Mutation Operators for Preprocessor-Based Variability

Mustafa Al-Hajjaji
University of Magdeburg
Germany
mustafa.al-hajjaji@ovgu.de

Fabian Benduhn
University of Magdeburg
Germany
fabian.benduhn@ovgu.de

Thomas Thüm
TU Braunschweig
Germany
t.thuem@tu-braunschweig.de

Thomas Leich
METOP GmbH
Germany
thomas.leich@metop.de

Gunter Saake
University of Magdeburg
Germany
gunter.saake@ovgu.de

ABSTRACT

Mutation testing has emerged as one of the most promising techniques to increase the quality of software-intensive systems. In mutation testing, random faults based on a predefined set of mutation operators are automatically injected into a program to evaluate test suites. The effectiveness of mutation testing strongly depends on the representativeness of the mutation operators. Existing operators are not sufficient to represent typical faults caused by variability. Thus, we propose a set of mutation operators for software with preprocessor-based variability. We derive the operators systematically based on a taxonomy of variability-related faults and evaluate them by investigating their applicability to real-world faults that have previously been identified in research on configurable software systems. Our goal is to leverage mutation testing to highly-variable software for its practical application and to enable empirical evaluation of testing techniques.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*mutation testing, mutation operators, variability*; D.2.2 [**Software Engineering**]: Design Tools and Techniques

Keywords

Variability; Mutation Testing; Mutation Operators; Preprocessor-Based Variability

1. INTRODUCTION

Mutation testing is a fault-based testing technique that has been proposed to measure the effectiveness of test suites. A mutation operator defines a modification to be made in the source code. The resulting programs are called mutants [14]. Test cases are used to execute these mutants with the goal to generate an incorrect output, i.e., to identify the mutants. The test suite is effective if it distinguishes between the program and its mutants. Mutation testing is a mature technique that has often shown its value for evaluating software testing techniques and test cases [14].

The quality of mutation testing largely depends on the used mutation operators. Ideally, mutation operators represent realistic faults, i.e., they mimic typical faults that programmers usually make. Thus, typical mutation operators may replace logical operators, remove statements, or change the value of an expression.

Recently, researchers have proposed to apply mutation testing for configurable software [5, 13, 20, 19, 21, 26], i.e., software for which many different variants can be generated [4]. For such systems, conventional mutation operators do not suffice. In principal, it is possible to use conventional mutation operators directly on the generated software variants. However, the mutants would not represent variability-specific faults. While operators for feature models have already been proposed [5, 13, 19, 21], they do not cover all relevant types of variability-related faults (e.g., faults in the source code). Furthermore, in previous work [26], we have used conventional mutation operators on variable software to evaluate analysis techniques. As we only used simple mutation operators, the resulting mutants may not represent real variability-related faults and therefore, we identified the need for variability-specific operators.

In this paper, we propose a set of mutation operators for configurable software systems. We focus on preprocessor-based variability because it is widely used to implement configurable software systems. In this preliminary work, we aim to include operators that cover all relevant parts of configurable software by considering variability model, development artifacts, and the mapping between both. In particular, we make the following contributions:

- We systematically derive a set of mutation operators for software with preprocessor-based compile-time variability from a taxonomy of variability-related faults.
- We evaluate to which extent the operators represent realistic faults by mapping them to real faults from a variability faults database[1] of two open-source systems, Linux kernel and Busybox.

In Section 2, we briefly introduce the necessary concepts used in this paper. Variability-related faults are represented in Section 3. In Section 4, we present mutation operators for preprocessor-based

[1]http://vbdb.itu.dk/

VaMoS '16, January 27-29, 2016, Salvador, Brazil
© 2016 ACM. ISBN 978-1-4503-4019-9/16/01...$15.00
DOI: http://dx.doi.org/10.1145/2866614.2866626

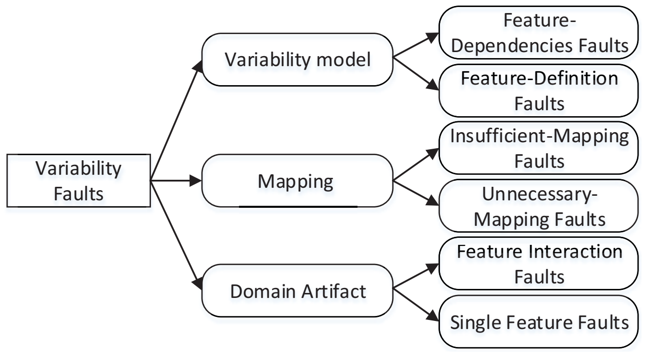
Variability Faults
Variability model
Mapping
Domain Artifact
Feature-Dependencies Faults
Feature-Definition Faults
Insufficient-Mapping Faults
Unnecessary-Mapping Faults
Feature Interaction Faults
Single Feature Faults

Figure 1: Taxonomy of Variability-Based Faults

variability and evaluate them in Section 5. Related work is discussed in Section 6 and we conclude our work in Section 7.

