Efficient Mutation Testing in Configurable Systems

Mustafa Al-Hajjaji*, Jacob Krüger*†, Fabian Benduhn*, Thomas Leich† and Gunter Saake*
*University of Magdeburg, Germany
Email: mustafa.alhajjaji, jacob.krueger, fabian.benduhn, gunter.saake@ovgu.de
†Harz University of Applied Sciences, Germany
Email: tleich@hs-harz.de

***Abstract*—Mutation testing is a technique to evaluate the quality of test cases by assessing their ability to detect faults. Mutants are modified versions of the original program that are generated automatically and should contain faults similar to those caused by developers' mistakes. For configurable systems, existing approaches propose mutation operators to produce faults that may only exist in some configurations. However, due to the number of possible configurations, generating and testing all mutants for each program is not feasible. To tackle this problem, we discuss to use static analysis and adopt the idea of T-wise testing to limit the number of mutants. In particular, we *i)* discuss dependencies that exist in configurable systems, *ii)* how we can use them to identify code to mutate, and *iii)* assess the expected outcome. Our preliminary results show that variability analysis can help to reduce the number of mutants and, thus, costs for testing.**

I. Introduction

Mutation testing is a promising technique to evaluate the effectiveness of test cases [16, 17]. A mutant is generated automatically as a modified, possibly faulty, version of the original program by applying *mutation operators* to the source code. Specific mutation operators are proposed for different programming techniques and languages to exploit their properties [28]. The faults in the resulting mutants can be seen as simulating typical faults of software developers. Test cases are executed on these mutants to assess whether they are able identify the injected faults.

Traditionally, mutation testing has been considered as an approach that, while potentially providing a large benefit, is rather expensive [16, 21]. The costs result from executing a large number of mutants against test cases. Several cost reduction techniques have been proposed to tackle this problem, mainly utilizing two ideas. Firstly, as it is not feasible to generate mutants for all possible faults, different approaches try to minimize the number of mutants without significantly loosing effectiveness [7, 29, 39]. Secondly, some techniques aim to optimize the execution process itself [40].

In our research, we focus on highly-configurable systems in which software variability is one of the major challenges. Configurable systems enable companies (or even customers) to configure software systems based on a set of *features*, which are defined as an increment in functionality recognized by customers [4, 6]. The goal of this line of research is to investigate the application of mutation testing to highly-configurable systems. This is more expensive than for a single program due to the huge number of possible products caused by variability. In other words, in the context of configurable system, for each mutant, test cases need to be executed against several products, because the introduced faults may not appear in all configurations.

In previous work, we proposed an initial set of mutation operators suitable to produce mutants that mimic faults caused by variability, and as such representative for highly-configurable systems [3]. The operators are specifically designed for systems implemented with preprocessor techniques, which are widely used in practice [4, 27]. This proposed set of operators is the first to produce mutants that cannot only mimic faults in feature models but also in domain artifacts, which represent feature implementations, and in the mapping between model and code. We found that the proposed operators cause real variability faults and that approximately 50% of the analyzed faults are in domain artifacts.

In this paper, we build on this previous work and discuss potential techniques to reduce costs by influencing the way mutants are generated and tested. For this, we propose to utilize static variability analysis [24, 25] to identify at which point in the source code a mutation operator should be applied. Furthermore, we discuss to adopt the idea of T-wise testing [32, 34] to reduce the number of products we have to test. More precisely, we contribute the following:

- We discuss the characteristics of mutation operators proposed in previous work in more detail. In particular, we introduce a distinction between different types of mutation operators for domain artifacts of configurable systems. This can help to categorize and identify operators that are suitable for different test scenarios.
- We propose to utilize variability analysis and T-wise testing techniques to reduce the number of mutants. Hence, we provide a starting point for suitable cost reduction techniques and, thus, enable efficient mutation testing in configurable systems.
- We exemplify our approach on a running example and present a preliminary evaluation to demonstrate the potential applicability of the approach.

The remaining paper is structured as follows. In Section II, we provide background on mutation testing and configurable systems, focusing on preprocessors. Afterwards, we introduce our approach in Section III and present a preliminary evaluation in Section IV. We then describe related work in Section V and conclude in Section VI.

