Incremental Identification of T-Wise Feature Interactions

Sabrina Böhm
University of Ulm
Germany

Sebastian Krieter
University of Ulm
Germany

Tobias Heß
University of Ulm
Germany

Thomas Thüm
University of Ulm
Germany

Malte Lochau
University of Siegen
Germany

ABSTRACT

Developers of configurable software use the concept of selecting and deselecting features to create different variants of a software product. In this context, one of the most challenging aspects is to identify unwanted interactions between those features. Due to the combinatorial explosion of the number of potentially interacting features, it is currently an open question how to systematically identify a particular feature interaction that causes a specific fault in a set of software products. In this paper, we propose an incremental approach to identify such t-wise feature interactions based on testing additional configurations in a black-box setting. We present the algorithm Inciident, which generates and selects new configurations based on a divide-and-conquer strategy to efficiently identify the feature interaction with a preferably minimal number of configurations. We evaluate our approach by considering simulated and real interactions of different sizes for 48 real-world feature models. Our results show that on average, Inciident requires 80 % less configurations to identify an interaction than using randomly selected configurations.

CCS CONCEPTS

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

KEYWORDS

Software Product Lines, Configurable Systems, Feature Interaction, Feature-Model Analysis

ACM Reference Format:
Sabrina Böhm, Sebastian Krieter, Tobias Heß, Thomas Thüm, and Malte Lochau. 2024. Incremental Identification of T-Wise Feature Interactions. In *18th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS 2024), February 07–09, 2024, Bern, Switzerland.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3634713.3634715

1 INTRODUCTION

The goal of software product lines is to manage configurable software, where the variability is determined by configurations (i.e., feature selections) [5, 18]. Features can be combined in a configuration under the adherence to constraints, which can be represented, for instance, in a feature model [7, 8]. A feature model specifies all possible combinations of features that lead to valid configurations.

One of the biggest challenges associated with highly configurable software systems is to deal with the ever-increasing number of possible feature combinations [1, 5, 19, 30]. A particular combination of features may interact, causing undesired behavior, which was not intended or anticipated for the overall system. Conversely, some features are designed to work together and should therefore only occur in combination [38]. Unforeseen *feature interactions* can cause failures and security vulnerabilities [1, 19, 30, 38, 39, 41], emphasizing the importance of identifying these interactions [38].

While it is infeasible to generate and test all possible configurations of a product line, there are several techniques, such as combinatorial interaction testing, to identify failing configurations [2, 29, 34, 36, 50]. T-wise sampling algorithms aim to generate a small but representative subset of configurations that covers all possible combinations of t features [6, 22, 25–29, 36, 40, 43]. Although the majority of existing t-wise sampling algorithms generate a suitable sample, which contains failing configurations, they do not further investigate found failures. Thus, even if a failing configuration can be found, finding out the responsible feature interaction requires additional manual effort. Furthermore, as the number of potential interactions between t features grows polynomially with the total number of features and exponentially with the value of t, it is difficult to pinpoint a failure to a specific feature interaction [5].

For example, in the Linux kernel, which is regarded as one of the most complex configurable systems, developers are not able to test every possible configuration [20]. A Linux developer reported at the FOSD Meeting that problematic configurations (typically random configurations) can be found automatically during continuous integration, but locating the origin of the problem (i.e., the feature interaction) is a labor-intensive manual process. To the best of our knowledge, there exists no good strategy to help developers in finding the involved features. This paper wants to tackle the problem of identifying t-wise feature interactions causing failing configurations.

We propose the algorithm *Inciident* (*Inc*remental *i*nteraction *ident*ification) to identify faulty t-wise feature interactions by generating and testing further configurations. For this, we consider a black-box setting, which allows the algorithm to work independently of particular implementation artifacts and the concrete implementation technique (e.g., preprocessors or plug-ins). Given at least one failing configuration containing a specific fault, Inciident computes all potential feature-interaction candidates up to

VaMoS 2024, February 07–09, 2024, Bern, Switzerland
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0877-0/24/02
https://doi.org/10.1145/3634713.3634715

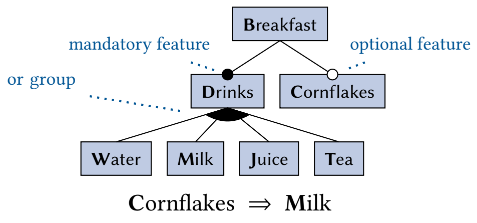

Figure 1: A feature model describing a breakfast scenario

an assumed interaction size t. Inciident then employs a divide-and-conquer strategy, which generates and tests new configurations, to narrow down the set of potential interactions until it is able to identify a single interaction causing the fault. Ideally, using this strategy, the number of newly generated and tested configurations is minimal and grows only logarithmically in the number of features. We consider Inciident to be incremental in two dimensions, (1) it incrementally generates new configurations and (2) incrementally increases the assumed interaction size t up to a given maximum. In summary, we contribute the following:

- We present the concept of *incremental identification of interactions* and the algorithm *Inciident* to determine feature interactions causing failing configurations (see Section 3).
- We evaluate our concept in terms of efficiency and effectiveness by trying to identify simulated and real t-wise feature interactions in 48 real-world feature models (see Section 4).
- We provide an open-source implementation and a replication package of our experiments [10] (see Section 4.1).

2 BACKGROUND

In the following, we describe fundamentals of software product lines, focusing on the concepts of feature models (Section 2.1) and configurations (Section 2.2). Further, we explain the notion of feature interactions in a nutshell (Section 2.3).

2.1 Feature Models

A *feature model* describes the *problem space* of an entire software product line, which represents the requirements and views of the corresponding stakeholders [5, 18]. A feature model defines the possible combinations of features, from which valid configurations can be inferred. In comparison, the implementation artifacts of a product line describe the *solution space*, which represents the actual products that can be derived from valid configurations [5, 18].

Definition 2.1 (Feature Model). A feature model is a tuple $\mathcal{M} = (F, D)$, where $F = \{f_1, \dots, f_n\}$ is a set of features and $D = \{d_1, \dots, d_m\}$ is a set of dependencies over a set of Boolean literals $\mathbb{L}(\mathcal{M}) = \{\neg f_1, \dots, \neg f_n, f_1, \dots, f_n\}$, where $f_i \in F, 1 \leq i \leq n$.

In Figure 1, we depict an example feature model as a feature diagram [7, 8]. Due to brevity, in the following, we use abbreviations for the feature names represented by the first letter of their name.

2.2 Configurations

We create a configuration by selecting features for a desired software product. Our formal notation of a configuration is based on the in- and exclusion of features from a feature model.

Definition 2.2 (Configuration). Let $\mathcal{M} = (F, D)$ be a feature model, a configuration $c = \{l_1, \dots, l_k\} \in \mathbb{L}(\mathcal{M})$ with $\{l, \neg l\} \not\subseteq c$ is a set of literals containing at most one literal per feature (i.e., either positive, negative, or none). A feature $f \in F$ is included in c, iff c contains its positive literal and excludes f, iff c contains its negative literal.

- A configuration c is called *partial*, iff the configuration does not contain a literal for every feature (i.e., $|c| < |F|$). Otherwise, it is called *complete*.
- A configuration c is called *valid*, iff there exists a complete configuration $c' \supseteq c$ that satisfies all dependencies in D. Otherwise, it is called *invalid*.
- The *configuration space* $\mathbb{C}(\mathcal{M})$ is the set of all valid configurations of a feature model $\mathcal{M}$.

In the remainder of the paper, we use the term *configuration* to refer to a *complete valid configuration*, if not stated otherwise.

