Coverage Metrics for T-Wise Feature Interactions

Sabrina Böhm*, Tim Jannik Schmidt†, Sebastian Krieter‡, Tobias Pett§, Thomas Thüm¶, Malte Lochau‖

* University of Ulm
0009-0003-4605-8574

† University of Ulm, Paderborn University
0009-0005-1650-9500

‡ TU Braunschweig, Germany
0000-0001-7077-7091

§ Karlsruhe Institute for Technology (KIT)
0000-0001-7652-6525

¶ TU Braunschweig, Germany
0000-0001-8069-9584

‖ University of Siegen
0000-0002-8404-753X

***Abstract*—Software is typically configurable by means of compile-time or runtime variability. As testing every valid configuration is infeasible, t-wise sampling has been proposed to systematically derive a relevant subset of the configurations for testing to cover interactions among t features. Practitioners started to apply t-wise sampling algorithms, but can often only test samples partially due to restricted resources and compare those partial samples based on their t-wise coverage. However, there is no consensus in the literature on how to compute the t-wise coverage in the literature. We propose the first systematic framework to define coverage metrics for t-wise feature interactions. These metrics differ in the features and feature interactions being considered. We found evidence for at least six different metrics in the literature. In an empirical evaluation, we show that for a partial sample the coverage differs up to 21% and for some metrics only half of the feature interactions need to be covered. As a long-term impact, our work may help to improve the efficiency and effectiveness of both, t-wise sampling and coverage computations.**

***Index Terms*—Configurable Systems, Software Product Lines, Feature Interactions, Combinatorial Interaction Testing, T-Wise Sampling, T-Wise Coverage, Coverage Criteria**

I. Introduction

The goal in managing configurable software is to handle variability, represented by features (i.e., configuration options) [1, 2, 3, 4]. A combination of features can be specified in a configuration that adheres to certain constraints, which can be represented, for instance, in a feature model [3, 4]. A feature model systematically defines all possible combinations of features that result in valid configurations, and a set of these valid configurations is referred to as a sample.

One of the main challenge in highly configurable software engineering is testing the provided variability. As in black-box testing approaches, configurations are derived from the feature model to generate the test input, whereas we abstract from concrete program code. Similar to the concept of statement coverage, where the idea is to cover each statement of a program at least once, we investigate features as well as the interaction of multiple features. In more detail, we are interested in the selection and deselection of multiple features (i.e., for two optional features we have four possible interactions: both selected, one selected, and both deselected).

The goal of t-wise sampling is to achieve a full coverage of all valid combination of t features, such that every combination is contained in at least one configuration [1, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]. To generate such t-wise covering samples, there exist several sampling algorithms aiming at small but representative samples [7, 9, 14, 15, 16, 17, 18, 19]. However, achieving 100% t-wise feature interaction coverage is often not feasible for large models because of the combinatorial explosion [15, 17]. To this end, the interaction size t is typically restricted to a small number (i.e., $t \leq 3$) to ensure applicability of the algorithms [20].

In practice, it is common to test only a limited number of configurations rather than a 100% t-wise covered sample [18], as the testing process can be very time-consuming and expensive. Regarding the testing process, our industrial partners prefer a higher coverage value of a sample for an a priori limited number of configurations. To this end, the percentage of the coverage is important as it serves as an indicator for the fault prediction rate [19].

Most researchers state that they include all combinations of all subsets of t features for the coverage computation, but we found coverage metrics that exclude certain features or feature interactions in the literature [15, 21, 22, 23, 24, 25, 26]. We observed that for samples with a coverage below 100% there exist variations in the definition of the coverage metric that can lead to significantly different results. In practice, our industrial partners wonder which coverage metric to use. In research, this lack of explicit definition means that coverage values may be inconsistent and are not directly comparable across and even within studies, thus, obstructing reproducibility of experimental results. This may also impact certifiability of testing processes in critical domains as requested by recent functional safety standards (e.g., ISO 9126 [27]).

To the best of our knowledge, we propose the first framework for t-wise feature interaction coverage, which can be used to precisely define and compare coverage metrics. We identify several metrics used in the literature and discuss their advantages and disadvantages. We evaluate these metrics in terms of efficiency and effectiveness trade-offs by generating samples for feature models and applying several metrics for the coverage computation. In summary, we contribute the following:

- We provide a systematic overview on filters that can be applied for a t-wise feature interaction coverage metric.

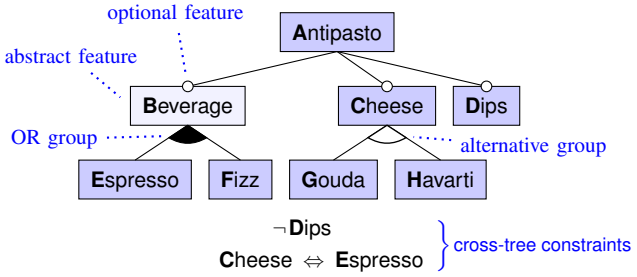

Fig. 1. The example feature model Antipasto

- We examine state-of-the-art t-wise feature interaction coverage metrics we found in the literature.
- We evaluate different coverage metrics on multiple samples for 48 feature models.

II. Background

In the following, we explain foundational principles of configurable software using a motivating example.

A *feature model* represents configuration options (i.e., features) of a configurable software and their interdependencies [1, 2, 3, 4]. The feature model can be represented as a *feature diagram*, where features are represented as nodes and dependencies as edges and additional *cross-tree constraints* [1, 4]. In Figure 1, we depict the feature diagram for our running example, the Antipasto model. Due to brevity, we abbreviate all feature names using their respective first letter.

For simplicity, we use an equivalent logical representation, rather than a graphical one, in our definition of feature models:

Definition 1 (Feature Model)**.** A feature model is a tuple $M = (F, D)$, where $F = \{f_1, \ldots, f_n\}$ is a set of features and $D = \{d_1, \ldots, d_m\}$ is a set of dependencies over the set of Boolean literals $L(F) = \{\neg f_1, \ldots, \neg f_n, f_1, \ldots, f_n\}$, where $f_i \in F, 1 \leq i \leq n$. We denote the *universe of all feature models* with $\mathcal{M}$.

A feature model describes the allowed combinations of features, and thus describes whether a configuration is valid. We define a *configuration* as a set of literals.

Definition 2 (Configuration)**.** Let $M = (F, D)$ be a feature model, a configuration $c = \{l_1, \ldots, l_k\} \subseteq L(F)$ with $\{l, \neg l\} \nsubseteq c$ 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 it is excluded, 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*.

We define $C(M)$ to be the set of all valid configurations of a feature model M. We call any subset of valid configurations S a *sample* (i.e., $S \subseteq C(M)$).

TABLE I
Table of valid pair-wise feature interactions for the feature model Antipasto

	A	¬A	B	¬B	C	¬C	D	¬D	E	¬E	F	¬F	G	¬G
¬H	✓		✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓
H	✓		✓		✓			✓	✓		✓	✓		✓
¬G	✓		✓	✓	✓	✓		✓	✓	✓	✓	✓		
G	✓		✓		✓			✓	✓		✓	✓		
¬F	✓		✓	✓	✓	✓		✓	✓	✓				
F	✓		✓		✓	✓		✓	✓	✓				
¬E	✓		✓	✓		✓		✓						
E	✓		✓		✓			✓						
¬D	✓		✓	✓	✓	✓								
D														
¬C	✓		✓	✓										
C	✓		✓											
¬B	✓													
B	✓													

The set of dependencies induces some semantic properties that may hold for certain features. A feature is *abstract*, if it is not associated with any solution space artifact, but is only used for structuring the model (e.g., feature B) [28]. In contrast, we call all features that are not abstract *concrete* (e.g., feature C). A feature is called *core*, if it is selected in all valid configurations (e.g., feature A) [1]. Analogous, a feature is called *dead*, if it is deselected in all valid configurations (e.g., feature D) [1]. For instance, we can select feature A and deselect all other features, resulting in a valid configuration $c_1 = \{\mathsf{A}, \neg\mathsf{B}, \neg\mathsf{C}, \neg\mathsf{D}, \neg\mathsf{E}, \neg\mathsf{F}, \neg\mathsf{G}, \neg\mathsf{H}\}$, because all child features of A are optional. We define a *t-wise feature interaction* as a valid partial configuration of size t.

Definition 3 (Feature Interaction)**.** Let $M = (F, D)$ be a feature model, a t-wise feature interaction $i = \{l_1, \ldots, l_t\} \subseteq L(F)$ is a valid (partial) configuration of size t (i.e., $|i| = t$). We denote the universe of all *sets* of interactions by $\mathcal{I}$.

We call an interaction with interaction size $t = 1$ a one-wise interaction (e.g., $\{\mathsf{C}\}$), with $t = 2$ a pair-wise interaction (e.g., $\{\mathsf{C}, \neg\mathsf{D}\}$) and so forth. Exemplary, in Table I, we list all pair-wise interactions, where a check mark denotes that the feature interaction is valid. For instance, we see that every pair-wise interaction containing the feature A is valid, except the interaction $\{\mathsf{A}, \mathsf{D}\}$ due to feature-model constraints. Note that for each pair of features there are four possible feature interactions, (i.e., both features are selected, exactly one is selected, or neither feature is selected) [1, 13].