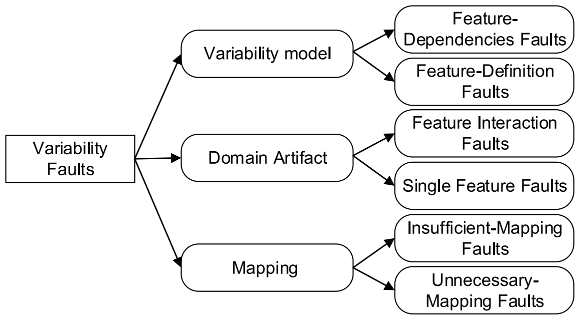
Variability Faults
Variability model
Domain Artifact
Mapping
Feature-Dependencies Faults
Feature-Definition Faults
Feature Interaction Faults
Single Feature Faults
Insufficient-Mapping Faults
Unnecessary-Mapping Faults

Fig. 1. Taxonomy of variability-based faults [3].

II. Background

In this section, we introduce foundations about mutation testing and variability in preprocessor-based systems.

A. Mutation Testing

With mutation testing, locations in the source code are selected at which syntactic changes are made to create a set of faulty programs called mutants [21]. The potential faults in mutants represent mistakes programmers often make. These mutants are then used to evaluate the ability of test cases to detect faults in the source code.

The main challenge in mutation testing are the corresponding costs of executing a large number of mutants against the test cases. Hence, several techniques have been proposed to reduce these costs. Offutt and Untch [31] cluster cost reduction techniques into three groups:

- **Do fewer**, where the number of mutants is reduced.
- **Do faster**, where the execution time of mutation testing is minimized.
- **Do smarter**, where the generation of mutants and their execution against test cases is optimized.

As we aim to reduce testing costs by influencing the way these mutants are generated and tested, we can cluster our approach in the *do fewer* and *do smarter* groups.

While mutation testing is an effective approach in assessing the quality of test cases, generating mutants to represent all potential faults for a program is unfeasible. Hence, it is common in mutation testing to target only specific types of faults [21]. In this article, we target variability faults, i.e., faults that appear only in some products in a configurable system. These variability faults are clustered into three groups [1, 3, 5, 15], which we illustrate in Figure 1. Firstly, model-based faults, which can occur in feature definitions or dependencies between them. Secondly, domain artifact faults, which can occur in domain artifact, for instance in the implementation. Finally, mapping-based faults, which can occur in mappings between model and domain artifacts. This taxonomy is based on the assumption that faults in generated products can be traced back to the aforementioned layers. We focus on faults in domain artifacts as they usually represent most faults [1, 3].

B. Variability in Preprocessor-Based Systems

In configurable systems, users can customize products to meet their requirements by selecting and deselecting configuration options [9] and, hence, deriving different products. Several techniques have been proposed to implement configurable systems [4]. In particular, preprocessor-based mechanisms, mainly the *C preprocessor* (Cpp) tool, are widely used in practice [22].

In preprocessor-based variability mechanisms, directives (a.k.a. macros) control syntactical program transformations. Those directives are clustered into four types: file inclusion, macro definition, macro substitution, and conditional inclusion [27]. For preprocessor-based systems, macro definitions (`#define`) and conditional inclusions (`#ifdef`) are the most important directives.

To illustrate how to implement variability with Cpp, we show a running example in Figure 2, where we illustrate a condensed code snippet from `Lynx`[1]. This example originally contains 100 lines of code and seven nested preprocessor conditionals in the file *LYLocal.c*. The macro `#define` is used to select features (e.g., `HAVE_TYPE_UNONWAIT` on Line 2). Logical operators (e.g., ||) are used in the Cpp to combine multiple features to complex expressions (e.g., `#if defined(__DJGPP__) || defined(_WINDOWS)` on Line 11).

A condition that controls the inclusion or exclusion of feature code is called a feature expression. Based on the output of a feature expression, i.e., true or false, the following source code up to the next, `#ifdef`, `#endif`, or `#else` is compiled or not. The presence or absence of feature code can be controlled with source code annotations `#ifdef` and `#ifndef`, for instance on Line 21 and Line 47, respectively. Each source code fragment that is encapsulated by `#ifdef` can be an optional feature, including other `#ifdef` macros, which are defined as nested `#ifdefs`. The specification of alternative features depends mainly on else cases.

III. Mutation Analysis

Applying mutation operators in the context of annotated variability is a challenging task. Using our running example in Figure 2, we illustrate how mutation operators can be applied. Basically, our approach is to describe the characteristics of a mutation operator to define the type of fault it injects. By utilizing variability analysis, we aim to identify at which points in the source code an operator can be used to introduce variability faults. To further reduce costs, we propose to adopt T-wise testing and assess only a subset of variants that cover all feature interactions of a specific degree.