From the configuration space $\mathbb{C}(\mathcal{M})$, we can infer certain feature properties [7]. We call a feature *core* if there exists no valid configuration for the feature model, in which the feature is deselected (e.g., B). Analogously, we call a feature *dead* if there exists no valid configuration, in which the feature is selected. A feature that is neither core nor dead we call a *variant* feature (e.g., C).

Each configuration describes a subset of the entire configuration space $\mathbb{C}(\mathcal{M}, c)$, consisting of all valid configurations that are a superset of the given configuration (i.e., $\mathbb{C}(\mathcal{M}, c) = \{c' \in \mathbb{C}(\mathcal{M}) \mid c' \supseteq c\}$). Note that, for a partial configuration c there may exist another configuration $c' \supset c$ that describes the same subset of the configuration space (e.g., $c = \{\neg M, J\}$ and $c' = \{B, D, \neg M, J, \neg C\}$). This happens when some features become *conditionally core* or *dead* [8] under a partial configuration. By adding all literals of conditionally core and dead features to a partial configuration c, we get the largest (partial) configuration $\hat{c}$ that describes a particular configuration space (i.e., $\forall c \in \mathbb{C}(\mathcal{M}) : \mathbb{C}(\mathcal{M}, \hat{c}) = \mathbb{C}(\mathcal{M}, c) \land |\hat{c}| \geq |c|$).

2.3 Feature Interactions

A feature interaction of one or more features can be seen as a particular behavior that cannot be deduced from the individual behaviors associated to the features involved [5]. In this work, we deliberately abstract from concrete implementation artifacts in the solution space and instead investigate feature interactions from a problem space point of view.

Definition 2.3 (Feature Interaction). A t-wise feature interaction I is a valid (partial) configuration of size t (i.e., $|I| = t$) that contains only literals of variant features. Given a feature model $\mathcal{M}$, the set $\mathcal{I}^t(\mathcal{M})$ contains all possible t-wise feature interactions of $\mathcal{M}$.

We call an interaction with interaction size $t = 1$ a one-wise interaction (e.g., $I = \{W\}$), for $t = 2$ a pair-wise interaction (e.g., $I' = \{W, \neg J\}$), and for all $t > 2$ a higher-order interaction. It is crucial to not omit deselected features, as absence of features can also be responsible for an unexpected software behavior [1, 19]. We limit our considered feature set for interactions to variant features due to the non-existing variability induced by core and dead features.

Approaches for t-wise interaction testing generate and test a small but representative subset of all configurations (i.e., a sample S) [22, 25–29, 40, 43]. These approaches aim to achieve full t-wise coverage of the tested system, which means that every interaction

i that contains t features must be present (i.e., covered) in at least one configuration (i.e., $\forall i \in \mathcal{I}^t(\mathcal{M})\ \exists c \in S : i \subseteq c$). This is a well-known problem in the field of sampling, where these numerous combination possibilities of features challenge the creation of a representative sample for the system [26, 36].

As an example, the feature model in Figure 1 has 39 feature combinations (e.g., W $\wedge \neg$ M) that must be covered for full pair-wise coverage. Let us assume that the configuration $c = \{\neg W, M, J, T, \neg C\}$ fails during testing. From this information alone, we cannot deduce that the failure is caused by the interaction $i = \{M, \neg C\}$. At this point, we would need to manually investigate the origin of the fault. Thus, we aim to improve this process by providing guidance to developers through proposing a set of potentially involved features to identify the feature interaction.

3 IDENTIFICATION OF INTERACTIONS

We propose the concept of *incremental identification of interactions*, for finding the smallest t-wise feature interactions responsible for a given fault. In Algorithm 1, we propose our corresponding base algorithm. Given at least one failing configuration, the algorithm generates and tests additional configurations to incrementally decrease the set of t-wise feature interactions potentially responsible for the failure.

Algorithm 1: Base algorithm for interaction identification

```
Input   : feature model $\mathcal{M}$, interaction size $t$
Output  : potential interactions $P$
Global  : configuration pool CP
1  $C_\times \leftarrow$ failing configurations from CP
2  $C_\checkmark \leftarrow$ non-failing configurations from CP
3  $P^t \leftarrow$ compute potential interactions from $C_\times$ and $C_\checkmark$
4  while $|P^t| > 1$ and $c \leftarrow$ generate configuration for $\mathcal{M}$ do
5      if test $c$ then
6          $C_\checkmark \leftarrow$ add $c$
7          $P^t \leftarrow$ remove interactions from $c$
8      else
9          $C_\times \leftarrow$ add $c$
10         $P^t \leftarrow$ keep interactions from $c$
11     end
12 end
13 CP $\leftarrow C_\times$ and $C_\checkmark$
14 return if validate $P^t$ then $P^t$ else $\emptyset$
```

As input, we require a feature model and a value for the presumed interaction size t. We start with a global *configuration pool* CP, which contains an initial set of tested configurations. First, we partition the set of configurations from CP into the sets $C_\times$, containing all failing, and $C_\checkmark$, containing all non-failing configurations. Second, we compute the set of all potential interactions P^t for the input interaction size t based on the known configurations. To this end, we compute the literals common to all failing configurations and then generate all possible t-tuples over this set, excluding every tuple that is contained in any configuration from $C_\checkmark$. Third, we generate and test a new configuration c and, based on the test result, we reduce the set of potential interactions P^t. Each new configuration is either added to the set $C_\times$ or $C_\checkmark$. The algorithm repeats the third step until either it cannot generate any new configuration c or the potential interactions P^t contain less than two interactions. Finally, the algorithm either returns the set P^t or no result.

To be able to apply our algorithm, we make the following important assumptions. The fault we are looking for (1) is produced by exactly one feature interaction and (2) can be detected in a reliable and reproducible manner. In this paper, we abstract the testing process by considering a black-box oracle with fault information to be able to identify the interaction for any given configuration (e.g., via unit testing). We deliberately agree to these assumptions, as we currently aim to demonstrate the feasibility of the overall approach of improving the manual process for identifying feature interactions.

3.1 Compute Potential T-Wise Interactions

It is crucial for our algorithm to compute all t-wise interactions that could be candidates for causing failing configurations. We define the set of all potentially faulty t-wise feature interactions, also called *potential interactions*, as follows.

Definition 3.1 (Potential Feature Interactions). Let c be a failing configuration and t a desired interaction size, the function $p^t(c) = \{I \in \mathcal{I}^t(\mathcal{M}) \mid I \subseteq c\}$ maps c to the set P^t of all potential t-wise feature interactions causing the configuration c to fail.

We know that the faulty interaction has to be contained in every failing configuration and, analogously, it cannot be part of a correct configuration. Thus, we can compute the set of potential faulty interactions by including all t-wise interactions from $C_\times$ and excluding all t-wise interactions from $C_\checkmark$. Thus, we first create an initial set of all potential interactions P_0^t from one configuration in $C_\times$ (i.e., $P_0^t = p^t(c_0)$), and then iteratively refine the set for each other configuration, depending on whether the configuration passes (i.e., $P_{k-1}^t \setminus p^t(c_k)$) or fails (i.e., $P_{k-1}^t \cap p^t(c_k)$). To further refine the set P_k^t, we incrementally generate and test new configurations. Exemplary, we consider the failing configuration $c = c_0 = \{\neg W, M, J, T, \neg C\}$ from Section 2.3. For an interaction size $t = 2$ we can compute all potential pair-wise interactions P_0^2 based on the literals in c_0 (i.e., $P_0^2 = \{\{\neg W, M\}\{\neg W, J\}, \{\neg W, T\}, \{\neg W, \neg C\}, \{M, J\}, \{M, T\}, \{W, \neg C\}, \{J, T\}, \{J, \neg C\}, \{T, \neg C\}\}$).