As feature interactions can lead to faults, it is crucial to test for any undesirable behavior [29, 30, 31, 32, 33]. In product-line testing, the goal is to have a preferably small, yet sufficiently effective sample, which can identify as many faults as possible. With t-wise sampling algorithms, we are able to generate a sample called t-wise sample that contains every valid t-wise feature interaction in at least one of its configurations [6, 7, 14, 15, 16, 19, 34, 35, 36, 37]. However, for practical applications, the efficiency of a sample must also be taken into account. A popular reference for the efficiency is simply its size (i.e., the number of contained configurations), as this often correlates with the testing effort. While reducing the number of configurations from a t-wise sample increases its efficiency, it may also obstruct its effectiveness (i.e., its

ability to detect faulty interactions).

Coverage is frequently used as a metric for effectiveness [20], but there is no common definition. Thus, we are interested in the relative number of *t-wise feature interaction coverage* of a sample, because it serves as an indicator for its fault detection rate [19]. Let us consider a sample s with $c_1 = \{\mathsf{A}, \neg\mathsf{B}, \neg\mathsf{C}, \neg\mathsf{D}, \neg\mathsf{E}, \neg\mathsf{F}, \neg\mathsf{G}, \neg\mathsf{H}\}$ and a second configuration $c_2 = \{\mathsf{A}, \mathsf{B}, \mathsf{C}, \neg\mathsf{D}, \mathsf{E}, \neg\mathsf{F}, \mathsf{G}, \neg\mathsf{H}\}$ (i.e., $s = \{c_1, c_2\}$) derived from the feature model in Figure 1, for which we aim to compute the t-wise coverage. For an interaction size $t = 2$, Johansen et al. [13] would compute all valid pair-wise interactions contained in s and divide it by all possible pair-wise interactions of the model as listed in Table I. Therefore, the pair-wise coverage of s would be $\approx 69\%$, which means that for this concrete sample s 50 out of 73 possible pair-wise interactions are covered (i.e., $\frac{50}{73}$). However, if Hentze et al. [23] would compute the coverage for s, the relative number of the pair-wise coverage would be $\approx 58\%$ (i.e., $\frac{19}{33}$). This difference occurs because researchers consider different coverage metrics to compute t-wise coverage. In the next section, we investigate different coverage metrics to gain more insights into the different ways to compute t-wise feature interaction coverage.

III. Metrics For T-Wise Coverage

In this section, we present several metrics that can be used for computing t-wise feature interaction coverage. We base our metrics on a general way of determining a set of feature interactions. We subsequently filter this set with a list of predefined filters, which are motivated from t-wise feature interaction coverage metrics we found in the literature. In Section III-A, we propose a definition of t-wise feature interaction coverage. In Section III-B, we present filters that can be applied for coverage metrics. In Section III-C, we discuss performance and complexity aspects of the filters. In Section III-D, we describe the combination of filters. In Section III-E, we present the coverage metrics found in the literature. In the remainder of the paper, we use the term *t-wise coverage* to refer to *t-wise feature interaction coverage*, if not stated otherwise.

A. T-Wise Feature Interaction Coverage

A canonical method to compute t-wise coverage is to build the set of all valid t-wise feature interactions (i.e., valid partial configurations), as done by Cohen et al. [11]. This rules out all invalid feature combinations according to the underlying feature model constraints. For instance, the literal $\neg\mathsf{A}$ is never part of a valid interaction, as can be seen in the respective column, which is highlighted in gray in Table I. To derive the set of all interactions for a given feature model and a value for t, we define the set of all valid interactions as a function $\circledast$ receiving a feature model M and a value $t \in \mathbb{N}$ as input:

$$\begin{aligned} \circledast \quad &: \quad \mathcal{M} \times \mathbb{N} \to \mathcal{I} \\ \circledast(M,t) \;&\mapsto\; \{i \subseteq L(F) : \\ &\qquad |i| = t,\ \exists c \in C(M) : i \subseteq c\} \end{aligned}$$

Given the set of all t-wise interactions, as listed for $t = 2$ in Table I, the default metric for t-wise coverage $Cov_{default}$ can be computed by dividing the number of interactions contained in a sample by the total number of interactions [24]:

$$\begin{aligned} Cov_{default} \quad &: \quad \mathbb{N} \times \mathcal{M} \times \mathcal{C} \to [0,1]_{\mathbb{R}} \\ (t,M,S) \;&\mapsto\; \begin{cases} 1, & \text{if } \mid \circledast(M,t) \mid = 0 \\ \dfrac{\mid \{i \in \circledast(M,t) : \exists c \in S : i \in c\} \mid}{\mid \circledast(M,t) \mid}, & \text{else} \end{cases} \end{aligned}$$

In the literature, the definition of this coverage metric is often refined in various ways to exclude certain features or interactions from the computation [23, 15, 37, 22, 17, 24, 25, 26]. To account for all emerging different coverage metrics, we use the following modified definition for *t-wise coverage*:

$$\begin{aligned} Cov \quad &: \quad (\mathcal{I} \to \mathcal{I}) \times \mathbb{N} \times \mathcal{M} \times \mathcal{C} \to [0,1]_{\mathbb{R}} \\ (\blacktriangledown,t,M,S) \;&\mapsto\; \begin{cases} 1, & \text{if } \mid \blacktriangledown(\circledast(M,t)) \mid = 0 \\ \dfrac{\mid \{i \in \blacktriangledown(\circledast(M,t)) : \exists c \in S : i \in c\} \mid}{\mid \blacktriangledown(\circledast(M,t)) \mid}, & \\ & \text{else} \end{cases} \end{aligned}$$

This definition allows for a given filter function $\blacktriangledown : \mathcal{I} \to \mathcal{I}$ to exclude certain interactions from being considered in the computation. Thus, any specific coverage metric can be described entirely by specifying the value for t and the filter function $\blacktriangledown$. A mathematically accurate representation of the filter function would be $\blacktriangledown' : (\mathcal{I}, \mathcal{M}) \to (\mathcal{I}, \mathcal{M})$. However, to improve readability and because the feature model M is fixed in our context, we omit the second parameter. In the following, we formalize multiple filter functions, motivated by coverage metrics encountered in existing literature.

B. Filtering Feature Interactions

In the following, we present filters excluding subsets of features (i.e., Section III-B2 to Section III-B5), merging features (i.e., Section III-B6 and Section III-B7), and excluding subsets of interactions (i.e., Section III-B8 and Section III-B9).

*1) No filter (**Default**):* The default coverage metric $Cov_{default}$, described above, is a special case of our coverage definition when using the identity function as a filter:

$$\blacktriangledown_{default}(I) = I$$

*2) Excluding core features (**CF**):* When computing coverage, we can think of excluding certain features for the computation of the t-wise coverage. Core features are part of every complete valid configuration and can not be deselected (e.g., feature A in Table I) [1]. Excluding all t-wise interactions containing core features reduces the set of t-wise interactions that must be considered for the coverage computation. Applying a metric excluding core features for coverage computation is done by several approaches described in the literature [23, 15, 37, 17, 24, 25, 26]. Formally, we define the corresponding filter $\blacktriangledown_{CF}$, which filters all interactions containing core features, as follows:

$$\blacktriangledown_{CF}(I) = \{i \in I :\ i \cap core(M) = \emptyset\}$$

TABLE II
TABLE OF VALID PAIR-WISE FEATURE INTERACTIONS WITHOUT CORE FEATURES FOR THE FEATURE MODEL ANTIPASTO

	A	¬A	B	¬B	C	¬C	D	¬D	E	¬E	F	¬F	G	¬G
¬H			✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓
H			✓		✓			✓	✓		✓	✓		✓
¬G			✓	✓	✓	✓		✓	✓	✓	✓	✓		
G			✓		✓			✓	✓		✓	✓		
¬F			✓	✓	✓	✓		✓	✓	✓				
F			✓		✓	✓		✓	✓	✓				
¬E			✓	✓		✓		✓						
E			✓		✓			✓						
¬D			✓	✓	✓	✓								
D														
¬C			✓	✓										
C			✓											
¬B														
B														

where $core(M)$ is the set of all core features of a feature model M (i.e., the positive literals contained in every complete valid configuration).

Considering our running example model Antipasto, we exclude all interactions containing core features (i.e., every interaction containing the feature A) by applying $\blacktriangledown_{\text{CF}}$, as can be seen highlighted in blue in Table II.

*3) Excluding dead features (**DF**):* Similar to core features, dead features are never contained in a complete valid configuration and can, therefore, never be selected (e.g., feature D in Table I) [1]. Analogous to interactions containing core features, combinations of the negative literal of a dead feature and other features may be ignored when computing coverage. We define the corresponding filter $\blacktriangledown_{DF}$, which filters all interactions containing dead features, as follows:

$$\blacktriangledown_{DF}(I) = \{i \in I : i \cap dead(M) = \emptyset\}$$

where $dead(M)$ is the set of all dead features of the feature model M (i.e., all negative literals contained in every complete valid configuration). Applying $\blacktriangledown_{\text{DF}}$ in the model Antipasto, excludes all interactions containing dead features (i.e., every interaction containing the deselected feature ¬D).

We consider core and dead features separately because depending on the specific use case, we might want to exclude only one of these for the coverage computation. The computation of core and dead features is known as the backbone of a propositional formula, used as a typical preprocessing technique in the area of SAT-Solving [38]. In fact, all coverage metrics we found in literature that exclude core features also exclude dead features [23, 15, 37, 17, 24, 25, 26].