In previous work, we defined three types of mutation operators for domain artifacts [3]:

- **Conditionally Applying Conventional Operator (CACO):** CACO applies a conventional mutation operator in the context of variability. For example, it may replace a logical operator in a variable code block or change a configuration option.
- **Removing Complete Ifdef Blocks (RCIB):** A RCIB operator removes an `#ifdef` block and, thus, changes the program behaviour for some products.

[1]http://lynx.invisible-island.net/, 15.11.2016

```
1  #include <HTUtils.h>
2  #define HAVE_TYPE_UNIONWAIT
3   // ...
4  static int LYExecv(char *path, char **argv, char *msg) {
5   int rc = 0;
6  #if defined(VMS)
7   CTRACE((tfp, " [...] ", path));
8  #else
9   int n;
10  char *tmpbuf = 0;
11 #if defined(__DJGPP__) || defined(_WINDOWS)
12  (void) msg;
13  stop_curses();
14  HTSprintf0(&tmpbuf, "%s", path);
15  for (n = 1; argv[n] != 0; n++)
16   HTSprintf(&tmpbuf, " %s", argv[n]);
17  HTSprintf(&tmpbuf, "\n");
18  rc = LYSystem(tmpbuf) ? 0 : 1;
19 #else
20  int pid;
21 #ifdef HAVE_TYPE_UNIONWAIT
22  union wait wstatus;
23 #else
24  int wstatus;
25 #endif /* HAVE_TYPE_UNIONWAIT */
26  if (TRACE) {
27   CTRACE((tfp, " [...] ", path));
28   for (n = 0; argv[n] != 0; n++)
29    CTRACE((tfp, " [...] ", n, argv[n]));
30  }
31  rc = 1;
32  stop_curses();
33  pid = fork();
34  switch (pid) {
35  case -1:
36   HTSprintf0(&tmpbuf, gettext(" [...] "), msg);
37   rc = 0;
38  break;
39  case 0:
40 #ifdef USE_EXECVP
41   execvp(path, argv);
42 #else
43   execv(path, argv);
44 #endif /* USE_EXECVP */
45   exit(EXIT_FAILURE);
46  default:
47 #ifndef HAVE_WAITPID
48   while (wait(&wstatus) != pid) ;
49 #else
50   while (-1 == waitpid(pid, &wstatus, 0)) {
51 #ifdef EINTR
52    if (errno == EINTR)
53    continue;
54 #endif /* EINTR */
55 #ifdef ERESTARTSYS
56    if (errno == ERESTARTSYS)
57    continue;
58 #endif /* ERESTARTSYS */
59    break;
60   }
61 #endif /* !HAVE_WAITPID */
62  if ( [...] ) {
63   HTSprintf0(&tmpbuf, gettext(" [...] "), msg);
64   rc = 0;
65  } }
66 #endif /* __DJGPP__ || _WINDOWS */
67  if (rc == 0)
68   LYSleepAlert();
69  start_curses();
70  if (tmpbuf != 0) {
71   if (rc == 0)
72    HTAlert(tmpbuf);
73   FREE(tmpbuf);
74  }
75 #endif /* VMS */
76  CTRACE((tfp, "LYexecv ->%d\n", rc));
77  return (rc);
78 }
```

Fig. 2. Nested variability in Lynx.

- **Moving Code around Ifdef Blocks (MCIB):** The MCIB operator moves code around an `#ifdef` block.

In this section, we further analyze these operators. Therefore, we first define two characteristics to describe such operators. Afterwards, we discuss corresponding cost reduction techniques.

A. *Characteristics of Domain Artifact Operators*

Based on the types of mutation operators, we span two dimensions to describe and classify them in the context of variability. These dimensions are the *type of change* (i.e., how to usefully apply it) and *affected variability* (i.e., which fault is injected). Hence, thee two dimensions help to argue at which point a mutation operator can be usefully applied.

1) Type of Change: We can differ mutation operators by the change they apply: either *syntactically* by mutating the content of a statement, or *structurally* by mutating the order of statements. Syntactical changes are the basis for most existing operators and include, for instance, string or regular expression replacements. Such operators simulate mistakes and typos done by developers and can be applied without further knowledge of a program's structure. For the three defined operators this behavior corresponds mostly with CACOs. For instance, in our example in Figure 2, a syntactical change can be made in the boolean condition on Line 11, where the logical operator && could be changed to || to manipulate the condition's output.

