, Thüm T, Leich T, Saake G. Mutation Operators for Preprocessor-Based Variability. *International Workshop on Variability Modelling of Software-Intensive Systems*. VaMoS, ACM: Salvador, Brazil, 2016; 81–88, doi:10.1145/2866614.2866626.
9. Al-Hajjaji M, Krüger J, Benduhn F, Leich T, Saake G. Efficient Mutation Testing in Configurable Systems. *International Workshop on Variability and Complexity in Software Design*. VACE, IEEE: Buenos Aires, Argentina, 2017; 2–8, doi:10.1109/VACE.2017.3.
10. Clements P, Northrop L. *Software Product Lines*. Addison-Wesley: Boston, MA, USA, 2002.
11. Pohl K, Böckle G, van der Linden FJ. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer: Berlin Heidelberg, Germany, 2005.
12. Apel S, Batory D, Kästner C, Saake G. *Feature-Oriented Software Product Lines*. Springer: Berlin Heidelberg, Germany, 2013, doi:10.1007/978-3-642-37521-7.
13. Just R, Kapfhammer GM, Schweiggert F. Do Redundant Mutants Affect the Effectiveness and Efficiency of Mutation Analysis? *International Conference on Software Testing, Verification and Validation*. ICST, IEEE: Montreal, QC, Canada, 2012; 720–725, doi:10.1109/ICST.2012.162.

14. Madeyski L, Orzeszyna W, Torkar R, Jozala M. Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation. *IEEE Transactions on Software Engineering* 2014; **40**(1):23–42, doi:10.1109/TSE.2013.44.
15. Carvalho L, Guimarães MA, Ribeiro M, Fernandes L, Al-Hajjaji M, Gheyi R, Thüm T. Equivalent Mutants in Configurable Systems: An Empirical Study. *International Workshop on Variability Modelling of Software-Intensive Systems*. VaMoS, ACM: Madrid, Spain, 2018; 11–18, doi:10.1145/3168365.3168379.
16. Papadakis M, Jia Y, Harman M, Le Traon Y. Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique. *International Conference on Software Engineering*. ICSE, IEEE: Florence, Italy, 2015; 936–946, doi:10.1109/ICSE.2015.103.
17. Mathur AP, Wong WE. An Empirical Comparison of Data Flow and Mutation-Based Test Adequacy Criteria. *Software Testing, Verification and Reliability* 1994; **4**(1):9–31, doi:10.1002/stvr.4370040104.
18. Siami Namin A, Andrews JH, Murdoch DJ. Sufficient Mutation Operators for Measuring Test Effectiveness. *International Conference on Software Engineering*. ICSE, ACM: Leipzig, Germany, 2008; 351–360, doi:10.1145/1368088.1368136.
19. Devroey X, Perrouin G, Papadakis M, Legay A, Schobbens PY, Heymans P. Featured Model-Based Mutation Analysis. *International Conference on Software Engineering*. ICSE, ACM: Austin, TX, USA, 2016; 655–666, doi:10.1145/2884781.2884821.
20. Prehofer C. Feature-Oriented Programming: A Fresh Look at Objects. *European Conference on Object-Oriented Programming*. ECOOP, Springer: Uppsala, Sweden, 1997; 419–443, doi:10.1007/BFb0053389.
21. Gacek C, Anastasopoules M. Implementing Product Line Variabilities. *ACM SIGSOFT Software Engineering Notes* 2001; **26**(3):109–117, doi:10.1145/379377.375269.
22. Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes C, Loingtier JM, Irwin J. Aspect-Oriented Programming. *European Conference on Object-Oriented Programming*. ECOOP, Springer: Uppsala, Sweden, 1997; 220–242, doi:10.1007/BFb0053381.
23. Apel S, Kästner C, Lengauer C. Language-Independent and Automated Software Composition: The FeatureHouse Experience. *IEEE Transactions on Software Engineering* 2013; **39**(1):63–79, doi:10.1109/TSE.2011.120.
24. Schobbens PY, Heymans P, Trigaux JC. Feature Diagrams: A Survey and a Formal Semantics. *International Conference Requirements Engineering*. RE, IEEE: Minneapolis/St. Paul, MN, USA, 2006; 139–148, doi:10.1109/RE.2006.23.
25. Chen L, Babar MA. A Systematic Review of Evaluation of Variability Management Approaches in Software Product Lines. *Information and Software Technology* 2011; **53**(4):344–362, doi:10.1016/j.infsof.2010.12.006.