*4) Excluding abstract features (**AF**):* It is possible to exclude features not owning an implementation like abstract features [28]. A feature is called *abstract*, if it is not associated with any solution space artifact, but is used for structuring the model (e.g., feature B in Figure 1) [28]. In contrast, all features that are not abstract are called *concrete* (e.g., feature C). To this end, Kowal et al. [22] exclude abstract features in the coverage computation. We define the filter $\blacktriangledown_{AF}$, which filters the interactions containing abstract features:

$$\blacktriangledown_{AF}(I) = \{i \in I : i \cap abstract(M) = \emptyset\}$$

where $abstract(M)$ is the set of all abstract features (containing positive and negative literals) of the feature model M.

In contrast to the core and dead feature exclusion, we do not exclude tuples that would be covered anyway, but we are explicitly omitting tuples containing certain features from the coverage metric. Similar to excluding core and dead features, excluding any interaction containing an abstract feature may save computational effort and memory consumption for calculating the coverage. As abstract features have no influence on the derived software product, it can be reasonable to omit them [22].

*5) Excluding arbitrary subsets of features (**ArbF**):* Considering the three filter functions above, we can generalize the principle to a filter function that excludes an arbitrary set of features. A reason for this can be additional domain knowledge about the configurable system. For instance, in a given testing scenario (e.g., regression testing or testing of subsystems) only certain types of features or only a certain subtree of the feature model may be relevant. For example, Hentze et al. [23] only consider features in their approach that are mapped to software artifacts of the system and, thus, exclude hardware features. Moreover, Kowal et al. [22] propose to exclude features from t-wise interaction testing that are unlikely to interact. We define the filter $\blacktriangledown_{ArbF}$, which filters the interactions containing literals from a given set:

$$\blacktriangledown_{ArbF}(I) = \{i \in I : i \cap l_{filter} = \emptyset\}$$

where l_{filter} is an arbitrary set of literals.

*6) Merging atomic feature sets (**AFS**):* Features are part of an *atomic feature set*, if they always occur together in every valid configuration (e.g., features C and E in Figure 1) [3, 23, 39, 40, 41]. More precisely, if any feature within the atomic feature set is (de-)selected, its dependencies imply that all other features in the set must also be (de-)selected. In a partial configuration, any feature in an atomic set may be replaced by any other feature in the same set without changing the configuration space described by the partial configuration. Thus, for each atomic feature set, we may select one feature as a representative for the other features.

In Figure 1, we see that the features C and E must always be both selected or both deselected in a valid configuration due to the cross-tree constraint C ⇔ E. We visualize the atomic feature set exclusion in Table III highlighted in blue, where the representative feature C is highlighted in orange. Furthermore, there are two specific atomic feature sets: the set of core features and the set of dead features. We define the filter $\blacktriangledown_{AFS}$, which filters the interactions that contain any feature from an atomic feature set that is not a representative:

$$\blacktriangledown_{\text{AFS}}(I) = \{i \in I : \forall a \in atomicFeatureSets(M): \\ |a| > 1 \Rightarrow i \cap (a \setminus \{\neg\Pi_1(a), \Pi_1(a)\}) = \emptyset\}$$

where $atomicFeatureSets(M)$ is the set of all atomic feature sets of the feature model M and $\Pi_1(a)$ is a function that deterministically selects one element from a set a.

The atomic feature set analysis is used as a preprocessing technique in sampling to reduce the size of the underlying

TABLE III
TABLE OF VALID PAIR-WISE FEATURE INTERACTIONS WITH MERGING ATOMIC FEATURE SETS FOR THE FEATURE MODEL

	A	¬A	B	¬B	C	¬C	D	¬D	E	¬E	F	¬F	G	¬G
¬H	✓		✓	✓	✓	✓		✓			✓	✓	✓	✓
H	✓		✓		✓			✓			✓	✓		✓
¬G	✓		✓	✓	✓	✓		✓			✓	✓		
G	✓		✓		✓			✓			✓	✓		
¬F	✓		✓	✓	✓	✓		✓						
F	✓		✓		✓	✓		✓						
¬E														
E														
¬D	✓		✓	✓	✓	✓								
D														
¬C	✓		✓	✓										
C	✓		✓											
¬B	✓													
B	✓													

model and, furthermore, to analyze models in automated reasoning [39, 3, 40, 41]. Computing all atomic feature sets of a feature model results in a partition of the feature set into multiple disjoint subsets. Given such a partition, this effectively allows to exclude interactions that contain features that are not a representative in an atomic set, as these interactions are automatically considered by the remaining ones.

*7) Merging atomic literal sets (**ALS**):* Atomic literal sets are a generalization of atomic feature sets. An atomic literal set contains all literals (positive and negative) that must be included in a valid configuration if any one of the literals from the set is included in the configuration. We define the filter $\blacktriangledown_{ALS}$, which filters the interactions that contain any literal that is not a representative from an atomic literal set:

$$\blacktriangledown_{ALS}(I) = \{i \in I : \forall a \in atomicLiteralSets(M): |a| > 1 \Rightarrow i \cap (a \setminus \{\Pi_1(a)\}) = \emptyset\}$$

where $atomicLiteralSets(M)$ is the set of all atomic literal sets of the feature model M and $\Pi_1(a)$ is a function that deterministically selects one element from a set a. While the set of core features and the set of dead features constitute separate atomic feature sets (i.e., $core(M) \in atomicFeatureSets(M)$ and $dead(M) \in atomicFeatureSets(M)$), together they form an atomic literal set (i.e., $core(M) \cup dead(M) \in atomicLiteralSets(M)$).

*8) Excluding parent-child interactions (**PCI**):* Dependencies between features are often defined by the hierarchical structure of feature diagrams. The selection of a feature always implies the selection of its parent in the feature diagram. Siegmund et al. [21] therefore disregard interactions containing a feature and its parent in their approach, as these interactions are already implied whenever the feature is included in a configuration. Thus, the idea is to omit parent-child relations when computing coverage. In Figure 1, we have certain features that are always selected if their children are selected (e.g., feature A and feature C are parents of feature H). We formalize the filter $\blacktriangledown_{PCI}$ introduced by Siegmund et al. [21], which filters all interactions containing a feature and its direct parent simultaneously:

$$\blacktriangledown_{PCI}(I) = \{i \in I : \forall pc \in parentChild(M): pci \not\subseteq i\}$$

where $parentChild(M)$ is the set of all parent-child literal sets for the feature model M.

*9) Excluding arbitrary subsets of interactions (**ArbI**):* Similar to the exclusion of arbitrary features, we may exclude arbitrary interactions. This can be useful, if we only want to consider specific sets of interactions (e.g., changed interactions in regression testing) or to exclude interactions for which domain knowledge shows that there is no interaction in concrete software artifacts. For instance, the sampling algorithm *YASA* [15] allows to specify an arbitrary set of t-wise interactions for creating a t-wise sample. We define the filter $\blacktriangledown_{ArbI}$, which excludes an arbitrary set of interactions:

$$\blacktriangledown_{ArbI}(I) = I \setminus I_{filter}$$

where I_{filter} is an arbitrary set of interactions.

C. Computational Effort and Performance of Filters

Some of the analyses required for applying the filters can be very time-consuming and resource-intensive. However, considering fewer t-wise interactions when computing coverage may speed up the computation, increase performance and may reduce memory consumption. It is often suggested that core and dead analysis can be done in a reasonable amount of time [3, 70, 71]. Abstract features are marked in the model file and can, therefore, be used almost directly. Likewise, parent-child interactions can be easily extracted from the propositional formula of the model (i.e., by analyzing the implications in the conjunctive normal form of the model).

When considering all t-wise interactions between features in an atomic feature set and other features, there is no advantage in considering these interactions because they are already tested if one of the features of the atomic feature set is considered. Hentze et al. [23] reduce the considered interactions by the atomic feature set analysis, enabling the sampling of a model that could not be sampled before. However, the computation of atomic feature sets can be very time-consuming and even infeasible for large models [3, 39, 40], which may outweigh any potentially gained performance benefits. The trade-off between the computational effort of the analyses and the reduction of the considered interactions may depend on the feature model.

D. Combining Filters Into Coverage Metrics

Above, we defined multiple filter functions, each of which can be used to derive a metric using the formula in Section III. Is is also possible to combine filters, which is used in the literature [23, 15, 37, 22, 17, 24, 25, 26]. Similar to a pipe-and-filter architecture, multiple filters can be applied sequentially, with each filterw using the output of the previous one. For instance, excluding core and dead features at the same time is often applied in the literature [23, 15, 37, 17, 24, 25, 26]. We can define a corresponding filter by composition, as follows:

$$\blacktriangledown_{CF-DF} = \blacktriangledown_{CF} \circ \blacktriangledown_{DF}$$

TABLE IV
METRICS FOUND IN THE LITERATURE: ✓ FILTER APPLIED, ⊘ FILTER NOT APPLIED, ? UNSURE.
METRIC INFORMATION: * OBTAINED FROM TOOL, ** ACQUIRED BY CONTACTING AUTHORS, *** INDIRECT FROM OTHER REFERENCES.