2. BACKGROUND

Mutation testing is a fault-based testing approach with the goal to assess the effectiveness of test cases to improve the quality of software [14]. In mutation testing, mutants are injected automatically into programs by changing the source code according to predefined mutation operators, typically representing simple syntactical modifications. The underlying assumption is that such mutants represent realistic bugs sufficiently. When a mutated program is tested, there are two possible outcomes. First, one of the test cases fails, i.e., it detects the fault – this is also called killing the mutant. Second, all test cases run successfully. Then it is called a live mutant.

Mutation testing has shown its value as one of the most promising techniques to assess the effectiveness of test cases and has been applied to many programming languages and formal notations [14]. It has been applied to numerous programming languages, such as C [22], Java [8], and AspectJ [3]. Moreover, it has been used to test the specifications or models of software at the design level, such as state charts [11], finite state machines [6], and feature models [5, 13, 19, 21]. However, the quality of mutation testing depends mainly on the quality of the used mutation operators, i.e, to which extent they correlate with faults.

In this paper, we consider the application of mutation testing to configurable software systems, in which variability is made explicit in terms of features that represent characteristics of the software [4]. Such systems contain a variability model (e.g, feature model [15] or a Kconfig model) that defines a set of features and their valid combinations. Each valid combination, called configuration, can be used to generate a desired variant.

Several techniques can be used to implement configurable systems [4]. In this paper, we focus on the use of variability implemented with preprocessors. Using this approach, code is annotated with preprocessor statements to achieve a mapping to certain configurations in terms of features. Depending on the configuration, unnecessary code will be removed by the preprocessor during compilation.

3. VARIABILITY-BASED FAULTS

In this paper, we propose a set of mutation operators that represent potential varaibility faults. Thus, we start by discussing the nature of these faults and present a taxonomy of variability-based faults that we have derived from the literature [5, 12, 1, 23, 7].

As illustrated in *Figure* 1, we distinguish between faults in three layers: Faults in the variability model, faults in the domain artifacts (e.g., implementation), and faults in the mapping between both. This distinction is commonly used in research on variable systems in general [7] and has also been used to categorize variability faults [1]. We focus on these faults, because we can assume that faults in the generated products can be traced back to the aforementioned layers. In this section, we discuss each fault independent of the implementation technique and exemplify the faults examples for each category from a variability faults database maintained by Abal et al. [1].

Fault 1 Simplified fault for undefined symbol reference[2]

```
1: #if defined(CONFIG_ARCH_OMAP2420)
2: void omap2_set_globals_242x(void)
3: {
4:       return;
5: }
6: #endif
7: #ifdef CONFIG_MACH_OMAP_H4
8: static void omap_h4_map_io(void)
9: {
10:       omap2_set_globals_242x();     // ERROR
11: }
12: #endif
13: int main(int argc, char** argv)
14: {
15: #ifdef CONFIG_MACH_OMAP_H4
16:       omap_h4_map_io();
17: #endif
18: }
```

3.1 Variability-Model Faults

First, we consider faults in the variability model, which it defines a set of features and their dependencies. Arcaini et al. propose fault categories for feature models that include feature-based and constraint-based faults [5]. We generalize this notion to arbitrary types of variability models and, thus, distinguish between two types of variability model faults: feature-definition faults and feature-dependency faults.

Feature-definition faults are related to the definition of the set of features, i.e., the set of features defined in the variability model differs from the intended set of features. For instance, some features could be missing in the variability model.

Feature-dependency faults are related to wrong dependencies between features. Even if a variability model includes the correct set of features, their dependencies can be faulty - resulting in an undesired set of valid products. Note, that in practice both types of faults often appear in combination. Feature-definition faults also lead to a wrong set of products and if there are multiple features involved, it can be hard to differentiate between both types of faults.

The example code in Fault 1 can lead to an error in Line 10 if executed. The reason is that function `omap2_set_globals_242x()` is called even if feature `ARCH_OMAP2420` is not defined. However, in this case the function will be undefined, leading to the aforementioned error. Thus, there is a dependency between feature `MACH_OMAP_H4` and feature `ARCH_OMAP2420` that is not correctly defined. Hence, we consider it as a variability model fault, in general, and as a feature dependency fault, in particular.