26. Czarnecki K, Grünbacher P, Rabiser R, Schmid K, Wąsowski A. Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. *International Workshop on Variability Modelling of Software-Intensive Systems*. VaMoS, ACM: Leipzig, Germany, 2012; 173–182, doi:10.1145/2110147.2110167.
27. Lee K, Kang KC, Lee J. Concepts and Guidelines of Feature Modeling for Product Line Software Engineering. *International Conference on Software Reuse*. ICSR, Springer: Austin, TX, USA, 2002; 62–77, doi:10.1007/3-540-46020-9_5.
28. Berger T, Rublack R, Nair D, Atlee JM, Becker M, Czarnecki K, Wąsowski A. A Survey of Variability Modeling in Industrial Practice. *International Workshop on Variability Modelling of Software-Intensive Systems*. VaMoS, ACM: Pisa, Italy, 2013; 7:1–7:8, doi:10.1145/2430502.2430513.
29. Kästner C, Apel S. Integrating Compositional and Annotative Approaches for Product Line Engineering. *Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering*. McGPLE, University of Passau: Nashville, TN, USA, 2008; 35–40.
30. Krüger J, Schröter I, Kenner A, Kruczek C, Leich T. FeatureCoPP: Compositional Annotations. *International Workshop on Feature-Oriented Software Development*. FOSD, ACM: Amsterdam, The Netherlands, 2016; 74–84, doi:10.1145/3001867.3001876.
31. Thüm T. Product-Line Specification and Verification with Feature-Oriented Contracts. PhD Thesis, Otto-von-Guericke University, Magdeburg, Germany 2015.
32. Apel S, Kästner C, Lengauer C. FeatureHouse: Language-Independent, Automated Software Composition. *International Conference on Software Engineering*. ICSE, IEEE: Vancouver, BC, Canada, 2009; 221–231, doi:10.1109/ICSE.2009.5070523.
33. Kästner C, Apel S, Kuhlemann M. A Model of Refactoring Physically and Virtually Separated Features. *International Conference on Generative Programming: Concepts and Experiences*. GPCE, ACM: Denver, CO, USA, 2009; 157–166, doi:10.1145/1621607.1621632.
34. Kuhlemann M. Refactoring Feature Modules: Disciplined Generation of Reusable Modules. PhD Thesis, Otto-von-Guericke University, Magdeburg, Germany 2011.
35. Kernighan BW, Ritchie DM. *The C Programming Language*. Prentice Hall: Upper Saddle River, NJ, USA, 1988.
36. Medeiros F, Kästner C, Ribeiro M, Nadi S, Gheyi R. The Love/Hate Relationship with the C Preprocessor: An Interview Study. *European Conference on Object-Oriented Programming*. ECOOP, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik: Wadern, Germany, 2015; 495–518, doi:10.4230/LIPIcs.ECOOP.2015.495.
37. Hunsen C, Zhang B, Siegmund J, Kästner C, Leßenich O, Becker M, Apel S. Preprocessor-Based Variability in Open-Source and Industrial Software Systems: An Empirical Study. *Empirical Software Engineering* 2016; **21**(2):449–482, doi:10.1007/s10664-015-9360-1.
38. Chevalley P. Applying Mutation Analysis for Object-Oriented Programs Using a Reflective Approach. *Asia-Pacific Software Engineering Conference*. APSEC, IEEE: Macao, China, 2001; 267–270, doi:10.1109/APSEC.2001.991487.
39. Anbalagan P, Xie T. Automated Generation of Pointcut Mutants for Testing Pointcuts in AspectJ Programs. *International Symposium on Software Reliability Engineering*. ISSRE, IEEE: Seattle, WA, USA, 2008; 239–248, doi:10.1109/ISSRE.2008.58.
40. Fraser G, Wotawa F. Mutant Minimization for Model-Checker Based Test-Case Generation. *Testing: Academic and Industrial Conference Practice and Research Techniques*. TAICPART, IEEE: Windsor, United Kingdom, 2007; 161–168, doi:10.1109/TAIC.PART.2007.30.

41. Batth SS, Vieira ER, Cavalli A, Uyar MÜ. Specification of Timed EFSM Fault Models in SDL. *International Conference on Formal Techniques for Networked and Distributed Systems*. FORTE, Springer: Tallinn, Estonia, 2007; 50–65, doi:10.1007/978-3-540-73196-2_4.
42. Gopinath R, Alipour MA, Ahmed I, Jensen C, Groce A. On the Limits of Mutation Reduction Strategies. *International Conference on Software Engineering*. ICSE, IEEE: Austin, TX, USA, 2016; 511–522, doi:10.1145/2884781.2884787.