Structural operators require information on the source code and its variability. They do not mimic faulty written but faulty placed statements, for instance, a mandatory variable definition in an optional `#ifdef` block. Hence, it can be important to know at which position in the source code variability is placed. In the context of our operators, this includes RCIB, MCIB, and a set of CACO. A structural change in Figure 2 is to move Line 45 in front of Line 40, which may change the systems' behavior when the feature `Use_Execvp` is selected.

2) Affected Variability: Any applied mutation operator can affect either a code segment of a single feature (*Single Feature Fault*) or a variability point, where several features interact (*Feature Interaction Fault*). Mutating a single feature that does not interact with others limits the resulting mutants and does only assess this one. To do this, we have to ensure that no external dependencies, for instance a parameter that is changed by other features, exist and have to consider nested variability. However, assessing mutants on a single feature without its dependencies is not different from mutation testing in single systems.

In contrast, we focus on mutating feature interactions to assess the quality of test cases to handle the optional feature problem [23]. Testing several features at once is difficult due to feature dependencies, which makes it problematic to evaluate the results. For instance, if we mutate the variable definition of `errno` (Figure 2 Lines 52 and 56), it affects at least the conditionals `Eintr` and `Erestartsys`. Rodrigues et al. [38] refer to these dependencies as *maintenance* and *impact* points. At any maintenance or impact point, we can change a dependable variable by applying a mutation operator. Following impact points use this variable and, hence, are affected by the change.

With those two dimensions, we can describe the way a mutation operator affects domain artifacts. As we aim to introduce feature interaction faults to assess dependencies in the variable code, we also need to ensure that we apply operators accordingly.

B. Cost Reduction Techniques

Several cost reduction techniques have been proposed for mutation testing [7, 21, 29, 40]. Reducing costs is essential for configurable systems as we can create 2^n products from n optional configuration options. Hence, testing all possible products seems unrealistic [14], even without mutating them.

Budd [7] calculates the number of mutants to be in the order of the square of referenced variables. However, this depends on the mutation operators that are used and the changes these apply to the source code. For instance, Polo et al. [35] illustrate that the statement `return a+b` can be mutated at least in 20 different ways. Thus, it seems unfeasible to assess all possible mutants in every product and against all test cases.

To tackle this problem, we propose to combine *dependency analysis* and adopt the idea of *T-wise testing*. With the first, we aim to identify mutation points that, due to feature interactions, have a high probability of introducing variability faults. With the later approach we aim to then limit the subset of products to test in order that all T-wise feature interactions are considered at least once. We remark, that deselected features can also cause faults while testing, for instance because they define a variable that is used somewhere else [1].

1) Dependency Analysis: The first step to reduce the number of variability-based mutants is to analyze a systems' variable code and focus on mutating feature dependencies. For instance, testing a feature that does only print a log into a console does not consider any interactions. As a result of dependency analysis, we reduce the number of mutants in the source code as well as the number of products [32, 34]. Rodrigues et al. [38] define three types of feature dependencies in their work:

- **Intraprocedural** dependencies refer to situations in which the same code element, for instance a variable, is shared among features within one function. In Figure 2, for example, the variable `errno` is shared by `Eintr` (Line 52) and `Erestartsys` (Line 56), which are both part of the same function. Mutating one reference of the variable impacts all following references, which is why some configurations may behave incorrectly. Compared to other dependencies, such situations are easy to spot due to their locality.
- **Interprocedural** dependencies describe features that share a code element among different functions, for instance by passing a variable. We illustrate such a situation on Lines 41 and 43 of Figure 2, in which different functions are called, depending on the selected feature. Mutating the corresponding variables `path` and `argv` affects the behavior of different feature interactions.
- **Global** dependencies are similar to intraprocedural ones, but the shared element is defined globally and, thus, might be overlooked easier. In Figure 2, such a dependency exist on Line 26. There, the variable `TRACE` is referenced but not defined previously. We found that `TRACE` is defined with the Cpp in the file *HTUtils.h*. The variable is changed and referenced at several occasions, not only in this example, and interacts with different features.