Metric	CF	DF	AF	ArbF	AFS	ALS	PCI	ArbI	Paper
M1	✓	✓	⊘	✓	✓	⊘	⊘	⊘	Hentze et al. [23]**
M2	✓	✓	⊘	✓	⊘	⊘	⊘	✓	Krieter et al. [15]*, Krieter et al. [37]**
M3	⊘	⊘	✓	✓	⊘	⊘	⊘	⊘	Kowal et al. [22]
M4	✓	✓	⊘	⊘	⊘	⊘	⊘	⊘	Al-Hajjaji et al. [17]**, Pett et al. [24]**, Heß et al. [25], Pett et al. [26]**, Böhm et al. [42]
M5	⊘	⊘	⊘	⊘	⊘	⊘	✓	⊘	Siegmund et al. [21]*
M6	⊘	⊘	⊘	⊘	⊘	⊘	⊘	⊘	Johansen et al. [13]*, Johansen et al. [14]*, Johansen et al. [43], Ferreira et al. [44]***, Oh et al. [45]*, Baranov et al. [16]*
	?/⊘	?/⊘	?/⊘	?/⊘	?/⊘	?/⊘	?/⊘	?/⊘	Cohen et al. [11], Lei et al. [46], Cohen et al. [47], Lei et al. [48], Perrouin et al. [7], Oster et al. [34], Garvin et al. [49], Shi et al. [50], Haslinger et al. [51], Henard et al. [52], Liebig et al. [53], Marijan et al. [54], Marijan et al. [54], Lopez-Herrejon et al. [55], Medeiros et al. [6], Ferreira et al. [44], Ruland et al. [56], Luthmann et al. [57], Pett et al. [58], Luo et al. [59], Ferreira et al. [60], Pett et al. [61], Baranov and Legay [62], Tavassoli et al. [63], Xiang et al. [64], Baranov et al. [65], Bombarda et al. [66], Abolfazli et al. [67], Luo et al. [68], Luo et al. [69]

For a visual demonstration, we created a website[1], where interested readers can try out combinations of filters to see which interactions are ruled out for our example Antipasto.

E. State-Of-The-Art T-Wise Coverage Metrics

To identify existing coverage metrics, we conducted a literature study. We started our research with the survey "A Classification of Product Sampling for Software Product Lines" by Varshosaz et al. [20] and filtered the linked papers by the tags "*t*-wise" and "pair-wise". From those papers, we include all those which consider the absence and presence of features in *t*-wise coverage (i.e., for two optional features four possible interactions: both selected, one selected, and both deselected). To search for newer publications, we also looked into papers that cited this survey. We did not perform a systematic literature study but instead aimed to get some evidence of used *t*-wise coverage in the research.

In Table IV, we show an overview of found *t*-wise coverage metrics. Each row stands for one metric, we found in the literature. We state which filters are applied (i.e., ✓) and not applied (i.e., ⊘) for the coverage metric. By reading each paper, we attempted to identify their definition of coverage. If an explicit definition of the coverage metric was not provided, we examined the tool, if available. In addition, we reached out to some authors via email to inquire about their tool and their definition of *t*-wise coverage. They sometimes shared further papers with us, which we have analyzed as well.

The majority of researchers state that they consider all combinations of t features when computing coverage. By looking into their tools, we found further information for some papers. For instance, Siegmund et al. [21] explicitly define that they include abstract features, whereas Kowal et al. [22] exclude abstract features. Johansen et al. [13] exclude core and dead feature when they sample with their algorithm *ICPL*, but when computing the coverage they include core and dead features. Considering the work of Hentze et al. [23] (i.e., metric M1), we can define a corresponding filter composition $\blacktriangledown_{CF-DF-ArbF-AFS} = \blacktriangledown_{CF} \circ \blacktriangledown_{DF} \circ \blacktriangledown_{ArbF} \circ \blacktriangledown_{AFS}$.

[1]https://t-wise-coverage.github.io/

The descriptions of the coverage computation in the papers are imprecise, thus we were not able to answer our questions about the coverage definition for every paper. We suggest that the majority of the papers listed in the last row of Table IV use the metric M6, which does not apply any filters to exclude certain features or interactions, as they did not mention any exclusion. However, due to our uncertainty, we marked them with a question mark. This strengthens our thoughts to investigate the coverage computation in more detail and raise awareness of differences in used coverage metrics.

IV. EVALUATION

In this section, we present the results of evaluating the different metrics of *t*-wise feature interaction coverage. We used 48 feature models, mostly from real-world scenarios, by generating *t*-wise samples and by applying different coverage metrics for different sizes of partial samples (i.e., subsets of the 100% *t*-wise sample). We compare the metrics applying different filters presented in Section III, where $\blacktriangledown_{default}$ is considered as our ground truth. We evaluate all combinations of filters, however, we decided to show results for metrics that cover a variety of combinations of filters, among others found in the literature (cf. Section III-E), namely metrics using the

- default filter (i.e., $\blacktriangledown_{default}$, called default metric) (**M6**)
- abstract filter (i.e., $\blacktriangledown_{AF}$) (**M3** omitting arbitrary features)
- atomic literal set filter (i.e., $\blacktriangledown_{ALS}$)
- parent child interactions filter (i.e., $\blacktriangledown_{PCI}$) (**M5**)
- core and dead filters (i.e., $\blacktriangledown_{CF-DF} = \blacktriangledown_{CF} \circ \blacktriangledown_{DF}$) (**M4** and **M2** without arbitrary features and interactions)
- filters core, dead, and atomic literal set $\blacktriangledown_{CF-DF-ALS} = \blacktriangledown_{CF} \circ \blacktriangledown_{DF} \circ \blacktriangledown_{ALS}$ (**M1** without arbitrary features)
- filters core, dead, atomic literal set, abstract, and parent child interactions $\blacktriangledown_{CF-Df-AF-ALS-PCI} = \blacktriangledown_{CF} \circ \blacktriangledown_{DF} \circ \blacktriangledown_{AF} \circ \blacktriangledown_{ALS} \circ \blacktriangledown_{PCI}$

We provide an open-source implementation of the filters used for our coverage metrics written in Java based on the FeatJAR library.[2] We provide a replication package[3] for our

[2]https://github.com/FeatureIDE/FeatJAR
[3]https://doi.org/10.5281/zenodo.14717243

evaluation and the data generated by our experiments.

The focus of our research questions is to investigate the impact of the choice of the metric (i.e., RQ_1), the number of interactions that must be considered (i.e., RQ_{2a} and RQ_{2b}) and the required time (i.e., RQ_{3a} and RQ_{3b}) to compute the coverage. In our evaluation, we address the following research questions.

- RQ_1 Does the choice of a metric impact the coverage value?
- RQ_{2a} How does the choice of a metric affect the number of filtered interactions?
- RQ_{2b} How does on the number of features affect the number of filtered interactions?
- RQ_{3a} How does the choice of a metric affect the computation time?
- RQ_{3b} How does the number of features affect the computation time?

In RQ_1, we investigate how the values of the coverage differ when applying different coverage metrics. In RQ_{2a}, we inspect the number of interactions that have to be considered for the computation of the coverage. More precisely, we analyze how the set of considered interactions decreases compared to the default metric. This is crucial because, for some larger feature models, feature-model analyses can only be applied if the number of considered interactions is kept sufficiently low [23]. In RQ_{2b}, we investigate the reduction of the filtered interactions in relation to the size of the feature model. With RQ_{3a}, we consider the time to compute the coverage for a given partial sample including the time to compute the analyses (e.g., atomic feature set analysis) that are required for the applied filters. Our timeout value is 60 min. We did not measure the time for generating the samples.

A. Experiment Setup

To evaluate our concept, we perform several experiments to investigate the values of the coverage for different metrics.

For the experiments, we selected 48 feature models from various real-world sources and domains, including systems software, communication, finance, gaming, and e-commerce [58, 61, 72, 73, 74]. These models have been used for, among other things, the evaluation of sampling algorithms. These models encompass a broad range of features, from 9 to 2513 and a number of clauses between 13 and 15,692. To ensure variety in feature-model sizes, we selected small to medium-sized models from examples provided by the tool FeatureIDE [73]. We also included more complex, real-world feature models [58, 72, 74], as well as six models of varying sizes from the eCos system, documented by Knüppel et al. [72]. Additionally, we used feature models from real-world Kconfig systems provided by Pett et al. [61], choosing both the earliest and latest versions for each system.

In this setup, the independent variables are the underlying feature model, the sample size, and the used metric. Our dependent variables are the value of the coverage, the number of considered interactions, and the run time that is required for each execution of one experiment. As a limit for the interaction size, we consider $t \in \{1,2,3\}$, as higher values would result in prohibitively long computation times for larger models.

We use the algorithm *YASA* [15] to compute one-wise, pair-wise, and three-wise samples. First, we compute all analyses required by the filters for each feature model. Second, we generate five t-wise samples for each model and each interaction size t. For each of these samples, we generate five new samples with the same configurations, but in a different random order. From these 25 samples, we create partial samples using the first 20%, 40%, 60%, 80%, and 100% of configurations. Note that, this ensures that each partial sample is a subset of a larger (partial) sample, which improves comparability among samples later on. For each partial sample, we compute the coverage for all metrics using all combinations of our filters. We measure the time required for each filter and its associated analyses, then we add up the total time for all the filters used, along with the time required for the coverage computation itself. When presenting the results of our evaluation, we always refer to median values, if not stated otherwise.

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 22.04.4 LTS; *Java*: openjdk 17.0.12; *JVM memory*: 128 GB.

B. Results

1) RQ_1: We find that, for some metrics, the coverage value differs up to 21%. In Figure 2, we show the difference in the relative coverage for a given feature model and a coverage metric compared to our ground truth metric $\blacktriangledown_{default}$. We split the results into three parts, interaction size $t = 1$, $t = 2$, and $t = 3$. We see that the coverage increases or decreases compared to the default metric, where the general trend tends to show a decrease for all t. If we apply the filter $\blacktriangledown_{CF-Df-AF-ALS-PCI}$, we see a coverage reduction up to $\approx 21\%$ for an interaction size $t = 3$. Note that we cannot compute three-wise samples for the largest 11 models due to the timeout.