In Table 1, we provide an overview of the further reduction procedure of the potential interactions P^t after generating and testing configurations. We assume to have some procedure that generates an arbitrary configuration on demand. To further reduce the set P_0^2, we proceed by generating a new configuration, for instance $c_1 = \{\neg W, \neg M, J, T, \neg C\}$, and test if it fails. Assume that c_1 passes the test and, therefore, we exclude all interactions that are contained in c_1. Note that c_1 is now stored in the set of non-failing configurations $C_\checkmark$. Next, we reduce the set of all potential interactions P_0^2 (i.e., $|P_0^2| = 10$) to P_1^2 (i.e., $|P_1^2| = 4$), as shown in Table 1. We generate the next configuration $c_2 = \{W, M, \neg J, T, \neg C\}$ and test if it fails. Suppose c_2 fails and, therefore, we exclude all interactions that are not contained in c_2. The considered configuration reduces P_1^2 to P_2^2 (i.e., $|P_2^2| = 2$). Then, we save c_2 in the set of failing configurations $C_\times$. We continue to generate a next configuration $c_3 = \{\neg W, M, \neg J, \neg T, \neg C\}$ and assume that its test fails. We update

Table 1: Example of incremental identification of interactions for the failing configuration $c_0 = \{\neg W, M, J, T, \neg C\}$

$c_0 = \{\neg W, M, J, T, \neg C\}$ $\lightning$		$c_1 = \{\neg W, \neg M, J, T, \neg C\}$ ✓		$c_2 = \{W, M, \neg J, T, \neg C\}$ $\lightning$		$c_3 = \{\neg W, M, \neg J, \neg T, \neg C\}$ $\lightning$	
$p^2(c_0)$	$P^2_0 = p^2(c_0)$	$P^2_0 \cap p^2(c_1)$	$P^2_1 = P^2_0 \setminus p^2(c_1)$	$P^2_1 \cap p^2(c_2)$	$P^2_2 = P^2_1 \cap p^2(c_2)$	$P^2_2 \cap p^2(c_3)$	$P^2_3 = P^2_2 \cap p^2(c_3)$
$\{\neg W, M\}$	$\{\neg W, M\}$		$\{\neg W, M\}$		$\{\neg W, M\}$		
$\{\neg W, J\}$	$\{\neg W, J\}$	$\{\neg W, J\}$	$\{\neg W, J\}$				
$\{\neg W, T\}$	$\{\neg W, T\}$	$\{\neg W, T\}$	$\{\neg W, T\}$				
$\{\neg W, \neg C\}$	$\{\neg W, \neg C\}$	$\{\neg W, \neg C\}$	$\{\neg W, \neg C\}$				
$\{M, J\}$	$\{M, J\}$		$\{M, J\}$		$\{M, J\}$		
$\{M, T\}$	$\{M, T\}$		$\{M, T\}$	$\{M, T\}$	$\{M, T\}$		$\{M, T\}$
$\{M, \neg C\}$	$\{M, \neg C\}$		$\{M, \neg C\}$	$\{M, \neg C\}$	$\{M, \neg C\}$	$\{M, \neg C\}$	$\{M, \neg C\}$
$\{J, T\}$	$\{J, T\}$	$\{J, T\}$	$\{J, T\}$				
$\{J, \neg C\}$	$\{J, \neg C\}$	$\{J, \neg C\}$	$\{J, \neg C\}$				
$\{T, \neg C\}$	$\{T, \neg C\}$	$\{T, \neg C\}$	$\{T, \neg C\}$				

the interactions from P^2_2 and reduce the potential interactions to P^2_3 (i.e., $|P^2_3| = 1$). As there is only one remaining interaction in P^2_3 (i.e., $\{M, \neg C\}$), the algorithm returns this interaction as cause of the failing configurations.

3.2 Generate New Configurations

The generation of new configurations can be done in several ways, for instance by generating random configurations. Considering the configurations in the example, we have deliberately chosen helpful configurations, as each new configuration has excluded further interactions and has narrowed down the set of potential interactions P^t. However, by choosing a random generation approach, we may not be able to exclude interactions from P^t in every iteration (e.g., $c'_3 = \{W, \neg M, \neg J, \neg T, \neg C\}$ instead of c_3).

The choice of the next configuration is a crucial factor within our approach. Using a random approach can lead to testing many configurations that do not provide any new knowledge about the remaining potential interactions. Ideally, we would generate configurations that contain half of the remaining potential interactions and do not contain the other half. Thus, independent of the test result for each individual configuration, we would be able to exclude half of the potential interactions with each iteration. This approach would lead to a testing effort that increases only logarithmically in the number of interactions. Taking a closer look at our example in Table 1, we have configurations that fulfill these criteria and we see that our reduction of P^t is approximately logarithmically.

Generating such configurations is not trivial in practice, which is why we use a greedy strategy by generating multiple random configurations and test the one closest to covering about half of the remaining interactions. Note that, aside from the configuration we select for testing, we only generate configurations in the problem space without deriving any products in the solution space. Nevertheless, we limit the number of configurations that we randomly generate (i.e., in each iteration we generate $\lceil 2 \cdot \log_2 |P^t_0| \rceil$ configurations). This limit is based on the initial set of potential interactions, such that it generates more configurations for larger feature models, but is logarithmically bounded to maintain a feasible run time.

In case we cannot find any random configuration that includes and excludes at least one potential interaction from P^t_k, we attempt to generate such a configuration using the satisfying assignment returned by a SAT solver. In particular, we create a propositional formula based on the feature model with the additional constraint that at least one interaction from P^t_k must be selected and at least one must be deselected. To achieve this, we substitute each interaction from P^t_k with a fresh variable. Given a feature model $\mathcal{M} = (F, D)$ with $D = \{d_1, \ldots, d_m\}$ and $P^t_k = \{I_1, \ldots, I_i\}$, the resulting formula, which we use as input for the SAT solver, is:

$$\begin{aligned} d_1 \wedge \ldots \wedge d_m \wedge \quad & \textit{original formula of } \mathcal{M} \\ (x_1 \Leftrightarrow I_1) \wedge \ldots \wedge (x_i \Leftrightarrow I_i) \wedge \quad & \textit{substitutions} \\ (x_1 \vee \ldots \vee x_i) \wedge \quad & \textit{include at least one} \\ (\neg x_1 \vee \ldots \vee \neg x_i) \quad & \textit{exclude at least one} \end{aligned}$$

If the SAT solver is not able to find a configuration for this formula, the current set of potential interactions cannot be reduced any further, because the remaining candidates specify the same subset of the configuration space. In this case, we terminate the process and return the set P^t_k.

3.3 Inciident

In practice, we usually do not know the interaction size t of the feature interaction causing failing configurations. To tackle this issue, Inciident adapts the input interaction size dynamically and runs multiple iterations of the base algorithm, starting with an input interaction size of $t = 1$ until it reaches a given limit t_{max}.