In the context of this work, we focus on testing feature interactions and, thus, we need to identify such dependencies and mutate them. Still, we cannot guarantee that an operator will actually introduce a variability fault, but at least we increase the probability of generating one. For most CACO operators, we need to know which code affects which features. Based on this information, we are able to mutate only those statements that influence or are influenced by variability. To usefully apply RCIB, it is necessary to identify variable code blocks that have an impact on others. Removing an `#ifdef` block that does not interact will not help to test dependencies. Finally, for MCIB operators we need to know which lines of code are suitable to move around. Therefore, we must know which statements before or after an `#ifdef` block actually affect this block. Thus, we can reduce the number of useless mutants, for instance because they switch independent lines or move the same dependent code as part of larger blocks, and focus on significant ones. We aim to identify and assess such dependencies with static analysis, for instance by using the TypeChef tool [24, 25].

2) T-Wise Testing: To test mutants in configurable systems, we need to generate products based on the mutated domain artifacts. Creating all possible products is unfeasible in configurable systems, especially for those that have a large number of features. Hence, several approaches have been proposed to reduce the number of products to be tested, for instance T-wise testing [32, 34]. As a results, instead of testing all possible products, the goal is to identify a relevant subset that covers combinations of T features. We aim to utilize this idea and combine it with the dependency analysis to reduce the number of mutants. Basically, we propose to asses each mutant only once for each possible feature combination.

For this, the number of mutants can be reduced using T-wise testing as follows: If we apply pairwise testing, where $T = 2$, to generate products, we guarantee to cover all combinations of two features. Without combining feature dependencies and T-wise testing, there is a high probability that test cases will not kill the mutant. This is not due to the quality of test cases but because higher-order feature combinations may not appear in the generated products, as pairwise testing only guarantees to cover all combinations of feature pairs.

In addition, existing T-wise algorithms assume that all features interact with each other, which may not be the case in practice [36]. For our scenario, we aim to limit the number of feature interactions that are covered in the generated products by excluding features that do not interact. As input for this selection we can use the variability analysis during which we identify the dependencies.

Overall, we aim to utilize the characteristics of variability mutation operators by applying dependency analysis and T-wise testing. Thus, we can reduce the number of mutants and optimize their generation to suit configurable systems.

IV. Preliminary Evaluation

In this section, we present a preliminary evaluation to demonstrate the general feasibility our approach. We derive first insights of its applicability by considering our running example, which we show in Figure 2, and assessing the number of potential mutants for different strategies. We compare and discuss these results to argue that our approach is reasonable and provides a starting point for mutation testing in configurable systems.

For simplicity, we assume that we test the function `LYExecv` in Figure 2 as an independent unit. We only consider a single CACO operator: *Statement Deletion* (SSDL) [2] for our calculation. This operator is designed to show that each statement in the source code has an effect on the output. For this, it removes one statement, which can be a single line of code or a complete conditional block, at a time, ensuring that the program is still syntactically correct. Hence, preprocessor directives, variable definitions, or control flow statements are not removed.

At the beginning, we assess the costs of naive mutation testing considering neither cost reduction techniques nor feature interactions. In our example, we can remove 38 statements or conditional blocks to create mutants. Furthermore, 8 configuration options (e.g., `VMS`, `__DJGPP__`, `_WINDOWS`, `HAVE_TYPE_UNIONWAIT`, `USE_EXECVP`, `HAVE_WAITPD`, `EINTR`, `ERESTARTSYS`) are represented in the source code, wherefore 256 (i.e., 2^n) products can be configured. As a result, we would have $256 * 38 = 9728$ mutated products that we need to test. In the following, we illustrate how our approach can reduce this number.

First, we focus on feature dependencies and analyze which statements interact. A good example is the variable `rc` that is defined on Line 5. Several references to this variable are scattered among different features (e.g., on Lines 18, 31, 37, 64, 67, and 71). Thus, if we apply SSDL on any of these references, it may affect another feature's behavior. For instance, removing the statement `rc = 0` on Line 37 may change the outcome of conditions on Lines 64 and 67. In contrast, as there are no features afterwards, removing these two statements does not affect any interactions. Considering references that are later used in another feature and, thus, may affect it, we can find 6 statements and conditionals to mutate (e.g., on Lines 18, 31, 33, 34, 35, and 37). Those are also presented in Table I as points of maintenance. Mutating these statements will affect the corresponding variable and points of impact (cf. Table I). We remark, that some of the dependencies only appear when specific features are not selected [1].