3.2 Variability-Mapping Faults

Faults can also appear in the mapping between features de-

[2]http://vbdb.itu.dk/#bug/linux/6515e48

Fault 2 Simplified bug for Undefined reference to macro when !BLK_CGROUP [3]

```
1: #ifdef CONFIG_BLK_CGROUP
2: #define CFQ_WEIGHT_DEFAULT 500
3: #endif
4: #ifdef CONFIG_IOSCHED_CFQ
5: static long cfq_scale_slice()
6: {
7: long d = 1;
8: d = d * CFQ_WEIGHT_DEFAULT;      // ERROR
9: return d;
10: }
10: #endif
11: int main()
12: {
13: #ifdef CONFIG_IOSCHED_CFQ
14: cfq_scale_slice();
15: #endif
16: return 0;
18: }
```

fined in the variability model and domain artifacts. Depending on the implementation technique, this mapping can be established either explicitly by annotating code fragments, in terms of a separate mapping model [19], implicitly in composition-based techniques where dedicated modules, or plugins are mapped to features. For preprocessor-based variability, mapping faults are faults in the `ifdef` expressions used for conditional compilation or faults in a separate mapping model as used in some cases.

In a mapping fault, a certain code fragment is mapped to the wrong set of configurations. We distinguish two types of mapping faults: insufficient-mapping faults and unnecessary-mapping faults. In unnecessary-mapping faults, there is code mapped to some configurations in which it is not required, potentially causing errors. In insufficient-mapping faults, there is code missing in configurations in which it is required and may lead to errors. Note, that a possible mapping fault can be classified as a combination of these two types. In the general case, code can be mapped to some configurations that do not require it and may be missing in other configurations that require it.

As an example, Fault 2 shows a fault that can lead to an error in Line 8 when the code is executed. The macro `CFQ_WEIGHT_DEFAULT` in `blk-cgroup.h` is conditionally defined (Line 2) only when feature `BLK_CGROUP` is enabled. However, the macro is used also when it is not enabled, which leads to an compiler error in Line 8. Thus, Line 2 should not be mapped to feature `BLK_CGROUP`, which could be fixed by removing lines one and three. The result would be that the code is now mapped to all necessary configurations. As the fault is fixed in the mapping layer, we consider it as a mapping fault (missing code).

3.3 Domain Artifact Fault

Variability faults can also appear in the implementation of domain artifacts (e.g., in the source code). We adopt the notion of interaction faults from the literature [23, 12]. Feature interaction faults can be distinguished by their degree, i.e., the number of features involved in a certain interaction. We distinguish between two types, single feature faults and feature interaction faults. The degree of an interaction is the minimal partial configuration that exposes a fault, i.e., the fault is caused by the interaction of the involved features. Although faults in a single feature are not caused directly by variability, Garvin et al. [12] suggest that using interaction testing techniques, such as combinatorial interaction testing, can detect these faults easily, which also applies to the corresponding mutation operators that we propose in this paper.

Fault 3 Argument type incompatible with 'printk' format string [4]

```
1: #include <stdio.h>
2: #ifdef CONFIG_LBDAF
3: typedef unsigned long long sector_t;
4: #else
5: typedef unsigned long sector_t;
6: #endif
7: sector_t blk_rq_pos() {
8: return 0;
9: }
10: #ifdef CONFIG_AMIGA_Z2RAM
11: static void do_z2_request() {
12: printf("bad access: block=%lu\n", blk_rq_pos()); // ERROR
12: }
13: #endif
14: int main() {
15: #ifdef CONFIG_AMIGA_Z2RAM
16: do_z2_request();
18: #endif
16: return 0;
18: }
```

In Fault 3, the type `sector_t` is defined. Depending on whether feature `LBDAF` is enabled, it will represent a different type, unsigned long or unsigned long long. However, this difference is not considered in Line 12, where function `blk_rq_pos()` returns a value of type `sector_t`. As there is no safe cast from unsigned long long to unsigned long, this leads to an error for some configurations. Thus, there is an interaction between feature `AMIGA's ramdisk` and feature `LBDAF`. This fault can be fixed in the code layer by printing the value as unsigned long long in each case, because it can be safely cast in this direction. Thus, we consider it as a domain artifact fault (feature interaction fault).

4. MUTATION OPERATORS FOR PREPROCESSOR-BASED VARIABILITY

In this section, we propose mutation operators for configurable systems with preprocessor-based variablity. These mutation operators are designed to model realistic variability faults. Our proposal aims to fulfill two goals. First, the set of operators should cover most relevant fault types that can occur in practice. Thus, we propose a set of operators that is *complete* in relation to our taxonomy, i.e., whether all relevant fault types in our taxonomy are covered. Second, each mutation operator should reflect *realistic* faults related to variability, i.e, is each operator required to cover faults that actually occur in real-world systems. Based on our experience with preprocessor-based program families, we propose mutation operators for each category. Whether each of them actually represents realistic faults, is evaluated in Section 5. We present the set of operators in terms of the corresponding fault types. Accordingly, we distinguish between model, mapping, and domain artifact (code) operators.