43. Engström E, Runeson P. Software Product Line Testing – a Systematic Mapping Study. *Information and Software Technology* 2011; **53**(1):2–13, doi:10.1016/j.infsof.2010.05.011.
44. Henard C, Papadakis M, Le Traon Y. Mutation-Based Generation of Software Product Line Test Configurations. *International Symposium on Search Based Software Engineering*. SSBSE, Springer: Paderborn, Germany, 2014; 92–106, doi:10.1007/978-3-319-09940-8_7.
45. Arcaini P, Gargantini A, Vavassori P. Generating Tests for Detecting Faults in Feature Models. *International Conference on Software Testing, Verification and Validation*. ICST, IEEE: Graz, Austria, 2015; 1–10, doi: 10.1109/ICST.2015.7102591.
46. Reuling D, Bürdek J, Rotärmel S, Lochau M, Kelter U. Fault-Based Product-Line Testing: Effective Sample Generation Based on Feature-Diagram Mutation. *International Systems and Software Product Line Conference*. SPLC, ACM: Nashville, TN, USA, 2015; 131–140, doi:10.1145/2791060.2791074.
47. Koppen C, Störzer M. PCDiff: Attacking the Fragile Pointcut Problem. *European Interactive Workshop on Aspects in Software*. EIWAS, 2004.
48. Störzer M, Graf J. Using Pointcut Delta Analysis to Support Evolution of Aspect-Oriented Software. *International Conference on Software Maintenance*. ICSM, IEEE: Budapest, Hungary, 2005; 653–656, doi:10.1109/ICSM.2005.99.
49. Garvin BJ, Cohen MB. Feature Interaction Faults Revisited: An Exploratory Study. *International Symposium on Software Reliability Engineering*. ISSRE, IEEE: Hiroshima, Japan, 2011; 90–99, doi:10.1109/ISSRE.2011.25.
50. Rodrigues I, Ribeiro M, Medeiros F, Borba P, Fonseca B, Gheyi R. Assessing Fine-Grained Feature Dependencies. *Information and Software Technology* 2016; **78**:27–52, doi:10.1016/j.infsof.2016.05.006.
51. Siegmund J, Kästner C, Liebig J, Apel S. Comparing Program Comprehension of Physically and Virtually Separated Concerns. *International Workshop on Feature-Oriented Software Development*. FOSD, ACM: Dresden, Germany, 2012; 17–24, doi:10.1145/2377816.2377819.
52. Krüger J. Separation of Concerns: Experiences of the Crowd. *ACM Symposium on Applied Computing*. SAC, ACM: Pau, France, 2018; 2062–2063, doi:10.1145/3167132.3167458.
53. Anbalagan P, Xie T. Efficient Mutant Generation for Mutation Testing of Pointcuts in Aspect-Oriented Programs. *Workshop on Mutation Analysis*. IEEE: Raleigh, NC, USA, 2006; 13–18, doi:10.1109/MUTATION.2006.3.
54. Ferrari FC, Maldonado JC, Rashid A. Mutation Testing for Aspect-Oriented Programs. *International Conference on Software Testing, Verification and Validation*. ICST, IEEE: Lillehammer, Norway, 2008; 52–61, doi:10.1109/ICST.2008.37.
55. Tevanlinna A, Taina J, Kauppinen R. Product Family Testing: A Survey. *ACM SIGSOFT Software Engineering Notes* 2004; **29**(2):12–12, doi:10.1145/979743.979766.
56. Thüm T, Kästner C, Benduhn F, Meinicke J, Saake G, Leich T. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming* 2014; **79**:70–85, doi:10.1016/j.scico.2012.06.002.
57. Sedgewick R. *Algorithms*. Pearson: London, United Kingdom, 1988.
58. Lopez-Herrejon R, Batory D. A Standard Problem for Evaluating Product-Line Methodologies. *International Conference on Generative and Component-Based Software Engineering*. GCSE, Springer: Erfurt, Germany, 2001; 10–24.
59. Pacheco C, Lahiri SK, Ernst MD, Ball T. Feedback-Directed Random Test Generation. *International Conference on Software Engineering*. ICSE, IEEE: Minneapolis, MN, USA, 2007; 75–84, doi:10.1109/ICSE.2007.37.