While we now reduced the number of mutations in the domain source code, we secondly aim to test as few configurations as necessary. Therefore, we rely on our previous variability analysis and the idea of T-wise testing. 256 different configurations can be derived from the 8 options but some of these are syntactically identical. This is a result of feature interactions and encapsulated variability. For instance, on Line 6 it is checked whether `VMS` is defined. All other features only affect the code in the `else` clause and, thus, all configurations in which `VMS` is defined result in the same product.

TABLE I
PAIRWISE FEATURE DEPENDENCIES IN FIGURE 2.

Feature interactions	Variable	Point of Maintenance	Impact
¬VMS ∧ __DJGPP__	`rc`	18	67, 71
¬VMS ∧ _WINDOWS	`rc`	18	67, 71
¬VMS ∧ ¬__DJGPP__	`rc`	31, 35, 37	64
¬VMS ∧ ¬_WINDOWS	`rc`	31, 35, 37	64
¬__DJGPP__ ∧ HAVE_WAITPID	`pid`	33, 34	50
¬_WINDOWS ∧ HAVE_WAITPID	`pid`	33, 34	50
¬__DJGPP__ ∧ ¬HAVE_WAITPID	`pid`	33, 34	48
¬_WINDOWS ∧ ¬HAVE_WAITPID	`pid`	33, 34	48

Furthermore, we aim to utilize the idea of T-wise testing and cover all pairwise feature interactions. Applying T-wise testing for $T > 2$ requires additional configurations to assess all configurations. However, as we discussed in Section III, interactions between 3 or more features appear seldom in the created products [36]. Hence, corresponding mutants will rarely be killed despite the test cases having high quality. Based on the previous variability analysis, we identify those pairwise interactions that are potentially affected by mutations. For our example, this means we have to consider the feature dependencies displayed in Table I. Thus, we can cover these pairwise dependencies with 6 different configurations instead of using all 256.

Overall, we see in Table I that we can mutate each of the 6 configurations at 6 different maintenance points. Hence, we need only $6 * 6 = 36$ mutants to assess all feature interactions in our example. Compared to testing without previous analyses, we only consider approximately 0.37% of the previous 9728 mutants. However, we argue that those are the ones that allow an assessment of test cases for feature interactions. While these are preliminary results, they illustrate the potential of our approach. We have to remark, that we do not guarantee that we can detect all feature interactions with static analysis. Still, due to the idea of T-wise testing, we cover all pairwise feature interactions that are in the source code.

V. Related Work

In this section, we discuss existing work that is related to ours. There exist numerous approaches to utilize and optimize product lines testing [8]. However, in this section we focus on mutation testing for configurable systems and corresponding cost reduction techniques.

A. Mutation Testing in Configurable Systems

Arcaini et al. [5] propose to mutate feature models in order to find faults. They propose mutation operators that can mimic mistakes developers make during the design of feature models. Henard et al. [18] present a search-based approach to generate a set of products to detect faulty mutants in feature models. For this purpose, they create an altered version of feature models, which contain a fault within their propositional formula. However, the aforementioned approaches focus only on mutating feature models. In this paper, we discuss utilizing static analysis and T-wise testing to efficiently mutate the source code.

Lackner et al. [26] assess the quality of test cases for configurable systems by measuring their capability to detect faults. For this purpose, they comprise model-based mutation operators. Reuling et al. [37] propose to generate an effective set of configurations for fault-based configurable systems. Within their approach, they consider atomic and complex mutation operators. In this paper, we present a discussion on how the number of mutants can be reduced. These generated mutants can be introduced using not only model-based operators, but also domain artifact operators.

B. Cost Reduction Techniques

Several approaches have been proposed to reduce the costs of mutation testing by limiting either the number of generated mutants or the execution costs [21].

Early approaches to reduce the number of mutants aimed to sample subsets of these. For instance, Budd [7] and Zhang et al. [43] show that that small samples (10% and 5% respectively) are sufficient to achieve high accuracy with the full mutation score. Other authors further investigated this approach [30, 42]. Overall, different sampling strategies were proposed, for instance, by random [29], using clustering [11, 20], or using Bayesian sequential probability [39]. In contrast to these works, we do not sample the mutants based on a selection strategy but limit them to a specific problem (e.g., feature interactions) by applying variability analysis.