In Figure 3, we depict the increase of the pair-wise coverage per partial sample size. Exemplary, we show the results for two models, a smaller one and a larger one (i.e., all plots can be found in our replication package). We see that for the small model *axTLS*, the coverage using the default metric increases faster than using the metric with the filter $\blacktriangledown_{CF-Df-AF-ALS-PCI}$. For the model *am31_sim*, the coverage using the metric with the filter $\blacktriangledown_{CF-Df-AF-ALS-PCI}$ is higher than the default metric until the samples size reaches $\approx 65\%$.

2) RQ_{2a}: Some metrics have a substantially stronger impact on the number of filtered interactions than others. In Figure 4, we show reduction of the pair-wise interactions considered for the coverage computation compared to the default metric for all models and partial sample sizes (cf. replication package for all interaction sizes). We see all metrics applying the filter $\blacktriangledown_{ALS}$ reduce the number of considered interactions significantly. Using the filter $\blacktriangledown_{CF-Df-AF-ALS-PCI}$, we must consider $\approx 46\%$ of all pair-wise interactions compared to default, while for $t = 3$ the interaction reduction is $\approx 43\%$.

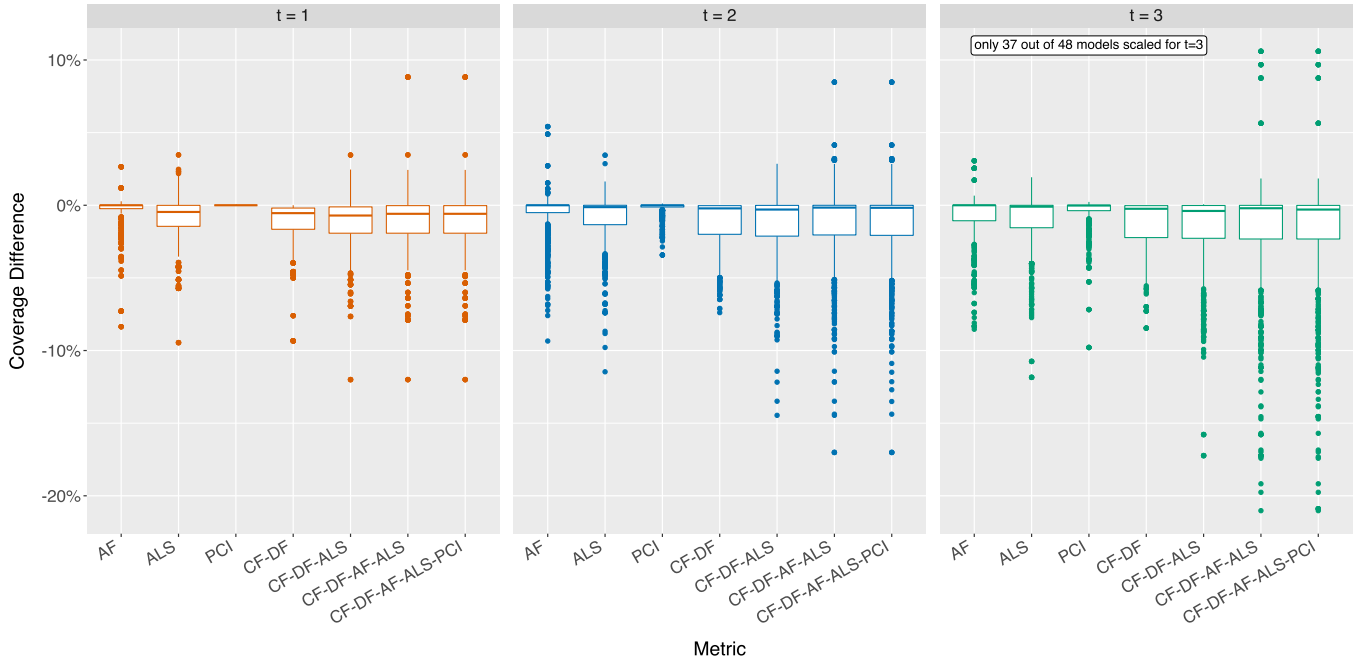

Fig. 2. Coverage Relative to the Default Metric

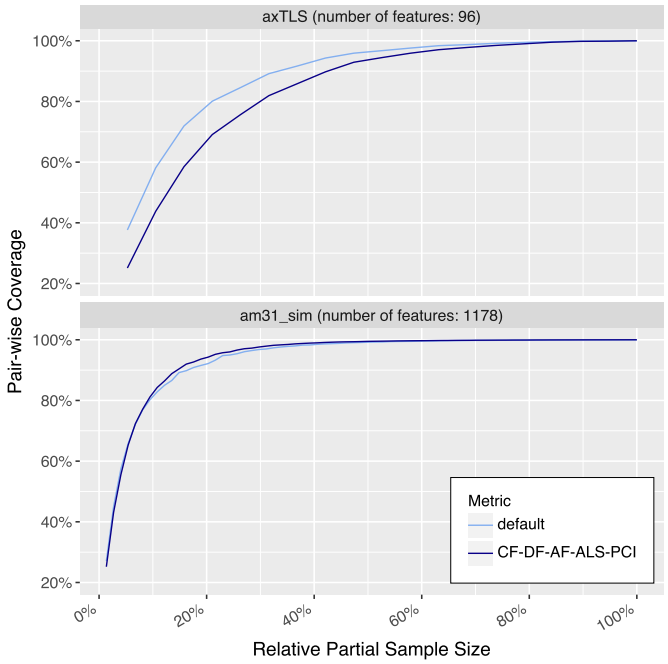

Fig. 3. Pair-wise Coverage per Partial Sample Size for *axTLS* and *am31_sim*

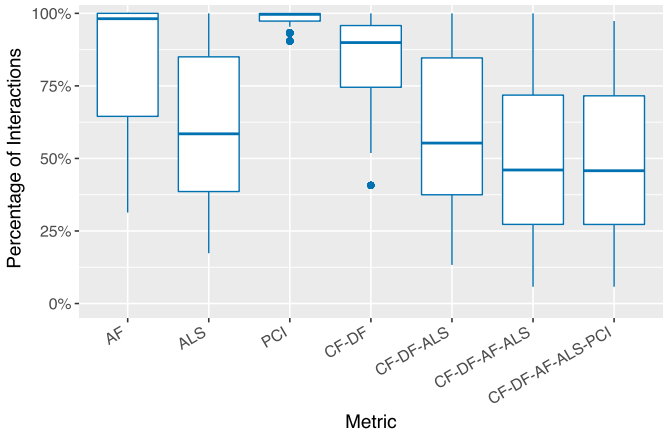

Fig. 4. Pair-wise Interactions per Metric Relative to the Default Metric

3) RQ_{2b}: The number of interactions seems linear proportional to the number of features for all metrics. In Figure 5, we show the absolute number of considered interactions with regard to the feature-model size. Note that the data presented in Figure 5 is plotted on a logarithmic scale for both the x and y axes. We depict the results for the default metric and the metric using the filter $\blacktriangledown_{CF-Df-AF-ALS-PCI}$ for all interaction sizes. We see that for smaller models, the number of considered interactions varies more between the two metrics than for the larger models. For the model *busybox* (number of features: 905), we must consider ≈93% of the pair-wise interactions by using the metric with the filter compared to the default metric (i.e., 1,499,997 out of 1,613,373 pair-wise interactions). For the model *calculate* (number of features: 9), we must consider ≈18% of the pair-wise interactions by using the metric with the filter compared to the default metric (i.e., 22 out of 120 pair-wise interactions).

4) RQ_{3a}: Some metrics require a substantially longer computation time than others. In Figure 6, we present the computation time required for each metric displaying the results