[3] http://vbdb.itu.dk/#bug/linux/f48ec1d

[4] http://vbdb.itu.dk/#bug/linux/e1fbd92

4.1 Variability-Model Operators

In general, it is possible to use various types of variability models, such as feature models [15], to define the variability in preprocessor-based systems. Mutation operators for feature models based on the hierachical structure [5, 19, 21] and on their propositional semantics [13] have been proposed in the literature However, in practice, developers typically use language-specific tools, such as Kconfig, to define the variability. In this paper, we use Kconfig to illustrate some operators. According to our taxonomy, we distinguish between feature-definition faults and feature-dependency faults. For each fault type, we present operators that manipulate the variability model.

Feature Definition Operators.

Feature definition operators modify the set of defined features, e.g., by removing feature definitions. For preprocessors, we propose the following operator.

RFDM - Remove Feature from Model The RFDM operator deletes the definition of a feature from the model (in our case Kconfig model). An error may occur if part of the code needs the deleted feature to be defined. We do not recommend removing features that are involved in constraints as this would lead to an invalid feature model. In this case, the feature could be removed safely by renaming it only in the variability model while keeping the original name in the mapping and code, or by using special algorithms to remove features [25].

Original Code	**RFDM Mutant**
...	...
config featureA	
bool "Feature A"	
default y	
...	...

Feature Dependency Operators.

Feature dependency operators modify dependencies between features defined in the variability model. We consider two main options. First, feature dependencies can directly be defined in the variability model. In the case of Kconfig, this is done by using `depends` and `select` statements. Second, we can use `#define` statements which conditionally depend on other features to express additional dependencies. We propose the following operators. Although we do not recommend the latter option for various reasons, it may occur in real systems and should be target of mutation testing, too.

MFDM - Modify Feature Dependency in Model The MFDM operator modifies a feature dependency from the variability model. In the case of Kconfig, this can be done by deleting a feature from a `depends` and `select` statement. This may cause an error, because the dependency between features is missing and new feature interactions may arise in the new configurations.

Original Code	**MFDM Mutant**
...	...
config featureA	config featureA
bool "Feature A"	bool "Feature A"
depends on featureB && featureC	*depends on featureB*
...	...

RCFD - Remove Conditional Feature Definition The RCFD operator removes a feature definition that conditionally depends on another feature by removing a `#define` statement from within an `ifdef` block. This modification manipulates the dependencies between features, which may lead to invalid and erroneous variants.

Original Code	**RCFD Mutant**
...	...
#ifdef featureA	#ifdef featureA
#define featureB	
#endif	#endif
...	...

ACFD - Add Condition to Feature Definition - The idea behind the ACFD operator is to add an `ifdef` condition around an existing `define` statement. This change adds an additional feature dependency, which may cause an error, because it may violate the combination between features. For instance, in the following example, we show that featureA will not be defined unless featureB is defined. This change may cause an error; if other features depend on featureA.

Original Code	**ACFD Mutant**
...	...
	#ifdef featureB
#define featureA	#define featureA
	#endif
...	...

4.2 Variability-Mapping Operators

Feature mapping operators change the mapping between features and code. The aforementioned mapping faults can be caused by adding code to some configurations, removing code from some configurations, or by moving code from some configurations to others. We propose mutation operators to perform these changes by modifying `ifdef` expressions.

Insufficient-Mapping Operators.

Insufficient-mapping operators modify the mapping between code and configurations by removing code from some configurations. We propose the following operators that add and modify `ifdef` statements.

AICC - Adding ifdef Condition around Code The AICC operator adds an `ifdef` condition around a code fragment. Thus, the code will only be enabled if the condition is satisfied. As a result, the code is removed from certain configurations in which it is required. We could also add an `ifndef` condition. Then the result would be that the code is removed from all configurations that do not fulfill the condition.

Original Code	AICC Mutant
...	...
	#ifdef featureA
function(int var)	function(int var)
	#endif
...	...

AFIC - Adding Feature to ifdef Condition The AFIC operator manipulates an `ifdef` condition by inserting an additional logical dependency to the expression, i.e., we add a feature using the logical AND operator (&&). Hence, the statements inside an `if` block will not be enabled unless the resulting condition is satisfied. The result is that the code is not mapped to all configurations in which it is required.

Original Code	AFIC Mutant
...	...
#if defined(featureA)	#if defined(featureA) *&& defined(featureB)*
function(int var);	function(int var);
#endif	#endif
...	...

Using other logical operators would have a different effect on the resulting type of fault, e.g., using the logical OR would result in an unnecessary-mapping fault.

Unnecessary-Mapping Operators.

Unnecessary-mapping operators modify the mapping between code and configurations by adding code to some configurations. We propose the following operators that manipulate `ifdef` statements.

RIDC - Remove ifdef Condition The RIDC operator deletes an `ifdef` condition. This operator enables part of the code to be executed, even without fulfilling the original conditions which may cause an error. The result is that code is added to some configurations in which it is not required.

Original Code	**RIDC Mutant**
...	...
#ifdef featureA	
function(int var);	function(int var);
#endif	
...	...