Other approaches aim to limit the used operators instead of sampling the mutants themselves [41]. For example, Deng et al. [10] found that the statement deletion operator, which we use for our evaluation, has a high accuracy while reducing the number of mutants significantly. Another way to reduce the cost is to optimize the execution process. Howden [19] proposes to execute only affected components in a mutant instead of the entire program. For our approach, we consider operators only if they address variability. Furthermore, we propose to utilize static analyses and T-wise testing to influence the generation and testing of mutants and, thus, reduce costs. Hence, our work is complementary to such approaches.

Devroey et al. [12, 13] focus on modeling mutants as variants of a product line based on feature transition systems. With heir approach, they limit test cases to those that reach a mutant, share common transition, and merge executions of the same states. This work is closely related to ours in its focus to reduce testing costs and can complement our approach. However, in contrast to mutating transition systems, we focus on the implementation level directly.

VI. Conclusion and Future Work

Mutation testing is a mature technique in single-system development used mainly to evaluate the quality of test cases. The main challenge in mutation testing is the computational costs due to the large number of mutants. Hence, several approaches have been proposed to reduce these costs by considering less mutants or optimizing the execution process.

In this paper, we discuss possibilities of reducing computational costs for mutation testing in configurable systems. Therefore, we propose to utilize static analysis and T-wise testing. Using these techniques will reduce the mutation testing costs, because:

- the number of considered products is reduced,
- fewer mutants are generated, and
- these mutants focus on variability faults.

Hence, our approach supports efficient mutation testing for configurable systems.

In future work, we plan to investigate three points. Firstly, due to the lack of an open-source program with test cases and an oracle to be used in such evaluations, we plan to generate these for a set of projects. Those can later be used to evaluate results and compare them. Secondly, we will implement a tool to generate mutants by focusing on domain mutation operators proposed in previous work [3]. Thirdly, in the aforementioned tool, we plan to integrate static analysis techniques, such as TypeChef [24, 25], in order to minimize computational costs to test feature interactions. Finally, we investigate how to omit equivalent or duplicated configurations and mutants in variability based on existing works, for instance by Papadakis et al. [33].

Acknowledgments

This research is supported by DFG grants LE 3382/2-1 and SA 465/49-1, and Volkswagen Financial Services AG.

References

[1] I. Abal, C. Brabrand, and A. Wasowski. 42 Variability Bugs in the Linux Kernel: A Qualitative Analysis. In *ASE*, pages 421–432. ACM, 2014.

[2] H. Agrawal, R. DeMillo, R. Hathaway, W. Hsu, W. Hsu, E. Krauser, R. J. Martin, A. Mathur, and E. Spafford. Design of Mutant Operators for the C Programming Language. Technical Report SERC-TR-41-P, Department of Computer Science, Purdue University, 1989.

[3] M. Al-Hajjaji, F. Benduhn, T. Thüm, T. Leich, and G. Saake. Mutation Operators for Preprocessor-Based Variability. In *VaMoS*, pages 81–88. ACM, 2016.

[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 *ICST*, pages 1–10, 2015.

[6] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *TSE*, 30(6):355–371, 2004.

[7] T. A. Budd. *Mutation Analysis of Program Test Data*. PhD thesis, Yale University, New Haven, CT, USA, 1980.

[8] I. D. Carmo Machado, J. D. McGregor, Y. a. C. Cavalcanti, and E. S. De Almeida. On Strategies for Testing Software Product Lines: A Systematic Literature Review. *IST*, 56(10):1183–1199, 2014.

[9] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[10] L. Deng, J. Offutt, and N. Li. Empirical Evaluation of the Statement Deletion Mutation Operator. In *ICST*, pages 84–93. IEEE, 2013.

[11] A. Derezińska. Toward Generalization of Mutant Clustering Results in Mutation Testing. In *SCCIS*, pages 395–407. Springer, 2015.

[12] X. Devroey, G. Perrouin, M. Cordy, M. Papadakis, A. Legay, and P.-Y. Schobbens. A Variability Perspective of Mutation Analysis. In *FSE*, pages 841–844. ACM, 2014.

[13] X. Devroey, G. Perrouin, M. Papadakis, A. Legay, P.-Y. Schobbens, and P. Heymans. Featured Model-Based Mutation Analysis. In *ICSE*, pages 655–666. ACM, 2016.

[14] E. Engström and P. Runeson. Software Product Line Testing – a Systematic Mapping Study. *IST*, 53(1):2 – 13, 2011.

[15] B. Garvin and M. Cohen. Feature Interaction Faults Revisited: An Exploratory Study. In *ISSRE*, pages 90–99, 2011.