In Figure 2, we present the main steps of Inciident. In Step 1, we take the input data consisting of the feature model, a failing configuration, and the input interaction size t_{max}. In Step 2, we set the input interaction size t (starting with $t = 1$). Afterward, we proceed with Step 3, the computation of all potential t-wise interactions. Next, we perform the *configuration loop*, which consists of Step 3 to Step 5. After generating (Step 4) and testing (Step 5) a configuration, we proceed with Step 3 and refine the set of potential interactions P^t (cf. Section 3.1). We finish the current interaction-size loop if P^t is empty, contains only one interaction, or if it is not possible to generate a configuration that includes and excludes at least one interaction from P^t. After computing a result for the current input interaction size t, we proceed with Step 2 and perform the *interaction-size loop* by increasing the interaction size t. When

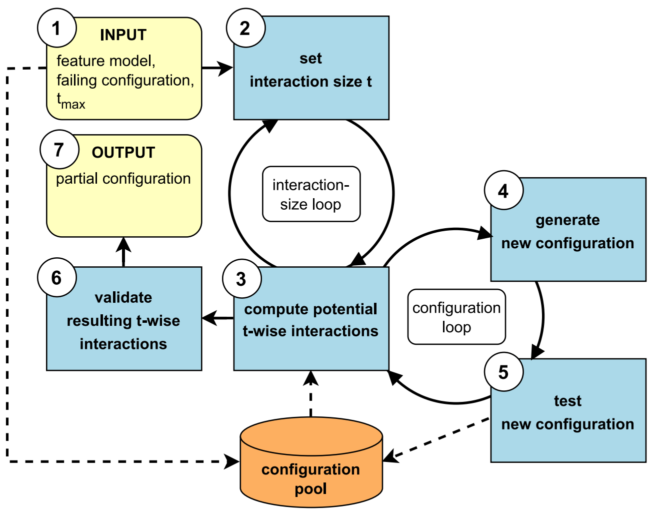
1
INPUT
feature model, failing configuration, t_max
2
set interaction size t
interaction-size loop
7
OUTPUT
partial configuration
4
generate new configuration
6
validate resulting t-wise interactions
3
compute potential t-wise interactions
configuration loop
5
test new configuration
configuration pool

Figure 2: Steps of the algorithm Inciident

we reach t_{max}, we proceed with Step 6, where we merge the results from each iteration into a single feature interaction (i.e., a partial configuration) and validate the result (cf. Section 3.4). In Step 7, our algorithm terminates by returning the computed feature interaction or the empty set if the result could not be validated.

Inciident makes use of intermediate results from each interaction-size loop. In each loop, it re-computes the set of potential interactions, reusing all configurations from the set CP. This potentially reduces run time and memory usage, as more potential interactions can be excluded before generating and testing new configurations.

3.4 Validate Resulting T-Wise Interactions

In Step 3, Inciident produces a set of potential interactions P^t for each value of t. As a final step, we have to check whether the resulting output is correct. An output may be incorrect, if the bound value t_{max} is too small or any of our assumptions is violated.

Given a set P^t, we determine a partial configuration c_p^t by computing the union set over all interactions in the set P^t (i.e., $c_p^t = \bigcup P^t$). This is possible, because all interactions in P^t are subsets of the initial failing configuration c_0 and, thus, the resulting partial configuration c_p^t is also a subset of c_0. Then, we determine $\hat{c}_p^t$ by adding all literals that c_p^t implies but not already contains (cf. Section 2.2). We argue that this is a useful step to avoid returning a misleading result. Within our black-box approach (i.e., considering only the problem space), we cannot distinguish the observable behavior of $\hat{c}_p^t$ and c_p^t, as every configuration that contains c_p^t also contains $\hat{c}_p^t$. Hence, we return $\hat{c}_p^t$, which contains all potentially relevant literals.

Second, we search the smallest value $t \leq t_{max}$ for which $\hat{c}_p^t$ still contains the feature interaction causing the fault. Given a value for t, we know that $\hat{c}_p^t$ does not contain the complete interaction if there exists a non-failing configuration $c_{\checkmark}^t$ that includes all interactions from P^t and excludes at least one interaction from P^{t+1}. Consequently, we look for the largest value $t \leq t_{max}$ for which there exists a $c_{\checkmark}^{t-1}$ but no $c_{\checkmark}^t$. Starting with $t = t_{max}$, we try to compute $c_{\checkmark}^{t-1}$, subsequently decreasing t by one until we are successful, in which case we select $\hat{c}_p^t$ as the correct partial configuration.

Finally, we validate the selected configuration $\hat{c}_p^t$ using a further test to cross-check if it is a plausible output. We generate two further configurations, one containing and one not containing $\hat{c}_p^t$. If the configuration containing $\hat{c}_p^t$ fails, we continue by testing the configuration without $\hat{c}_p^t$. If it passes, we terminate the algorithm and output $\hat{c}_p^t$. If one of the considered configurations does not correspond to the desired result, we terminate our algorithm with no result as an indicator that the chosen value t_{max} is too small.

In our example, we set P^2 to $\{\{M, \neg C\}\}$. Assume, we also identified $P^1 = \{\{M\}\}$ in the iteration for $t = 1$. First, we compute $\hat{c}_p^1 = \{M\}$ and $\hat{c}_p^2 = \{M, \neg C\}$. Second, we discard $\hat{c}_p^1$, because we can find the non-failing partial configuration $\{M, C\}$, which contains $\{M\}$, but not $\{M, \neg C\}$. Consequently, we select $\hat{c}_p^2$. Finally, we generate two test configurations, the first configuration including all interactions from P^2 (e.g., $c_{v1} = \{\neg W, M, J, \neg T, \neg C\}$) and the second excluding all interactions (e.g., $c_{v2} = \{\neg W, M, \neg J, \neg T, C\}$). We assume that c_{v1} fails and c_{v2} does not, which passes our validation successfully. Hence, we identified the partial configuration $\hat{c}_p^2 = \{M, \neg C\}$ as the cause of the failing configurations.

3.5 Discussion and Limitations

We start our algorithm when receiving a particular test result for at least one failing configuration. For most real-world faults, we have more knowledge, such as particular error messages, which sometimes can help to find the origin of failing configurations. To apply our concept, we assume to have a black-box scenario with configuration tests that lead to reproducible results for each failing configuration. This may limit the effectiveness of our approach because we cannot rely on having perfect fault information for real-world systems.

Considering our assumptions, we currently assume to have only one feature interaction in our system that causes failing configurations, which is not always given in practice. We accept this limitation to show the feasibility of our divide-and-conquer approach as presented. Therefore, when it comes, for instance, to error masking, our current approach does not guarantee to identify the interaction causing failing configurations, but it is conceivable to apply our concept by looking at each fault independently. Furthermore, we assume the occurring fault can be detected in a reproducible manner to be able to get the same result for each configuration. We would argue that the limitations we impose constitute a good trade-off to show the feasibility of our approach to automate the manual process of identifying the origin of the fault.

When computing the partial configuration for a set of potential interactions, we may have have more literals included in the configuration than involved in the interaction actually causing failing configurations. More precisely, due to the post-processing in Step 6 of Inciident, we always get the largest possible configuration that comprises the same configuration space as the partial configuration containing only the actual feature interaction.

4 EXPERIMENTAL EVALUATION

In this section, we present the results of evaluating our concept of incremental identification of t-wise interactions. We evaluated 48 real-world feature models by simulating one-, pair-, and three-wise feature interactions. We compare Inciident to a random approach (Random) to guide the selection of the next configurations to be tested. We base our evaluation on criteria commonly used in the context of sampling algorithms to determine their testing effectiveness, testing efficiency, and sampling efficiency [39, 49, 50]. Our focus is to investigate the effectiveness of Inciident to identify the interactions, the number of tested configurations, and the time required to perform the algorithm. In our evaluation, we address the following research questions.

RQ_1 How effectively can we identify the feature interaction that leads to failing configurations?

RQ_2 What computational effort is required for identifying the feature interaction?

$RQ_{2.1}$ How many configurations have to be tested to identify the feature interaction?

$RQ_{2.2}$ How much computation time is required to identify the feature interaction?

In RQ_1, we investigate how precisely we can identify a particular feature interaction based on failing configurations. We distinguish for the identified interaction, whether it is exact, a superset, subset, or different, and how often we get no result.

The research question concerning computational effort is divided in two parts. In $RQ_{2.1}$, we consider the number of configurations that have to be tested during our algorithm. This is crucial because we only achieve applicability of our approach if the number of tested configurations is realistic for real-world systems. Related to $RQ_{2.1}$, $RQ_{2.2}$ considers the computational time to perform our algorithm. Note that we exclude the time for deriving and testing actual products from generated configurations, as we abstract from the solution space, considering a black-box setting with perfect fault information.