RFIC - Removing Feature of ifdef Condition The RFIC operator removes the occurrence of a feature from an `ifdef` expression. This change may cause an error, because a code fragment is executed only if the new condition is satisfied. Thus, the code is now mapped to some additional configurations.

Original Code	RFIC Mutant
...	...
#if defined(featureA) *&& defined(featureB)*	#if defined(featureA)
function(int var);	function(int var);
#endif	#endif
...	...

Other Mapping Operators.

As already mentioned, some mapping faults can be seen as combinations of unnecessary-mapping faults and insufficient mapping faults. Besides the previously presented mapping operators, we propose two operators that move code from some configurations to others. The resulting effect of these operators can be seen as a combination of adding code to and removing code from configurations. The following operators achieve this by modifying the type of a given `ifdef` directive.

RIND - Replacing ifdef Directive with ifndef Directive The RIND operator changes an existing `ifdef` directive to an `ifndef` directive. As a result, the code that was originally mapped to certain configurations will now be compiled only if the original condition is not fulfilled. As a result, the mapping is modified and may lead to errors.

Original Code	**RIND Mutant**
...	...
#ifdef featureA	*#ifndef* featureA
function(int var);	function(int var);
#endif	#endif
...	...

RNID - Replacing ifndef Directive with ifdef Directive The RNID operator changes an `ifndef` directive to an `ifdef` directive. Thus, it can be seen as the reverse operation to the RIND operator. The difference is that the code is mapped to configurations that do not satisfy the condition.

Note, while all mapping faults can be seen as a combination of insufficient-mapping faults and unnecessary-mapping faults, not all possible mapping faults may be completely covered by only RIND and RNID operators.

4.3 Domain Artifact operators

Code operators modify the actual source code. Manipulating the code can cause either single feature faults or feature interaction faults. The following operators make changes in the code layer.

CACO - Conditionally Applying Conventional Operator The CACO operator can be seen as a way to adapt any conventional mutation operator and apply it in a variability-aware way. The idea is to modify the source code only at locations that have a certain presence condition, which is the conjunction of all nested `ifdef` conditions. When applying this operator, the number of involved features can be chosen based on the existing presence conditions in the source code. Thus, it can be used to simulate single feature faults (degree = 1) or feature interaction faults of a certain degree. In the following example, we decrement an integer inside a code fragment that conditionally depends on the interaction of two features.

Original Code	**CACO Mutant**
...	...
#if defined(featureA)&& defined(featureB)	#if defined(featureA)&& defined(featureB)
char array[5]	*char array[4]*
#endif	#endif
...	...

We do not only consider a single `ifdef` expression, but the presence condition related to the complete source code. For instance, nesting of multiple `ifdef` blocks needs to be considered. For this purpose, the existing concepts and tools that analyze presence conditions (e.g., TypeChef [16]) can be used to make the application of the CACO operator feasible.

RCIB - Removing Complete ifdef Block The RCIB operator deletes an entire `ifdef` block. We assume the presence of only actual source code without further `ifdef` directives inside the `ifdef` block. However, it could also be applied when there are additional directives, such as `define` statements, but the resulting fault would also be related to the variability model (feature dependency fault).

Original Code	**RCIB Mutant**
...	...
#ifdef featureA	
function(int var)	
#endif	
...	...

MCIB - Moving Code around ifdef Blocks The operator MCIB moves a certain code fragment around an `ifdef` block, i.e., from directly before to after the block or vice versa. In the following example, it moves a line of code from before the `ifdef` block to after it.

Original Code	**MCIB Mutant**
...	...
*int *var=Null;*	
#ifdef featureA	#ifdef featureA
...	...
#endif	#endif
	*int *var=Null;*
...	...

The MCIB operator will cause a potential fault only if the `ifdef` expression is satisfied because the order of code fragments will have actually changed. In contrast, if the condition is not satisfied, the effective order of code fragments does not change, i.e., because the `ifdef` block will not be considered anyway.

5. PRELIMINARY EVALUATION

In this section, we evaluate the proposed mutation operators by investigating whether they reflect realistic faults related to variability.

5.1 Evaluation Steps

We classified the real faults reported by [1] into groups based on the place where they occur. Then, we mapped proposed operators to these fault types, as can be seen in *Table* 1. This mapping has been performed manually based on the information about the faults available in the database, including the source code, a simplified patch, and a description for each fault. We observed that a fault type might be caused by more than one mutation operator. In addition, the faults reported in *Table* 1 and mapped to the proposed mutation operators cannot not be caused by other existing mutation operators. In this section, we focus mainly on the evaluation of the proposed mutation operators of preprocessor-based variability, because we argue that there is a lack of studies in preprocessor-based mutation testing. On the contrary, several studies have been presented to employ mutation testing on feature models for different purposes, such as finding faults in feature model [5] and generating products [13, 20, 21].