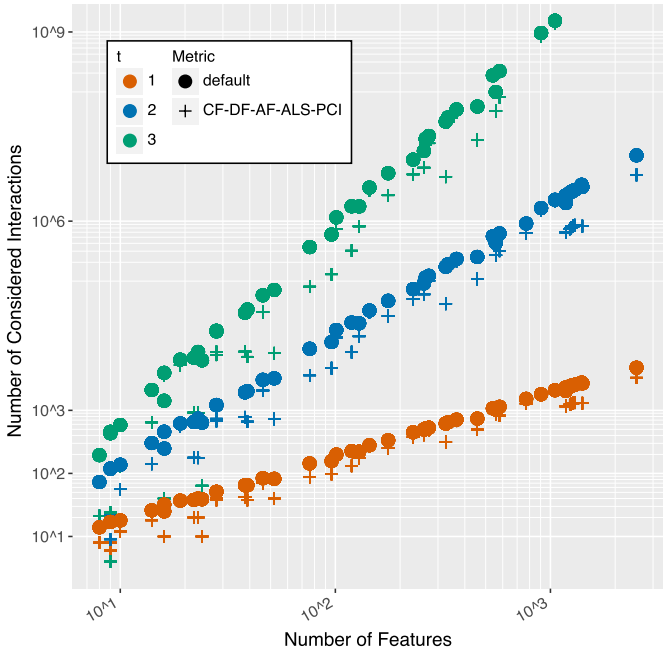

Fig. 5. Number of Considered Interactions per Metric per Feature Model

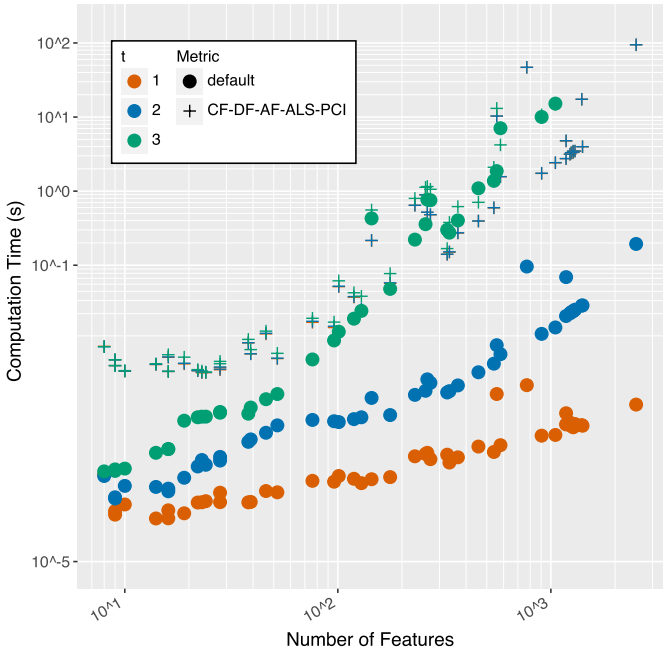

Fig. 7. Computation Time per Metric per Feature Model

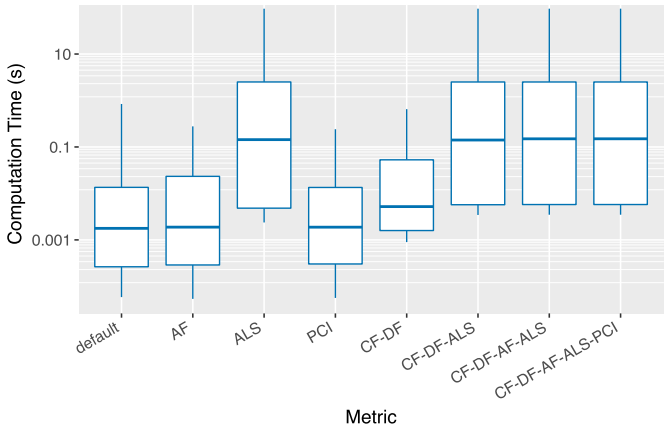

Fig. 6. Computation Time per Metric for an Interaction Size $t = 2$

for an interaction size $t = 2$. Note that the presented data is plotted on a logarithmic scale. We see that all metrics applying the filter $\blacktriangledown_{ALS}$ require more time than the other metrics. The required time for computing the coverage with the filter $\blacktriangledown_{CF-DF-AF-ALS-PCI}$ is ≈0.15s.

5) RQ_{3b}: The computation time seems to be approximately linear proportional to the number of features for all metrics. In Figure 7, we show the computation time dependent on the number of features from the underlying model. We depict the values for the default metric and metric with the filter $\blacktriangledown_{CF-Df-AF-ALS-PCI}$ for all interaction sizes. Note that, some of the data points for the metric using the filter $\blacktriangledown_{CF-Df-AF-ALS-PCI}$ for $t = 1$, $t = 2$ are hidden behind the data points for $t = 3$. We see that the time for most models is higher for the metric with the filter $\blacktriangledown_{CF-Df-AF-ALS-PCI}$ than the default metric. However, for the model *WaterlooGenerated* (number of features: 580) the computation time is faster using the metric with the filter (i.e., ≈4.1s) than the default metric (i.e., ≈7.6s) for $t = 3$.

C. Discussion

Regarding RQ_1, we see that coverage differs up to 21% depending on the choice of the metric. For several filters, the impact depends of course on the number of core, dead, and abstract features as well as the number of atomic sets of the model. The coverage value can either increase or decrease compared to the default metric, indicating that applying a filter does not inherently lead to a reduction in coverage.

In RQ_{2a}, we found that when applying the filter $\blacktriangledown_{CF-DF-AF-ALS-PCI}$ for smaller models, we reduce the considered interactions to 46% compared to the default metric. As expected, applying filters reduces the number of considered interactions for the coverage computation. Considering RQ_{2b}, we see that the number of filtered interactions depends on the model size. For smaller models, we may exclude approximately half of the interactions, whereas for larger models, a smaller percentage of interactions is excluded. This can help reduce memory usage when sampling or computing coverage.

Regarding RQ_{3a}, the coverage computation time is similarly independent of the interaction size t for the metric with the filter $\blacktriangledown_{CF-DF-AF-ALS-PCI}$, because the feature-model analyses require the most computation time. For an interaction size $t = 3$, using the metric with the filter $\blacktriangledown_{CF-DF-AF-ALS-PCI}$ results in faster computation times than using the default metric (note that the largest 11 models are excluded here). Although model analyses require their time, the reduced number of considered interactions allows for faster coverage computation.

The considered filters can be divided into filters that exclude interactions that are already covered by other interactions and filters that exclude features to be considered, for instance, by including domain knowledge. We recommend to reduce the set of considered interactions by excluding interactions that are already covered by others to improve performance. However, including domain knowledge to exclude arbitrary features and interactions may decrease the effectiveness of the sample.

D. Threats to Validity

1) Internal Validity: Our results are influenced by the randomness of the selection of the partial samples. To increase internal validity, we perform multiple iterations for each model, partial sample size, interaction size, and metric in order to make a more general prediction of the results of our experiments. Moreover, we report median values and illustrate the results using box plots.

Our implementation may have bugs that may bias the results. However, we rely on an existing sampling algorithm (*YASA* [15]) and most feature-model analyses were already available through the FeatJAR library[2], except for the analyses concerning the filters $\blacktriangledown_{AF}$, $\blacktriangledown_{ALS}$, and $\blacktriangledown_{PCI}$. To ensure replicability, the seed for the random number generator in the sampling step is used as a control variable.

2) External Validity: Our findings may not automatically be generalized to other feature models. However, our experiments have been performed on 48 feature models, where we tried to select a large set of models, which differ in their number of features, constraints, and domains. They also differ in the number of core, dead, and abstract features as well as all other anomalies, which effects the computation of the filters and, thus, the coverage metrics. Most of the models are realistic models from industrial applications, which shows that our findings are probably transferable to more industrial models.

V. Related Work

In software testing, there exist popular coverage metrics as statement coverage measuring whether every block of (optional) code is selected at least once [75, 76, 77]. Additionally, partial and full path coverage aim to ensure not only that certain paths in the software are tested, but that all possible execution paths are considered [75]. Similar to our approach, Rojas et al. [78] combine multiple coverage criteria by suggesting that optimizing several criteria by combining multiple metrics is feasible without increasing computational costs. These approaches are white-box approaches with access to program code, whereas combinatorial testing is a black-box approach [19]. In contrast, we abstract from concrete software artifacts and instead investigate a problem space point of view.

Existing work in combinatorial testing has explored various coverage criteria. Grindal et al. [79] provide a survey that discusses the properties of combination strategies and the coverage criteria used in combinatorial testing. Williams and Probert [80] first present the formal definition of t-wise coverage. Moreover, there exist performance-oriented approaches considering combinations of t features that must be selected at least once [81, 82, 83, 84, 85]. In contrast, we focus on combinations of t features including the deselection of features, which is the consensus in product-line testing [8, 86, 87, 88, 89, 90]. Besides t-wise sampling, there are other strategies as uniform random sampling [45] and distance-based sampling [81], which also investigate the reached coverage of their samples. In Section III-E, we provide related work that follows this approach when considering coverage. However, no existing work has systematically compared different t-wise interaction coverage metrics, as we propose in this paper.

VI. Conclusion and Future Work

In product-line testing, there is no consensus on how to compute the t-wise feature interaction coverage in the literature, which distorts the comparability across the literature. By excluding several features or interactions in the coverage computation, the coverage value can differ significantly. We propose the first systematic framework to define coverage metrics for t-wise feature interactions by providing a compilation of filters that can be applied when computing coverage. In an empirical evaluation, we investigate the impact of the choice of the metric by evaluating 48 feature models. We found that for a partial sample the coverage differs up to 21% depending on the coverage metric. Besides, the set of considered interactions is reduced up to half of all valid t-wise feature interactions.

We recommend to use a metric, where core and dead features are excluded, because this analysis is applicable in reasonable time even for large models [3, 70, 71]. In addition, we suggest to apply the atomic literal set filter to reduce the interactions that must be considered. In product-line testing, we suggest to exclude arbitrary features and interactions by using domain knowledge, to enable applicability of analysis even for larger models. As a long-term impact, our work may help to improve the efficiency of t-wise sampling algorithms by reducing the set of considered interactions to speed up the computation. Furthermore, we may help to improve the efficiency of coverage computations. Concluding, we propose that for all studies the used coverage metric should be explicitly defined to ensure comparability across studies.

As future work, we suggest to investigate how the coverage changes if the feature model changes. Moreover, we propose to adapt sampling algorithms based on the presented coverage metrics. As we restrict our current work on boolean feature models, it is also possible to adapt the coverage metrics to cardinality-based feature models. Our current implementation enables an easy extension by new filters. Therefore, it is possible to filter, for instance, further equivalent feature interactions.

Acknowledgment

We like to thank the community of the FOSD Meeting 2024[4] for the enriching discussions and feedback. This paper has been partially supported by DFG (German Research Foundation) in project LO 2198/4-1 and SFB 1608 – 501798263.

[4]https://set.win.tue.nl/event/fosd-meeting-2024/

References

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

[2] K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. ACM/Addison-Wesley, 2000.

[3] D. Benavides, S. Segura, and A. Ruiz-Cortés, "Automated Analysis of Feature Models 20 Years Later: A Literature Review," *Information Systems*, vol. 35, no. 6, pp. 615–708, 2010.