4.1 Algorithms

As there exists no state-of-the-art algorithm that realizes interaction identification the way we do, we cannot compare ourselves to other algorithms. We expect that the strategy for generating further configurations, the input interaction size t_{max} and the actual interaction size have an impact on the number of configurations tested (cf. Section 3.2), which we investigate in our evaluation. Therefore, we evaluate our concept by varying the most crucial step within our algorithm, the strategy for configuration generation (i.e., Step 4 from Figure 2) and selection of the input interaction size.

Inciident performs the identification of interactions until the input interaction size t_{max} is reached. We choose the configuration to be tested next deliberately as described in Section 3.2. The algorithm Random generates configurations randomly using the SAT solver Sat4J [31] (i.e., non-uniform random sampling).

To avoid testing every possible configuration, we set a maximum number of configurations to be tested (i.e., $\lceil 10 \cdot \log_2(P_0^t) \rceil$). Similar to the generation limit of Inciident, this limit allows to generate more configurations for feature models with more potential interactions. For almost all experiments, this limit lies above the number of configurations required by Inciident. Thus, Random also terminates when the maximum number of configurations is reached.

We provide an open-source implementation of Inciident[1] written in Java and based on the FeatJAR library[2]. In addition, we provide a replication package with the algorithms used in our evaluation and the data generated by our experiments [10].

4.2 Experiment Setup

To evaluate our concept, we perform several experiments to find multiple t-wise feature interactions for different systems.

4.2.1 Feature Interactions. We consider 48 feature models from different sources and from different domains, namely finance, systems software, e-commerce, gaming, and communication [24, 37, 44–46]. The chosen models cover a wide range of number of features (9–3,296) and number of constraints (13–15,692). For a wide range of feature-model sizes, we selected small- and medium-sized feature models from examples provided by the tool FeatureIDE [37]. We also used feature models from real-world Kconfig systems, provided by Pett et al. [44], for which we chose the earliest and latest versions for each system. In addition, we used more complex, real-world feature models [24, 45, 46]. Moreover, we use six models with different sizes of the eCos system from Knüppel et al. [24]. For all feature models containing less than 800 features (35), we simulate 100 one-wise, 100 pair-wise, and 100 three-wise interactions per feature model, per algorithm, and per input interaction size t. For feature models containing between 800 and 3,000 features (13), we simulate 10 one-, pair-, and three-wise interactions. For the largest feature model, which contains 3,296 features, we simulate only 3 one-, pair-, and three-wise interactions due to the high computation time of Random for larger models.

To simulate an interaction, we generate a random configuration and based on that we extract a partial configuration by randomly choosing a subset with t features that are neither core nor dead. Additionally, we use the random configuration as failing input configuration c_0 for our algorithm. Note that, as we choose the partial configuration for the simulated fault randomly, it may imply additional literals (i.e., conditionally core and dead features). In this case, we add these literals to the configuration to ease the comparison to the output of the algorithms in our evaluation (cf. Section 2.2 and Section 3.4). To ensure good comparability within our results, we use the same set of random configurations per model for all one-, pair-, and three-wise feature interactions, making sure that each one-wise interaction is a subset of a pair-wise interaction, which in turn is a subset of a three-wise interaction.

In addition, we use 10 real-world faults caused by feature interactions in the systems BusyBox (v1.23.1) and Linux (v3.18.5). Nine of the faults are caused by a single one-, two-, three-, or four-wise interactions. One fault is caused by two alternative one-wise interactions (i.e., $I_1 \vee I_2$), violating our first assumption.

4.2.2 Measurements. Our independent variables are the underlying feature model, chosen algorithm, actual interaction size t_a, and input interaction size t_{max}. The seed for the pseudo-random number generator is a control variable to enable replication. The

[1]https://github.com/skrieter/evaluation-interaction-analysis
[2]https://github.com/FeatureIDE/FeatJAR

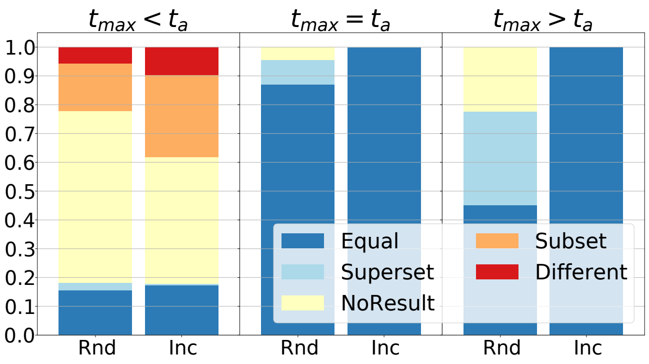

Figure 3: Effectiveness of identifying the interaction

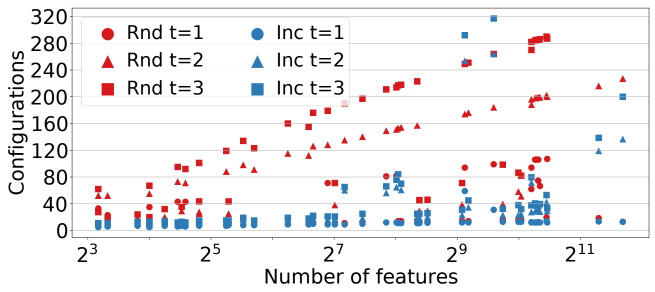

Figure 4: Median number of tested configurations by Inciident and Random per t_{max} and feature model

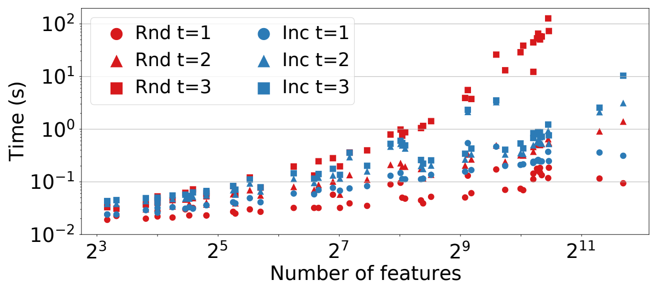

Figure 5: Median run time of Inciident and Random per t_{max} and feature model

dependent variables are the *number of tested configurations* and the *run time* that is required for each execution of one experiment.

As a limit for the input feature interaction size, we consider $t_{max} \in \{1, 2, 3\}$, because most of the known interactions do not contain more features [1, 20, 30]. Our test oracle checks whether a given configuration contains the simulated interaction.

Regarding RQ_1, we analyze the effectiveness of the identification of the simulated interactions in our approach. To this end, we differentiate between five result types. First, the found interaction is equal to the simulated interaction. Second, the found interaction is a subset of the simulated interaction (e.g., we simulated the interaction $\{\neg W, M, J\}$ and our result is $\{\neg W, J\}$). Third, the found interaction is a superset of the simulated interaction (e.g., we simulated the interaction $\{\neg W, M\}$ and our result is $\{\neg W, M, J\}$). Fourth, the found interaction is not empty and neither a sub- nor a superset of the simulated interaction (e.g., we simulated the interaction $\{\neg W, M\}$ and our result is $\{\neg W, T\}$). Fifth, the algorithm cannot compute a result.

For $RQ_{2.1}$, we consider the number of configurations we have to test in order to correctly identify an interaction. Regarding $RQ_{2.2}$, we inspect the time needed to perform the incremental identification of interactions for all experiments where we identified the simulated interaction.