In our evaluation, we study whether these operators can cause faults that are similar to real ones. In addition, we study how often each mutation operator can cause faults in the aforementioned database. The variable faults in the variability faults database are the results of the investigation [1] of two product lines: Linux kernel and Busybox. Busybox is a tiny version of many UNIX common utilities that are combined into a single executable. Busybox provides a quite complete environment for any small or embedded system. In the following, we present and discuss our results.

5.2 Results and Discussion

We investigated all 61 variable faults from the database. We exclude seven of them, because they are caused by faults in more than one layer. They can potentially be simulated by a combination of more than one mutation operator. However, we have not considered this yet for simplicity purposes.

As shown in *Table* 1, 13% of the faults can be caused by model operators. Operator *RCFD* causes five of seven of the reported model faults. In addition, we found that the operator *RFDM* cannot cause any of the faults in the database. For the mapping operators,

Layer	Faults	Operator	Linux	Busybox
Model			6	1
	Feature definition	RFDM	0	0
	Feature dependencies	MFDM RCFD ACFD	0 5 1	1 0 0
Mapping			5	7
	Insufficient-mapping	AICC AFIC	0 0	1 1
	Unnecessary-mapping	RIDC RFIC	3 1	4 1
	Other mapping faults	RIND RNID	1 0	0 0
Domain artifact			29	6
	Feature interaction & Single Feature	CACO RCIB MCIB	23 4 2	3 2 1
All layers			40	14

Table 1: The number of faults from the systems Linux kernel and Busybox that could be caused by each mutation operator.

we observe that they can cause approximately 22% of the reported faults. More than half of these faults can be caused by operator *RIDC*, while the rest of the operators can cause the other half. Furthermore, we observe that the mapping operator *RNID* do not cause any of the reported faults. The major part of the reported faults can be caused using the code operators (65%). About 74% of all code faults can be caused by operator *CACO*.

From *Table* 1, we observe that 35% of the faults can be triggered by using mutation operators from the model and mapping layers. In details, we observe that one mutation operator in each layer outperforms the other operators. As we show in *Table* 1, operators *RCFD, RIDC*, and *CACO* can cause 70% of the all reported faults. In case of operators *RCFD* and *RIDC*, we have no reason to assume that this observation can be generalized, e.g., to give these operators a higher priority for their practical application.

Operator *CACO* can cause 48% of the reported faults. Accordingly, we consider this operator to be important for the practical application. The main challenge to use this operator effectively is to find locations in the code where more than one feature interact with each other, and then even conventional mutation operators can be used to make changes for the variability system under test. However, several analysis techniques have been proposed to address this challenge [16].

Regarding the operators *RFDM* and *RIND*, which do not cause any fault in the database, we are planning further evaluation by using other case studies.

The preliminary results show that the majority of the proposed operators actually represent real faults from the considered systems. We also observed that a large part of the existing faults are covered by relatively few mutation operators, especially in the code layer.

5.3 Threats to Validity

One potential threat that may affect the internal validity is that we manually performed the classification for each fault based on a description from the database [1]. Thus, we relied on our subjective understanding of the faults. To mitigate these threats, we have additionally studied a simplified patch to fix the faults, which helped us to identify the actual source of the fault. Furthermore, we considered the accompanying information for each fault in the aforementioned database to help us in the evaluation of the proposed operators.

When analyzing the applicability of the CACO operator, we assumed the existence of conventional operators that can be used to apply the actual mutation at certain locations. Thus, we cannot ensure whether the operator can actually be used to introduce the identified faults in practice, especially in cases in which new code is introduced. However, this threat is not specifically related to variability and could be alleviated by choosing appropriate conventional operators.

A potential threat to external validity is the limited number of considered systems and faults. We evaluated the operators by analyzing 54 variable faults in two systems from the literature. To the best of our knowledge, this is the most comprehensive database of variability-related faults. After submitting the paper, new faults have been added to the aforementioned database, which we plan to consider in our future work. Furthermore, in our evaluation, we only used the simplified versions of the variability faults from the database. We do not consider possible differences between the original and the simplified fault, which may violate the validity of our results. In future work, we plan to include further faults from other systems to evaluate the proposed operators.

6. RELATED WORK

We discuss related studies that have been presented with regard to variability faults as well as mutation testing for variable software.

6.1 Variability Faults

Garvin et al. [12] perform an exploratory study on hundreds of faults in two open source systems GCC and Firefox. The goal of their study is to investigate the nature of the feature interaction faults in order to mimic these faults. They report that an analysis of associated guards and branch conditions might be enough to mimic such interaction faults. Our CACO operator is based on a similar idea to cause feature interaction faults. In contrast to our work, they only consider a single type of faults (feature interaction faults) and do not propose concrete mutation operators.