60. Laurent T, Papadakis M, Kintis M, Henard C, Le Traon Y, Ventresque A. Assessing and Improving the Mutation Testing Practice of PIT. *International Conference on Software Testing, Verification and Validation*. ICST, IEEE: Tokyo, Japan, 2017; 430–435, doi:10.1109/ICST.2017.47.
61. Fraser G, Arcuri A. EvoSuite: Automatic Test Suite Generation for Object-oriented Software. *ACM SIGSOFT Symposium and European Conference on Foundations of Software Engineering*. ESEC/FSE, ACM: Szeged, Hungary, 2011; 416–419, doi:10.1145/2025113.2025179.
62. Rueda U, Just R, Galeotti JP, Vos TE. Unit Testing Tool Competition—Round Four. *International Workshop on Search-Based Software Testing*. SBST, IEEE: Austin, TX, USA, 2016; 19–28.
63. Fraser G, Rojas JM, Campos J, Arcuri A. EvoSuite at the SBST 2017 Tool Competition. *International Workshop on Search-Based Software Testing*. SBST, IEEE: Buenos Aires, Argentina, 2017; 39–42, doi:10.1109/SBST.2017.6.
64. Inozemtseva L, Holmes R. Coverage is Not Strongly Correlated with Test Suite Effectiveness. *International Conference on Software Engineering*. ICSE, ACM: Hyderabad, India, 2014; 435–445, doi:10.1145/2568225.2568271.
65. Coles H, Laurent T, Henard C, Papadakis M, Ventresque A. PIT: A Practical Mutation Testing Tool for Java (Demo). *International Symposium on Software Testing and Analysis*. ISSTA, ACM: Saarbrücken, Germany, 2016; 449–452, doi:10.1145/2931037.2948707.
66. Papadakis M, Henard C, Harman M, Jia Y, Le Traon Y. Threats to the Validity of Mutation-Based Test Assessment. *International Symposium on Software Testing and Analysis*. ISSTA, ACM: Saarbrücken, Germany, 2016; 354–365, doi:10.1145/2931037.2931040.
67. Offutt AJ, Pan J. Automatically Detecting Equivalent Mutants and Infeasible Paths. *Software Testing, Verification and Reliability* 1997; **7**(3):165–192, doi:10.1002/(SICI)1099-1689(199709)7:3<165::AID-STVR143>3.0.CO;2-U.
68. Nie C, Leung H. A Survey of Combinatorial Testing. *ACM Computing Surveys* 2011; **43**(2):11:1–11:29, doi: 10.1145/1883612.1883618.
69. Roy CK, Cordy JR, Koschke R. Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. *Science of Computer Programming* 2009; **74**(7):470–495, doi:10.1016/j.scico.2009.02.007.
70. Feng Y, Liu X, Kerridge J. A Product Line Based Aspect-Oriented Generative Unit Testing Approach to Building Quality Components. *International Computer Software and Applications Conference*. COMPSAC, IEEE: Beijing,

China, 2007; 403–408, doi:10.1109/COMPSAC.2007.35.

71. Devroey X, Perrouin G, Cordy M, Papadakis M, Legay A, Schobbens PY. A Variability Perspective of Mutation Analysis. *International Symposium on Foundations of Software Engineering*. FSE, ACM: Hong Kong, China, 2014; 841–844, doi:10.1145/2635868.2666610.
72. Budd TA. Mutation Analysis of Program Test Data. PhD Thesis, Yale University, New Haven, CT, USA 1980.
73. Zhang L, Gligoric M, Marinov D, Khurshid S. Operator-Based and Random Mutant Selection: Better Together. *International Conference on Automated Software Engineering*. ASE, IEEE: Silicon Valley, CA, USA, 2013; 92–102, doi:10.1109/ASE.2013.6693070.
74. Wong WE, Mathur AP. Reducing the Cost of Mutation Testing: An Empirical Study. *Journal of Systems and Software* 1995; **31**(3):185–196, doi:10.1016/0164-1212(94)00098-0.
75. Offutt AJ, Rothermel G, Zapf C. An Experimental Evaluation of Selective Mutation. *International Conference on Software Engineering*. ICSE, IEEE: Baltimore, MD, USA, 1993; 100–107, doi:10.1109/ICSE.1993.346062.
76. Hussain S. Mutation Clustering. Master's Thesis, King's College, London, United Kingdom 2008.
77. Derezińska A. Toward Generalization of Mutant Clustering Results in Mutation Testing. *Soft Computing in Computer and Information Science*. SCCIS, Springer: Cham, Switzerland, 2015; 395–407, doi:10.1007/978-3-319-15147-2_33.