4.2.3 System Specifications. We ran our experiments on a server with the following specifications: *CPU*: 2x Intel Xeon E5-2630 v3 @ 2.4 GH; *RAM*: 256 GB DDR3; *OS*: Ubuntu 22.04.2 LTS; *Java*: openjdk 17.0.6.; *JVM memory*: 64 GB

4.3 Results

4.3.1 Effectiveness. In Figure 3, we show how effectively each algorithm can identify an interaction for a given actual interaction size t_a and input interaction size t_{max}. We show the percentage distribution of the five possible result types per algorithm for all experiments. For this, Figure 3 is split into three parts, (left) t_a is greater than t_{max}, (middle) t_a is equal to t_{max}, and (right) t_a is less than t_{max}. We see that if t_{max} is smaller than t_a, none of the algorithms produces reliable results (< 18% of correctly identified interactions). If t_{max} equals t_a, Inciident correctly identifies 100% of all simulated interactions for an input interaction size t_{max}, while Random correctly identifies ≈ 86%. If t_{max} is greater than t_a, Inciident still identifies 100% of all simulated interactions, while Random only identifies ≈ 45% correctly. If Random returned a subset or super set of the actual interaction, on average a subset is 85% and a super set 210% as big as a simulated interaction. In comparison, for Inciident, subsets are 85% and super set 152% as large as a simulated interaction on average.

Our experiments with real-world faults confirm these findings. If $t_{max} \geq t_a$, Random identifies 6 and Inciident 8 out of 10 interactions correctly. Both algorithms cannot find the fault in the `Linux` system caused by a four-wise interaction within a reasonable time (i.e., 1 hour) and cannot find the fault consisting of two alternative feature interactions, as this violates our first assumption.

4.3.2 Number of Configurations. In Figure 4, we show the median number of tested configurations (y-axis) of Inciident and Random, using different values for t_{max}, for the number of features (x-axis) in each feature model over all experiments, where t_{max} is greater than or equal to t_a. In this plot, we can see two trends. First, the number of configurations increases with a larger value of t_{max}. Second, the number of configurations increases with the number of features of the underlying feature model. For most feature models, even for models with up to 2,000 features, the number of configurations stays below 100. The median number of configurations for t_{max} = 3 is at most 317. In comparison, Random tests on average more configurations than Inciident (≈ 387% for t_{max} = 3).

4.3.3 Computation Time. Figure 5 depicts the median run time (y-axis) of Inciident and Random, using different values for t_{max}, for the number of features (x-axis) in each feature model over all experiments, where t_{max} is greater than or equal to t_a. For $t_{max} = 3$, the median time of Inciident is less than 11s for any feature model, with a maximum time of 364s for the second largest model. On average, Random requires less time than Inciident for $t_{max} = 1$ ($\approx 60\%$) and $t_{max} = 2$ ($\approx 71\%$) and more time for $t_{max} = 3$ ($\approx 121\%$).

4.4 Discussion

Regarding RQ_1, we see that the effectiveness of Inciident is dependent on the chosen value for the input interaction size t_{max}. A value lower than actual interaction size t_a yields unreliable results. However, if $t_{max} \geq t_a$ and our assumptions are satisfied Inciident always identifies the correct interaction, even if $t_{max} > t_a$, which is especially useful in practice, where we often do not know the interaction size beforehand. In contrast, testing arbitrary configurations and using a fixed value for t (i.e., Random) yields substantially worse results for $t_{max} = t_a$ and especially for $t_{max} > t_a$. Thus, when t_{max} is high enough, Inciident is strictly more effective than Random.

Regarding $RQ_{2.1}$, for most feature models, we see a logarithmic correlation between the number of features and median number of tested configurations. In addition, for most cases, the number of tested configurations required by Inciident is substantially lower than with Random. Still, for larger systems, the number of additional test configurations might be to high for practical applications. This may be mitigated by already available samples for the corresponding system, which can be used as input of Inciident. Regarding $RQ_{2.2}$, Inciident requires less than a minute of computing time even for the largest feature model in our experiments, which seems to be a feasible amount of time for most real-world applications.

4.5 Threats to Validity

Internal Validity. In our work, we use pseudo-random numbers to generate configurations and simulate interactions, which means that we may get good or bad results by coincidence. Due to the use of random numbers, measured values and their derived values such as the average value can be inaccurate. We address this problem by performing multiple iterations of experiments for each model, algorithm, input interaction size, and actual interaction size in order to make a more general prediction of the results of our experiments. Furthermore, we generate the interactions for lower interaction sizes by constructing strict subsets of higher interactions. This way, we ensure independence per experiment and reduce the selection bias regarding features contained in simulated interactions.

External Validity. In Section 3, we state our assumptions needed for our concept to work properly. However, in real-world scenarios, these assumptions may not always hold. We cannot guarantee that only one feature interaction causes the fault or that one interaction causes only one fault. In our experiments, we used simulated faults. Nevertheless, as we aim to demonstrate the feasibility of the overall approach, we argue that the limitations implied by the assumption constitute a good trade-off. Our experiments have been performed on 48 real-world feature models, which does not automatically generalize our findings to all feature models. However, when selecting feature models for our experiments, we tried to select a large set of models, which differ in their number of features, constraints, and domains. Many of the selected models are realistic models from industrial applications, which shows that our concept is probably transferable to more industrial models.

5 RELATED WORK

Many sampling algorithms are concerned with generating a representative sample of configurations [12–14, 21–23, 33, 36, 42, 43]. There are some literature surveys [39, 49, 50] considering the concept of combinatorial interaction testing, which is a promising approach to compute a small sample, in which each combination of t features appears in at least one configuration [14, 21, 22, 28, 29, 43]. As in our approach, combinatorial interaction testing focuses on covering certain t-wise feature interactions. In contrast, we need no specific coverage criteria or fully t-wise covered sample when generating configurations. Most sampling techniques only consider the problem space to generate suitable configurations. Including further information from solution space, such as test artifacts or code coverage, has rarely been used and seems to be understudied [50]. Our approach considers only the problem space as well.

None of these sampling algorithms deal with the actual identification process of the fault but stop when detecting failing configurations. Our concept of incremental identification of interactions can be combined with any sampling algorithm. Moreover, we need fault information and configuration testing on demand. There are several algorithms [3, 22, 28] that perform an incremental approach by increasing the coverage with each further configuration incrementally. In comparison, our goal is to incrementally exclude potential interactions to be a candidate for causing failing configurations.

Detecting and identifying feature interactions is challenging [5]. Many variability bugs involve multiple features and are hence feature-interaction bugs [1]. Calder and Miller [11] detect feature interactions by pair-wise analysis. Furthermore, Kuhn et al. [30] state that higher-order interactions are less likely to occur than pair-wise interactions. Currently, we do not know how common interaction faults are in practice or whether current interaction testing techniques are effective at finding the faults [19]. Abal et al. [1] provide a variability bug database with real-world interaction bugs, where 41 faults are caused by single features and 57 faults by feature interactions with an interaction size $t \geq 2$. In our evaluation, we focus on feature interactions with an interaction size from $t = 1$ till $t = 3$, as interactions of more than three features are seldom [1, 20, 30]. However, our concept and tool is applicable to higher-order interactions. Besides, variability bugs may also involve non-locally defined features (i.e., features defined in another subsystem) [1]. Of course, we can never know what interaction size is required to detect all faults in a system [4, 30]. This problem also is inherent to our concept of incremental identification of interactions.

Colbourn and McClary [17] introduce the concept of locating arrays and detecting arrays, which are special types of covering arrays [15, 16] that allow the localization of t-way interactions in a white-box scenario. Compared to that, we consider a black-box scenario independent of the underlying implementation technique. Locating-array algorithms need an entire sample as input, whereas we do not. Martínez et al. [35] present an adaptive approach to locate faults through locating arrays by considering the choice of each new test based on the outcome of all previous tests similar to our approach. When using locating arrays, Aldaco et al. [4] suppose that a maximum number d of existing faulty interactions is given in advance, which seems challenging. In comparison, all of the algorithms considering locating arrays do not consider constraints between features, whereas we are able to consider constraints. It could be possible to involve locating arrays as input for our approach.