A few other studies investigate the nature of faults for configurable systems. Sánchez et al. [23] indicate that there is a significant correlation between the number of faults and the non-functional attributes. The reported faults are triggered by the interaction of up-to four features. They simulate these interactions among features as faults to be used in their evaluation. They assume that if a product contains these feature interactions, the faults are detected. Similar to Sánchez et al. in [23], others [2, 24, 10] also simulate variability faults. However, these faults are not real faults. They are just combinations of features, which cannot be used to improve existing test suites. In this paper, we propose mutation operators that can be used to inject the system in order to generate real faults.

Abal et al. [1] perform a qualitative study on bugs collected from the Linux kernel repository. The main objective of their study is to understand the nature of variability faults. In particular, they investigate whether the variability faults are limited to a certain type of fault, certain features, or specific location of the source code. They report that the faults are not limited to specific features or specific locations. In this article, we rely on their study to evaluate our mutation operators.

Devine et al. [9] study the correlation between certain metrics and faults. They report that change metrics are more correlated to faults than static metrics. Similarly, Krishnan et al. [18] report that change metrics are a good predictor of fault-prone files over time. Krishnan et al. [17] investigate the relationship between the faults and the change of feature types for certain releases of Eclipse. They report that the number of faults decreases in the common features and in features that are rarely changed. In addition, they indicate that the number of faults does not change for the optional features and for the features that are often changed. The previous studies do not investigate the nature of faults, whether they are related to variability or not, as well as they do not propose mutation operators to mimic the gathered faults. However, the results of these studies could be used to improve mutation testing by identifying promising, error-prone locations for mutation.

6.2 Mutation Operators

Recently, mutation testing has been applied to configurable software systems. Arcaini et al. [5] propose a set of mutation operators for feature models in order to find faults in feature models. Henard et al. [13] introduce a search-based approach to generate a set of products with aims to detect mutants in feature models. They create an altered version of feature models, which contain a fault within its propositional formula. Papadakis et al. [20] employ the operators that are suggested by Henard et al. [13] and propose mutation analysis as an alternative of combinatorial interaction testing to select samples. However, these mutation operators are specific only for feature models. In this paper, we consider mutation operators for the variability model as well as operators for variability mapping and domain artifacts to cover more realistic faults.

Lackner et al. [19] propose mutation testing to assess the testing quality of configurable systems by measuring the capability to detect faults. Their approach comprises model-based mutation operators, but mutating feature models is not enough, as we observed in our results. Reuling et al. in [21] present a feature-diagram framework to generate an effective set of samples for fault-based configurable systems. They consider atomic and complex mutation operators. However, they consider only mutation operators in feature diagrams. In our work, we map the proposed operators to the reported faults. In addition, we present mutation operators from different layers, such as variability model, mapping, and code layer. In particular, the majority of real faults is not in the feature model.

7. CONCLUSION AND FUTURE WORK

The increasing interest in variable software systems in the academia and industry requires different types of testing techniques to improve the quality of variable software systems. Mutation testing is an approach used to evaluate such testing techniques. Existing mutation operators are not sufficient to represent variability-related faults. Thus, we proposed mutation operators for the variability model, variability mapping, and domain artifact for preprocessor-based variability. Furthermore, we evaluate the operators by investigating whether they can cause realistic variability faults. We analyzed reported faults from previous studies. Our preliminary results show, that the proposed set of mutation operators can represent real faults and cover all three layers of our fault taxonomy. However, more research is necessary for further evaluation of the proposed operators.

In future work, we are planning to evaluate the set of operators in additional case studies. Furthermore, we plan to identify representative distributions of fault types in different domains, implementation techniques, programming languages. That is, we want to consider not only which types of fault may occur but also how likely a certain type of fault is. We hope to utilize such knowledge to build tools for practical application.

8. ACKNOWLEDGMENTS

This work has been partially funded by the German Federal Ministry of Education and Research (01IS14017A, 01IS14017B). We thank reviewers for their constructive feedback.

9. REFERENCES

[1] I. Abal, C. Brabrand, and A. Wasowski. 42 Variability Bugs in the Linux Kernel: A Qualitative Analysis. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 421–432. ACM, 2014.

[2] M. Al-Hajjaji, T. Thüm, J. Meinicke, M. Lochau, and G. Saake. Similarity-Based Prioritization in Software Product-Line Testing. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 197–206. ACM, 2014.

[3] P. Anbalagan and T. Xie. Automated Generation of Pointcut Mutants for Testing Pointcuts in AspectJ Programs. In *Proc. Int'l Symposium Software Reliability Engineering (ISSRE)*, pages 239–248, 2008.

[4] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, 2013.

[5] P. Arcaini, A. Gargantini, and P. Vavassori. Generating Tests for Detecting Faults in Feature Models. In *Proc. Int'l Conf. Software Testing, Verification and Validation (ICST)*, pages 1–10, 2015.