[16] R. Gopinath, M. A. Alipour, I. Ahmed, C. Jensen, and A. Groce. On the Limits of Mutation Reduction Strategies. In *ICSE*, pages 511–522. ACM, 2016.

[17] R. G. Hamlet. Testing Programs with the Aid of a Compiler. *TSE*, 3(4):279–290, 1977.

[18] C. Henard, M. Papadakis, and Y. Le Traon. Mutation-Based Generation of Software Product Line Test Configurations. In *SBSE*, volume 8636 of *LNCS*, pages 92–106. Springer, 2014.

[19] W. E. Howden. Weak Mutation Testing and Completeness of Test Sets. *TSE*, 8(4):371–379, 1982.

[20] S. Hussain. Mutation Clustering. Master's thesis, King's College London, Strand, London, 2008.

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

[22] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *ICSE*, pages 311–320. ACM, 2008.

[23] C. Kästner, S. Apel, S. S. ur Rahman, M. Rosenmüller, D. Batory, and G. Saake. On the Impact of the Optional Feature Problem: Analysis and Case Studies. In *SPLC*, pages 181–190. Software Engineering Institute, 2009.

[24] 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 *OOPSLA*, pages 805–824. ACM, 2011.

[25] A. Kenner, C. Kästner, S. Haase, and T. Leich. TypeChef: Toward Type Checking #Ifdef Variability in C. In *FOSD*, pages 25–32. ACM, 2010.

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

[27] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *ICSE*, pages 105–114. IEEE, 2010.

[28] Y.-S. Ma, Y.-R. Kwon, and J. Offutt. Inter-Class Mutation Operators for Java. In *ISSRE*, pages 352–363. IEEE, 2002.

[29] A. P. Mathur and W. E. Wong. An Empirical Comparison of Data Flow and Mutation-Based Test Adequacy Criteria. *STVR*, 4(1):9–31, 1994.

[30] A. J. Offutt, G. Rothermel, and C. Zapf. An Experimental Evaluation of Selective Mutation. In *ICSE*, pages 100–107. IEEE, 1993.

[31] A. J. Offutt and R. H. Untch. In *Mutation Testing for the New Century*, chapter Mutation 2000: Uniting the Orthogonal, pages 34–44. Springer, 2001.

[32] S. Oster, F. Markert, and P. Ritter. Automated Incremental Pairwise Testing of Software Product Lines. In *SPLC*, pages 196–210. Springer, 2010.

[33] M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon. Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique. In *ICSE*, pages 936–946. IEEE, 2015.

[34] G. Perrouin, S. Sen, J. Klein, B. Baudry, and Y. Le Traon. Automated and Scalable T-Wise Test Case Generation Strategies for Software Product Lines. In *ICST*, pages 459–468. IEEE, 2010.

[35] M. Polo, M. Piattini, and I. G. R. de Guzmán. Decreasing the Cost of Mutation Testing with Second-Order Mutants. *STVR*, 19(2):111–131, 2009.

[36] E. Reisner, C. Song, K.-K. Ma, J. S. Foster, and A. Porter. Using Symbolic Evaluation to Understand Behavior in Configurable Software Systems. In *ICSE*, pages 445–454. ACM, 2010.

[37] 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 *SPLC*, pages 131–140. ACM, 2015.

[38] I. Rodrigues, M. Ribeiro, F. Medeiros, P. Borba, B. Fonseca, and R. Gheyi. Assessing Fine-Grained Feature Dependencies. *IST*, 78:27–52, 2016.

[39] M. Sahinoglu and E. H. Spafford. A Bayes Sequential Statistical Procedure for Approving Software Products. In *ASP*, pages 43–56, 1990.

[40] A. Siami Namin, J. H. Andrews, and D. J. Murdoch. Sufficient Mutation Operators for Measuring Test Effectiveness. In *ICSE*, pages 351–360. ACM, 2008.

[41] R. H. Untch. On Reduced Neighborhood Mutation Analysis Using a Single Mutagenic Operator. In *ACM-SE*, pages 71:1–71:4. ACM, 2009.

[42] W. E. Wong and A. P. Mathur. Reducing the Cost of Mutation Testing: An Empirical Study. *JSS*, 31(3):185–196, 1995.

[43] L. Zhang, M. Gligoric, D. Marinov, and S. Khurshid. Operator-Based and Random Mutant Selection: Better Together. In *ASE*, pages 92–102. IEEE, 2013.