There are several machine-learning approaches regarding configuration classification and generation. Temple et al. [48] generate configurations and test them through an oracle to resolve failing configurations by inferring constraints. Siegmund et al. [47] present a tool to analyze several configuration results to get as output which features interact with respect to non-functional properties. Those approaches focus on probabilistic fault prediction for configurations whereas we aim at exact results. In contrast to the probabilistic fault prediction of these machine learning approaches, which cannot guarantee exact results, we employ an analytic approach. Furthermore, when we already have many existing configurations, machine learning can perform a configuration classification easily, but when it comes to generating new configurations machine-learning approaches are far from trivial.

There exist also static-analysis and model-checking approaches addressing the detection of interaction faults [32, 38, 49]. In contrast to our approach, these are white-box approaches. For instance, Meinicke et al. [38] test different configurations, to find type errors, which then provide feature assignments as output like in our approach. These approaches are expensive in terms of time and restrictive, whereas our approach is always applicable for a given test suite.

6 CONCLUSION AND FUTURE WORK

We propose the concept of incremental identification of interactions to tackle the problem of identifying t-wise feature interactions that cause failing configurations. We use a greedy divide-and-conquer strategy that starts with a set of potential feature interactions and incrementally reduces this set by generating and testing additional configurations until it identifies the feature interaction responsible for a given fault. With Inciident, we provide an open-source implementation for our concept. Our evaluation shows that we can reliably identify any interaction that causes a particular fault, given that we choose an input value for the interaction size t that is at least as high as the actual interaction size. Further, we demonstrate that on average Inciident requires 80% less configurations to be tested than Random, while taking less than a minute of computing time even for the largest feature model in our experiments.

In future work, we plan to improve our concept by addressing its current limitations, which is handling faults caused by multiple feature interactions or faults that cannot be detected reliably for every configuration. In addition, we plan to optimize the strategy for configuration generation by including domain knowledge or information from the solution space. This way, we might create more selective configurations and, thereby, reduce testing effort.

ACKNOWLEDGMENTS

This paper is based on the master's thesis by Sabrina Böhm [9]. We like to thank members of the *Institute of Software Engineering and Programming Languages* from the University of Ulm for feedback on presentations about this topic. Moreover, we thank the community of the FOSD Meeting 2023[3] for the enriching discussions and feedback. The source code was developed with help by Jens Meinicke, to which we would like to express our thanks, too. This paper has been partly supported by DFG (German Research Foundation) in project LO 2198/4-1.

REFERENCES

[1] Iago Abal, Jean Melo, Stefan Stănciulescu, Claus Brabrand, Márcio Ribeiro, and Andrzej Wąsowski. 2018. Variability Bugs in Highly Configurable Systems: A Qualitative Analysis. *Trans. on Software Engineering and Methodology (TOSEM)* 26, 3 (Jan. 2018), 10:1–10:34. https://doi.org/10.1145/3149119

[2] Bestoun S. Ahmed, Kamal Z. Zamli, Wasif Afzal, and Miroslav Bures. 2017. Constrained Interaction Testing: A Systematic Literature Study. *IEEE Access* 5 (2017), 25706–25730. https://doi.org/10.1109/ACCESS.2017.2771562

[3] Mustafa Al-Hajjaji, Sebastian Krieter, Thomas Thüm, Malte Lochau, and Gunter Saake. 2016. IncLing: Efficient Product-line Testing Using Incremental Pairwise Sampling. In *Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE)* (Amsterdam, Netherlands). ACM, New York, NY, USA, 144–155. https://doi.org/10.1145/2993236.2993253

[4] Abraham N Aldaco, Charles J Colbourn, and Violet R Syrotiuk. 2015. Locating Arrays: A New Experimental Design for Screening Complex Engineered Systems. *ACM SIGOPS Operating Systems Review* 49, 1 (2015), 31–40. https://doi.org/10.1145/2723872.2723878

[5] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines.* Springer, Berlin, Heidelberg, Germany. https://doi.org/10.1007/978-3-642-37521-7

[6] Eduard Baranov, Axel Legay, and Kuldeep S. Meel. 2020. Baital: An Adaptive Weighted Sampling Approach for Improved t-Wise Coverage. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)* (Virtual Event, USA). ACM, New York, NY, USA, 1114–1126. https://doi.org/10.1145/3368089.3409744

[7] Don Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC).* Springer, Berlin, Heidelberg, Germany, 7–20. https://doi.org/10.1007/11554844_3

[8] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems* 35, 6 (2010), 615–708. https://doi.org/10.1016/j.is.2010.01.001

[9] Sabrina Böhm. 2022. *Identification of Feature Interactions Through Combinatorial Interaction Analysis.* Master's Thesis. University of Ulm, Germany. https://doi.org/10.18725/OPARU-47319

[10] Sabrina Böhm, Sebastian Krieter, Tobias Heß, Thomas Thüm, and Malte Lochau. 2023. *Evaluation Artifact for Incremental Identification of T-Wise Feature Interactions.* https://doi.org/10.5281/zenodo.10292607

[11] Muffy Calder and Alice Miller. 2006. Feature Interaction Detection by Pairwise Analysis of LTL Properties—A Case Study. *Formal Methods in System Design* 28, 3 (2006), 213–261.

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

[13] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. 2006. Coverage and Adequacy in Software Product Line Testing. In *Proc. Int'l Workshop on the Role of Software Architecture for Testing and Analysis (ROSATEA)* (Portland, Maine). ACM, New York, NY, USA, 53–63. https://doi.org/10.1145/1147249.1147257

[14] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. 2007. Interaction Testing of Highly-configurable Systems in the Presence of Constraints. In *Proc. Int'l Symposium on Software Testing and Analysis (ISSTA)* (London, United Kingdom). ACM, New York, NY, USA, 129–139. https://doi.org/10.1145/1273463.1273482

[15] Charles J Colbourn. 2004. Combinatorial Aspects of Covering Arrays. *Le Matematiche* 59, 1, 2 (2004), 125–172.

[16] Charles J Colbourn. 2011. Covering Arrays and Hash Families. https://doi.org/10.3233/978-1-60750-663-8-99

[17] Charles J Colbourn and Daniel W McClary. 2008. Locating and Detecting Arrays for Interaction Faults. *Journal of combinatorial optimization* 15 (2008), 17–48. https://doi.org/10.1007/s10878-007-9082-4

[18] Krzysztof Czarnecki and Ulrich Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications.* ACM/Addison-Wesley, New York, NY, USA.

[19] B.J. Garvin and M.B. Cohen. 2011. Feature Interaction Faults Revisited: An Exploratory Study. In *Proc. Int'l Symposium on Software Reliability Engineering (ISSRE).* 90–99. https://doi.org/10.1109/ISSRE.2011.25

[3]https://fosd23.uni-ulm.de/

[20] Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, and Benoit Baudry. 2019. Test Them All, Is It Worth It? Assessing Configuration Sampling on the JHipster Web Development Stack. *Empirical Software Engineering (EMSE)* 24, 2 (July 2019), 674–717. https://doi.org/10.1007/S10664-018-9635-4

[21] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. 2011. Properties of Realistic Feature Models Make Combinatorial Testing of Product Lines Feasible. In *Proc. Int'l Conf. on Model Driven Engineering Languages and Systems (MODELS)*. Springer, Berlin, Heidelberg, Germany, 638–652. https://doi.org/10.1007/978-3-642-24485-8_47

[22] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. 2012. An Algorithm for Generating T-Wise Covering Arrays from Large Feature Models. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)* (Salvador, Brazil). ACM, New York, NY, USA, 46–55. https://doi.org/10.1145/2362536.2362547

[23] Chang Hwan Peter Kim, Don Batory, and Sarfraz Khurshid. 2011. Reducing Combinatorics in Testing Product Lines. In *Proc. Int'l Conf. on Aspect-Oriented Software Development (AOSD)* (Porto de Galinhas, Brazil). ACM, New York, NY, USA, 57—68. https://doi.org/10.1145/1960275.1960284

[24] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. 2017. Is There a Mismatch Between Real-World Feature Models and Product-Line Research?. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)* (Paderborn, Germany). ACM, New York, NY, USA, 291–302. https://doi.org/10.1145/3106237.3106252

[25] Sebastian Krieter. 2020. Large-Scale T-Wise Interaction Sampling Using YASA. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 29:1–29:4.