[6] S. S. Batth, E. R. Vieira, A. Cavalli, and M. U. Uyar. Specification of Timed EFSM Fault Models in SDL. In *Proc. IFIP Int'l Conf. Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 50–65. Springer, 2007.

[7] P. Borba, L. Teixeira, and R. Gheyi. A Theory of Software Product Line Refinement. *Theoretical Computer Science*, 455(0):2–30, 2012.

[8] P. Chevalley. Applying mutation analysis for object-oriented programs using a reflective approach. In *Proc. Asia-Pacific Software Engineering Conference (APSEC)*, pages 267–270, 2001.

[9] T. R. Devine, K. Goseva-Popstajanova, S. Krishnan, R. R. Lutz, and J. J. Li. An Empirical Study of Pre-Release Software Faults in an Industrial Product Line. In *Proc. Int'l Conf. Software Testing, Verification and Validation (ICST)*, pages 181–190, 2012.

[10] F. Ensan, E. Bagheri, and D. Gasevic. Evolutionary Search-Based Test Generation for Software Product Line Feature Models. In *Proc. Int'l Conf. Advanced Information Systems Engineering (CAiSE)*, volume 7328, pages 613–628. Springer, 2012.

[11] G. Fraser and F. Wotawa. Mutant Minimization for Model-Checker Based Test-Case Generation. In *in Proc. of Workshop on Mutation Analysis, Published with Proc. of Testing: Academic and Industrial Conference Practice and Research Techniques*, pages 161–168, 2007.

[12] B. Garvin and M. Cohen. Feature Interaction Faults Revisited: An Exploratory Study. In *Proc. Int'l Symposium Software Reliability Engineering (ISSRE)*, pages 90–99, 2011.

[13] C. Henard, M. Papadakis, and Y. Le Traon. Mutation-Based Generation of Software Product Line Test Configurations. In C. Le Goues and S. Yoo, editors, *Search-Based Software Engineering*, volume 8636 of *Lecture Notes in Computer Science*, pages 92–106. Springer International Publishing, 2014.

[14] Y. Jia and M. Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Trans. Software Engineering (TSE)*, 37(5):649–678, 2011.

[15] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, 1990.

[16] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *Proc. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 805–824. ACM, 2011.

[17] S. Krishnan, R. R. Lutz, and K. Goševa-Popstojanova. Empirical Evaluation of Reliability Improvement in an Evolving Software Product Line. In *Proc. of Working Conf. on Mining Software Repositories (MSR)*, pages 103–112. ACM, 2011.

[18] S. Krishnan, C. Strasburg, R. R. Lutz, and K. Goševa-Popstojanova. Are Change Metrics Good Predictors for an Evolving Software Product Line? In *Proc. of Int'l Conf. on Predictive Models in Software Engineering (Promise)*, pages 1–10. ACM, 2011.

[19] H. Lackner and M. Schmidt. Towards the Assessment of Software Product Line Tests: A Mutation System for Variable Systems. In *Proc. Workshop Software Product Line Analysis Tools (SPLat)*, pages 62–69. ACM, 2014.

[20] M. Papadakis, C. Henard, and Y. Le Traon. Sampling Program Inputs with Mutation Analysis: Going Beyond Combinatorial Interaction Testing. In *Proc. Int'l Conf. Software Testing, Verification and Validation (ICST)*, pages 1–10, 2014.

[21] D. Reuling, J. Bürdek, S. Rotärmel, M. Lochau, and U. Kelter. Fault-Based Product-Line Testing: Effective Sample Generation Based on Feature-Diagram Mutation. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 131–140. ACM, 2015.

[22] H. A. Richard, R. A. Demillo, B. Hathaway, W. Hsu, W. Hsu, E. Krauser, R. Martin, A. P. Mathur, and E. H. Spafford. Design Of Mutant Operators For The C Programming Language. Technical Report SERC-TR-41-P, Purdue University, West Lafayette, Indiana, 1989.

[23] A. Sánchez, S. Segura, J. Parejo, and A. Ruiz-Cortés. Variability Testing in the Wild: The Drupal Case Study. *Software and System Modeling*, pages 1–22, 2015.

[24] A. B. Sánchez, S. Segura, and A. Ruiz-Cortés. A Comparison of Test Case Prioritization Criteria for Software Product Lines. In *Proc. Int'l Conf. Software Testing, Verification and Validation (ICST)*, pages 41–50. IEEE, 2014.

[25] T. Thüm, C. Kästner, S. Erdweg, and N. Siegmund. Abstract Features in Feature Modeling. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 191–200. IEEE, 2011.

[26] T. Thüm, J. Meinicke, F. Benduhn, M. Hentschel, A. von Rhein, and G. Saake. Potential Synergies of Theorem Proving and Model Checking for Software Product Lines. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 177–186. ACM, 2014.