[4] D. Batory, "Feature Models, Grammars, and Propositional Formulas," in *SPLC*. Springer, 2005, pp. 7–20.

[5] M. Lochau, I. Schaefer, J. Kamischke, and S. Lity, "Incremental Model-Based Testing of Delta-oriented Software Product Lines," in *TAP*. Springer, 2012, pp. 67–82.

[6] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel, "A Comparison of 10 Sampling Algorithms for Configurable Systems," in *ICSE*. ACM, 2016, pp. 643–654.

[7] 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*. IEEE, Apr. 2010, pp. 459–468.

[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*, vol. 56, no. 10, pp. 1183–1199, 2014.

[9] M. B. Cohen, M. B. Dwyer, and J. Shi, "Interaction Testing of Highly-configurable Systems in the Presence of Constraints," in *ISSTA*. ACM, 2007, pp. 129–139.

[10] C. H. P. Kim, D. Batory, and S. Khurshid, "Reducing Combinatorics in Testing Product Lines," in *AOSD*. ACM, 2011, pp. 57–68.

[11] M. B. Cohen, M. B. Dwyer, and J. Shi, "Coverage and Adequacy in Software Product Line Testing," in *ROSATEA*. ACM, 2006, pp. 53–63.

[12] G. Perrouin, S. Oster, S. Sen, J. Klein, B. Baudry, and Y. Le Traon, "Pairwise Testing for Software Product Lines: Comparison of Two Approaches," *SQJ*, vol. 20, no. 3-4, pp. 605–643, 2012.

[13] M. F. Johansen, Ø. Haugen, and F. Fleurey, "Properties of Realistic Feature Models Make Combinatorial Testing of Product Lines Feasible," in *MODELS*. Springer, 2011, pp. 638–652.

[14] ——, "An Algorithm for Generating T-Wise Covering Arrays From Large Feature Models," in *SPLC*. ACM, 2012, pp. 46–55.

[15] S. Krieter, T. Thüm, S. Schulze, G. Saake, and T. Leich, "YASA: Yet Another Sampling Algorithm," in *VaMoS*. ACM, Feb. 2020.

[16] E. Baranov, A. Legay, and K. S. Meel, "Baital: An Adaptive Weighted Sampling Approach for Improved t-Wise Coverage," in *ESEC/FSE*. ACM, 2020, pp. 1114–1126. [Online]. Available: https://doi.org/10.1145/3368089.3409744

[17] M. Al-Hajjaji, S. Krieter, T. Thüm, M. Lochau, and G. Saake, "IncLing: Efficient Product-line Testing Using Incremental Pairwise Sampling," in *GPCE*. ACM, Oct. 2016, pp. 144–155.

[18] A. Halin, A. Nuttinck, M. Acher, X. Devroey, G. Perrouin, and B. Baudry, "Test Them All, Is It Worth It? Assessing Configuration Sampling on the JHipster Web Development Stack," *EMSE*, vol. 24, no. 2, pp. 674–717, Jul. 2019.

[19] D. R. Kuhn, R. N. Kacker, and Y. Lei, *Introduction to Combinatorial Testing*, 1st ed. Chapman & Hall/CRC, 2013.

[20] M. Varshosaz, M. Al-Hajjaji, T. Thüm, T. Runge, M. R. Mousavi, and I. Schaefer, "A Classification of Product Sampling for Software Product Lines," in *SPLC*. ACM, Sep. 2018, pp. 1–13.

[21] N. Siegmund, M. Rosenmüller, M. Kuhlemann, C. Kästner, S. Apel, and G. Saake, "SPL Conqueror: Toward Optimization of Non-functional Properties in Software Product Lines," *SQJ*, vol. 20, no. 3-4, pp. 487–517, Sep. 2012.

[22] M. Kowal, S. Schulze, and I. Schaefer, "Towards Efficient SPL Testing by Variant Reduction," in *VariComp*. ACM, 2013, pp. 1–6.

[23] M. Hentze, C. Sundermann, T. Thüm, and I. Schaefer, "Quantifying the Variability Mismatch Between Problem and Solution Space," in *MODELS*. IEEE, Oct. 2022, pp. 322–333.

[24] T. Pett, T. Heß, S. Krieter, T. Thüm, and I. Schaefer, "Continuous T-Wise Coverage," in *SPLC*. ACM, Aug. 2023, pp. 87–98.

[25] T. Heß, T. J. Schmidt, L. Ostheimer, S. Krieter, and T. Thüm, "UnWise: High T-Wise Coverage From Uniform Sampling," in *VaMoS*. ACM, Feb. 2024, pp. 37–45.

[26] T. Pett, S. Krieter, T. Thüm, and I. Schaefer, "MulTi-Wise Sampling: Trading Uniform T-Wise Feature Interaction Coverage for Smaller Samples," in *SPLC*. ACM, 2024, p. 47–53.

[27] International Standard Organization (ISO), "International Standard ISO/IEC 9126, Information technology - Product Quality - Part1: Quality Model," https://cdn.standards.iteh.ai/samples/22749/d293dbe1fea54b3e853dfc5a07549390/ISO-IEC-9126-1-2001.pdf, 2001.

[28] T. Thüm, C. Kästner, S. Erdweg, and N. Siegmund, "Abstract Features in Feature Modeling," in *SPLC*. IEEE, Aug. 2011, pp. 191–200.

[29] J. Meinicke, C.-P. Wong, C. Kästner, T. Thüm, and G. Saake, "On Essential Configuration Complexity: Measuring Interactions in Highly-Configurable Systems," in *ASE*. ACM, Sep. 2016, pp. 483–494.

[30] I. Abal, J. Melo, S. Stănciulescu, C. Brabrand, M. Ribeiro, and A. Wąsowski, "Variability Bugs in Highly Configurable Systems: A Qualitative Analysis," *TOSEM*, vol. 26, no. 3, pp. 10:1–10:34, Jan. 2018.

[31] B. Garvin and M. Cohen, "Feature Interaction Faults Revisited: An Exploratory Study," in *ISSRE*, Nov. 2011, pp. 90–99.

[32] D. R. Kuhn, D. R. Wallace, and A. M. Gallo Jr., "Software Fault Interactions and Implications for Software Testing," *TSE*, vol. 30, no. 6, pp. 418–421, 2004.

[33] C. Nie and H. Leung, "A Survey of Combinatorial Testing," *CSUR*, vol. 43, no. 2, pp. 11:1–11:29, Feb. 2011.

[34] S. Oster, F. Markert, and P. Ritter, "Automated Incremental Pairwise Testing of Software Product Lines," in *SPLC*. Springer, 2010, pp. 196–210.

[35] S. Krieter, "Efficient Interactive and Automated Product-Line Configuration," Ph.D. dissertation, Jun. 2022.

[36] ——, "Large-Scale T-Wise Interaction Sampling Using YASA," in *SPLC*. ACM, Oct. 2020, pp. 29:1–29:4.

[37] S. Krieter, T. Thüm, S. Schulze, S. Ruland, M. Lochau, G. Saake, and T. Leich, "T-Wise Presence Condition Coverage and Sampling for Configurable Systems," Cornell University Library, Tech. Rep. arXiv:2205.15180, May 2022.

[38] M. Janota, I. Lynce, and J. Marques-Silva, "Algorithms for Computing Backbones of Propositional Formulae," *Ai Communications*, vol. 28, no. 2, pp. 161–177, 2015.

[39] W. Zhang, H. Zhao, and H. Mei, "A Propositional Logic-Based Method for Verification of Feature Models," in *ICFEM*. Springer, 2004, pp. 115–130.

[40] S. Segura, "Automated Analysis of Feature Models Using Atomic Sets," in *SPLC*, vol. 2. IEEE, Sep. 2008, pp. 201–207.

[41] R. Schröter, S. Krieter, T. Thüm, F. Benduhn, and G. Saake, "Compositional Analyses of Highly-Configurable Systems With Feature-Model Interfaces," in *SE*, J. Jürjens and K. Schneider, Eds. Gesellschaft für Informatik, Feb. 2017, pp. 129–130.

[42] S. Böhm, S. Krieter, T. Heß, T. Thüm, and M. Lochau, "Incremental Identification of T-Wise Feature Interactions," in *VaMoS*. ACM, Feb. 2024, pp. 27–36.

[43] M. F. Johansen, Ø. Haugen, F. Fleurey, A. G. Eldegard, and T. Syversen, "Generating Better Partial Covering Arrays by Modeling Weights on Sub-Product Lines," in *MODELS*. Springer, 2012, pp. 269–284.

[44] T. N. Ferreira, J. A. P. Lima, A. Strickler, J. N. Kuk, S. R. Vergilio, and A. Pozo, "Hyper-Heuristic Based Product Selection for Software Product Line Testing," *CIM*, vol. 12, no. 2, pp. 34–45, May 2017.

[45] J. Oh, P. Gazzillo, and D. Batory, "t-wise Coverage by Uniform Sampling," in *SPLC*. ACM, Sep. 2019, pp. 84–87.

[46] Y. Lei, R. N. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "IPOG: A General Strategy for T-Way Software Testing," in *ECBS*. IEEE, 2007, pp. 549–556.

[47] M. B. Cohen, M. B. Dwyer, and J. Shi, "Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints: A Greedy Approach," *TSE*, vol. 34, no. 5, pp. 633–650, 2008.

[48] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "IPOG/IPOG-D: Ipog/Ipog-D: Efficient Test Generation for Multi-Way Combinatorial Testing," *STVR*, vol. 18, no. 3, pp. 125–148, 2008.