[26] Sebastian Krieter. 2022. *Efficient Interactive and Automated Product-Line Configuration.* Ph. D. Dissertation. https://doi.org/10.25673/92625

[27] Sebastian Krieter, Thomas Thüm, Sandro Schulze, Sebastian Ruland, Malte Lochau, Gunter Saake, and Thomas Leich. 2022. *T-Wise Presence Condition Coverage and Sampling for Configurable Systems.* Technical Report arXiv:2205.15180. Cornell University Library. https://doi.org/10.48550/arXiv.2205.15180

[28] Sebastian Krieter, Thomas Thüm, Sandro Schulze, Gunter Saake, and Thomas Leich. 2020. YASA: Yet Another Sampling Algorithm. In *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)* (Magdeburg, Germany). ACM, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.1145/3377024.3377042

[29] D. Richard Kuhn, Raghu N. Kacker, and Yu Lei. 2013. *Introduction to Combinatorial Testing* (1st ed.). Chapman & Hall/CRC, London, UK.

[30] D. Richard Kuhn, Dolores R. Wallace, and Albert M. Gallo Jr. 2004. Software Fault Interactions and Implications for Software Testing. *IEEE Trans. on Software Engineering (TSE)* 30, 6 (2004), 418–421. https://doi.org/10.1109/TSE.2004.24

[31] Daniel Le Berre and Anne Parrain. 2010. The Sat4j Library, Release 2.2. *J. Satisfiability, Boolean Modeling and Computation* 7, 2-3 (2010), 59–64. https://doi.org/10.3233/SAT190075

[32] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. 2013. Scalable Analysis of Variable Software. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)* (Saint Petersburg, Russia). ACM, New York, NY, USA, 81–91. https://doi.org/10.1145/2491411.2491437

[33] Malte Lochau, Ina Schaefer, Jochen Kamischke, and Sascha Lity. 2012. Incremental Model-Based Testing of Delta-Oriented Software Product Lines. In *Proc. Int'l Conf. on Tests and Proofs (TAP)* (Prague, Czech Republic). Springer, Berlin, Heidelberg, Germany, 67–82.

[34] Roberto E. Lopez-Herrejon, Stefan Fischer, Rudolf Ramler, and Aalexander Egyed. 2015. A First Systematic Mapping Study on Combinatorial Interaction Testing for Software Product Lines. In *Proc. Int'l Workshop on Combinatorial Testing (IWCT)*. IEEE, Washington, DC, USA, 1–10. https://doi.org/10.1109/ICSTW.2015.7107435

[35] Conrado Martínez, Lucia Moura, Daniel Panario, and Brett Stevens. 2010. Locating Errors Using ELAs, Covering Arrays, and Adaptive Testing Algorithms. *SIAM Journal on Discrete Mathematics* 23, 4 (2010), 1776–1799. https://doi.org/10.1137/080730706

[36] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *Proc. Int'l Conf. on Software Engineering (ICSE)* (Austin, Texas). ACM, New York, NY, USA, 643–654. https://doi.org/10.1145/2884781.2884793

[37] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering Software Variability with FeatureIDE*. Springer, Berlin, Heidelberg, Germany. https://doi.org/10.1007/978-3-319-61443-4

[38] Jens Meinicke, Chu-Pan Wong, Christian Kästner, Thomas Thüm, and Gunter Saake. 2016. On Essential Configuration Complexity: Measuring Interactions In Highly-Configurable Systems. In *Proc. Int'l Conf. on Automated Software Engineering (ASE)* (Singapore, Singapore). ACM, New York, NY, USA, 483–494. https://doi.org/10.1145/2970276.2970322

[39] Changhai Nie and Hareton Leung. 2011. A Survey of Combinatorial Testing. *ACM Computing Surveys (CSUR)* 43, 2, Article 11 (Feb. 2011), 11:1–11:29 pages. https://doi.org/10.1145/1883612.1883618

[40] Sebastian Oster, Florian Markert, and Philipp Ritter. 2010. Automated Incremental Pairwise Testing of Software Product Lines. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Springer, Berlin, Heidelberg, Germany, 196–210. https://doi.org/10.1007/978-3-642-15579-6_14

[41] Sven Peldszus, Daniel Strüber, and Jan Jürjens. 2018. Model-Based Security Analysis of Feature-Oriented Software Product Lines. In *Proc. Int'l Conf. on Generative Programming and Component Engineering (GPCE)*. 93–106. https://doi.org/10.1145/3393934.3278126

[42] Gilles Perrouin, Sebastian Oster, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. 2012. Pairwise Testing for Software Product Lines: Comparison of Two Approaches. *Software Quality Journal (SQJ)* 20, 3-4 (2012), 605–643. https://doi.org/10.1007/s11219-011-9160-9

[43] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. 2010. Automated and Scalable T-Wise Test Case Generation Strategies for Software Product Lines. In *Proc. Int'l Conf. on Software Testing, Verification and Validation (ICST)*. IEEE, Washington, DC, USA, 459–468. https://doi.org/10.1109/ICST.2010.43

[44] Tobias Pett, Sebastian Krieter, Tobias Runge, Thomas Thüm, Malte Lochau, and Ina Schaefer. 2021. Stability of Product-Line Sampling in Continuous Integration. In *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)* (Krems, Austria). ACM, New York, NY, USA, Article 18, 9 pages. https://doi.org/10.1145/3442391.3442410

[45] Tobias Pett, Thomas Thüm, Tobias Runge, Sebastian Krieter, Malte Lochau, and Ina Schaefer. 2019. Product Sampling for Product Lines: The Scalability Challenge. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)* (Paris, France). ACM, New York, NY, USA, 78–83. https://doi.org/10.1145/3336294.3336322

[46] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wąsowski, and Krzysztof Czarnecki. 2011. Reverse Engineering Feature Models. In *Proc. Int'l Conf. on Software Engineering (ICSE)* (Waikiki, Honolulu, HI, USA). ACM, New York, NY, USA, 461–470. https://doi.org/10.1145/1985793.1985856

[47] Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, Sven Apel, and Gunter Saake. 2012. SPL Conqueror: Toward Optimization of Non-functional Properties in Software Product Lines. *Software Quality Journal (SQJ)* 20, 3-4 (Sept. 2012), 487–517. https://doi.org/10.1007/s11219-011-9152-9

[48] Paul Temple, José A. Galindo, Mathieu Acher, and Jean-Marc Jézéquel. 2016. Using Machine Learning to Infer Constraints for Product Lines. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)* (Beijing, China). ACM, New York, NY, USA, 209–218. https://doi.org/10.1145/2934466.2934472

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

[50] Mahsa Varshosaz, Mustafa Al-Hajjaji, Thomas Thüm, Tobias Runge, Mohammad Reza Mousavi, and Ina Schaefer. 2018. A Classification of Product Sampling for Software Product Lines. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)* (Gothenburg, Sweden). ACM, New York, NY, USA, 1–13. https://doi.org/10.1145/3233027.3233035