[49] B. J. Garvin, M. B. Cohen, and M. B. Dwyer, "Evaluating Improvements to a Meta-Heuristic Search for Constrained Interaction Testing," *EMSE*, vol. 16, no. 1, pp. 61–102, 2011.

[50] J. Shi, M. B. Cohen, and M. B. Dwyer, "Integration Testing of Software Product Lines Using Compositional Symbolic Execution," in *FASE*. Springer, Mar. 2012, pp. 270–284.

[51] E. N. Haslinger, R. E. Lopez-Herrejon, and A. Egyed, "Using Feature Model Knowledge to Speed Up the Generation of Covering Arrays," in *VaMoS*. ACM, 2013, pp. 16:1–16:6.

[52] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. L. Traon, "Multi-Objective Test Generation for Software Product Lines," in *SPLC*. ACM, 2013, pp. 62–71.

[53] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer, "Scalable Analysis of Variable Software," in *ESEC/FSE*. ACM, Aug. 2013, pp. 81–91.

[54] D. Marijan, A. Gotlieb, S. Sen, and A. Hervieu, "Practical Pairwise Testing for Software Product Lines," in *SPLC*. ACM, 2013, pp. 227–235.

[55] R. E. Lopez-Herrejon, J. Ferrer, F. Chicano, A. Egyed, and E. Alba, "Comparative Analysis of Classical Multi-Objective Evolutionary Algorithms and Seeding Strategies for Pairwise Testing of Software Product Lines," in *CEC*. IEEE, 2014, pp. 387–396.

[56] S. Ruland, L. Luthmann, J. Bürdek, S. Lity, T. Thüm, M. Lochau, and M. Ribeiro, "Measuring Effectiveness of Sample-Based Product-Line Testing," in *GPCE*. ACM, Nov. 2018, pp. 119–133.

[57] L. Luthmann, T. Gerecht, and M. Lochau, "Sampling Strategies for Product Lines With Unbounded Parametric Real-Time Constraints," *STTT*, vol. 21, no. 6, pp. 613–633, 2019.

[58] T. Pett, T. Thüm, T. Runge, S. Krieter, M. Lochau, and I. Schaefer, "Product Sampling for Product Lines: The Scalability Challenge," in *SPLC*. ACM, Sep. 2019, pp. 78–83.

[59] C. Luo, B. Sun, B. Qiao, J. Chen, H. Zhang, J. Lin, Q. Lin, and D. Zhang, "Ls-Sampling: An Effective Local Search Based Sampling Approach for Achieving High T-Wise Coverage," in *ESEC/FSE*. ACM, 2021, p. 1081–1092.

[60] F. Ferreira, G. Vale, J. P. Diniz, and E. Figueiredo, "Evaluating T-Wise Testing Strategies in a Community-Wide Dataset of Configurable Software Systems," *JSS*, vol. 179, p. 110990, 2021.

[61] T. Pett, S. Krieter, T. Runge, T. Thüm, M. Lochau, and I. Schaefer, "Stability of Product-Line Sampling in Continuous Integration," in *VaMoS*. ACM, Feb. 2021.

[62] E. Baranov and A. Legay, "Baital: An Adaptive Weighted Sampling Platform for Configurable Systems," in *SPLC*. ACM, 2022, pp. 46–49.

[63] S. Tavassoli, C. D. N. Damasceno, R. Khosravi, and M. R. Mousavi, "Adaptive Behavioral Model Learning for Software Product Lines," in *SPLC*, 2022, pp. 142–153.

[64] Y. Xiang, H. Huang, M. Li, S. Li, and X. Yang, "Looking For Novelty in Search-Based Software Product Line Testing," *TSE*, vol. 48, no. 7, pp. 2317–2338, 2022.

[65] E. Baranov, S. Chakraborty, A. Legay, K. S. Meel, and V. N. Variyam, "A Scalable T-Wise Coverage Estimator," in *ICSE*. ACM, 2022, p. 36–47. [Online]. Available: https://doi.org/10.1145/3510003.3510218

[66] A. Bombarda, S. Bonfanti, and A. Gargantini, "On the Reuse of Existing Configurations for Testing Evolving Feature Models," in *SPLC*. ACM, 2023, p. 67–76.

[67] A. Abolfazli, J. Spiegelberg, G. Palmer, and A. Anand, "A Deep Reinforcement Learning Approach to Configuration Sampling Problem," in *ICDM*, 2023, pp. 1–10.

[68] C. Luo, S. Lyu, Q. Zhao, W. Wu, H. Zhang, and C. Hu, "Beyond Pairwise Testing: Advancing 3-wise Combinatorial Interaction Testing for Highly Configurable Systems," in *ISSTA*. ACM, 2024, p. 641–653.

[69] C. Luo, J. Song, Q. Zhao, B. Sun, J. Chen, H. Zhang, J. Lin, and C. Hu, "Solving the t-wise Coverage Maximum Problem via Effective and Efficient Local Search-based Sampling," *TOSEM*, Aug. 2024, just Accepted. [Online]. Available: https://doi.org/10.1145/3688836

[70] M. Mendonça, A. Wąsowski, and K. Czarnecki, "SAT-Based Analysis of Feature Models Is Easy," in *SPLC*. Software Engineering Institute, 2009, pp. 231–240.

[71] E. Kuiter, T. Heß, C. Sundermann, S. Krieter, T. Thüm, and G. Saake, "How Easy Is SAT-Based Analysis of a Feature Model?" in *VaMoS*. ACM, Feb. 2024, pp. 149–151.

[72] A. Knüppel, T. Thüm, S. Mennicke, J. Meinicke, and I. Schaefer, "Is There a Mismatch Between Real-World Feature Models and Product-Line Research?" in *SE*, M. Tichy, E. Bodden, M. Kuhrmann, S. Wagner, and J. Steghöfer, Eds. Gesellschaft für Informatik, Mar. 2018, pp. 53–54.

[73] J. Meinicke, T. Thüm, R. Schröter, F. Benduhn, T. Leich, and G. Saake, *Mastering Software Variability With FeatureIDE*. Springer, 2017.

[74] S. She, R. Lotufo, T. Berger, A. Wąsowski, and K. Czarnecki, "Reverse Engineering Feature Models," in *ICSE*. ACM, 2011, pp. 461–470.

[75] H. Zhu, P. A. Hall, and J. H. May, "Software Unit Test Coverage and Adequacy," *CSUR*, vol. 29, no. 4, pp. 366–427, 1997.

[76] R. Tartler, D. Lohmann, C. Dietrich, C. Egger, and J. Sincero, "Configuration Coverage in the Analysis of Large-Scale System Software," *OSR*, vol. 45, no. 3, pp. 10–14, Jan. 2012.

[77] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat, "Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem," in *EuroSys*. ACM, 2011, pp. 47–60.

[78] J. M. Rojas, J. Campos, M. Vivanti, G. Fraser, and A. Arcuri, "Combining Multiple Coverage Criteria in Search-Based Unit Test Generation," in *Search-Based Software Engineering: 7th International Symposium*. Springer, Sep. 2015, pp. 93–108.

[79] M. Grindal, J. Offutt, and S. F. Andler, "Combination Testing Strategies: A Survey," *STVR*, vol. 15, no. 3, pp. 167–199, 2005.

[80] A. W. Williams and R. L. Probert, "A Measure for Component Interaction Test Coverage," in *AICCSA*. IEEE, 2001, pp. 304–311.

[81] C. Kaltenecker, A. Grebhahn, N. Siegmund, J. Guo, and S. Apel, "Distance-Based Sampling of Software Configuration Spaces," in *ICSE*. IEEE, 2019, pp. 1084–1094.

[82] J. Guo, D. Yang, N. Siegmund, S. Apel, A. Sarkar, P. Valov, K. Czarnecki, A. Wąsowski, and H. Yu, "Data-Efficient Performance Learning for Configurable Systems," *EMSE*, vol. 23, no. 3, pp. 1826–1867, Jun. 2018.

[83] A. Sarkar, J. Guo, N. Siegmund, S. Apel, and K. Czarnecki, "Cost-Efficient Sampling for Performance Prediction of Configurable Systems," in *ASE*. IEEE, 2015, pp. 342–352.

[84] N. Siegmund, S. S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake, "Predicting Performance via Automated Feature-Interaction Detection," in *ICSE*. IEEE, 2012, pp. 167–177.

[85] N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner, "Performance-Influence Models for Highly Configurable Systems," in *ESEC/FSE*. ACM, Aug. 2015, pp. 284–294.

[86] P. A. Da Mota Silveira Neto, I. D. Carmo Machado, J. D. McGregor, E. S. De Almeida, and S. R. De Lemos Meira, "A Systematic Mapping Study of Software Product Lines Testing," *IST*, vol. 53, no. 5, pp. 407–423, May 2011.

[87] J. Lee, S. Kang, and D. Lee, "A Survey on Software Product Line Testing," in *SPLC*. ACM, 2012, pp. 31–40.

[88] E. Engström and P. Runeson, "Software Product Line Testing - A Systematic Mapping Study," *IST*, vol. 53, pp. 2–13, Jan. 2011.

[89] B. P. Lamancha, M. P. Usaola, and M. P. Velthius, "Software Product Line Testing," *A Systematic Review. ICSOFT (1)*, pp. 23–30, 2009.

[90] J. Lee, S. Kang, and P. Jung, "Test Coverage Criteria for Software Product Line Testing: Systematic Literature Review," *IST*, vol. 122, p. 106272, 2020.