T-Wise Presence Condition Coverage and Sampling for Configurable Systems

Sebastian Krieter[1], Thomas Thüm[1], Sandro Schulze[2], Sebastian Ruland[3], Malte Lochau[4], Gunter Saake[2], and Thomas Leich[5]

[1]University of Ulm, Ulm, Germany
[2]University of Magdeburg, Magdeburg, Germany
[3]TU Darmstadt, Darmstadt, Germany
[4]University of Siegen, Siegen, Germany
[5]Harz University of Applied Sciences, Wernigerode, Germany

Abstract

Sampling techniques, such as t-wise interaction sampling are used to enable efficient testing for configurable systems. This is achieved by generating a small yet representative sample of configurations for a system, which circumvents testing the entire solution space. However, by design, most recent approaches for t-wise interaction sampling only consider combinations of configuration options from a configurable system's variability model and do not take into account their mapping onto the solution space, thus potentially leaving critical implementation artifacts untested. Tartler et al. address this problem by considering presence conditions of implementation artifacts rather than pure configuration options, but do not consider the possible interactions between these artifacts. In this paper, we introduce t-wise presence condition coverage, which extends the approach of Tartler et al. by using presence conditions extracted from the code as basis to cover t-wise interactions. This ensures that all t-wise interactions of implementation artifacts are included in the sample and that the chance of detecting combinations of faulty configuration options is increased. We evaluate our approach in terms of testing efficiency and testing effectiveness by comparing the approach to existing t-wise interaction sampling techniques. We show that t-wise presence condition sampling is able to produce mostly smaller samples compared to t-wise interaction sampling, while guaranteeing a t-wise presence condition coverage of 100%.

1 Introduction

Testing is an important task in software engineering to detect faults and to check intended behavior [1, 2]. However, exhaustive testing may be impossible and binds resources that could be used in other phases of development. This is especially an issue when testing highly-configurable systems, such as Software Product Lines (SPLs). A product of an SPL can be derived from a configuration, which consist of a list of configuration options (i.e.,features) that can be set either to *true* or *false*. The entirety of all possible configurations for an SPL is called the *configuration space*, which typically grows exponentially with the number of features [3, 4].

A straight-forward testing strategy for SPLs is product-based testing. For this purpose, a set of products is derived from a set of different configurations and then test cases are run on each selected product respectively [5]. As testing every possible configuration in a product-based manner is usually not feasible due to an enormous configuration space, sampling strategies

have been defined to generate a small yet representative set of configurations to test (i.e., a configuration sample). One such sampling strategy is *t-wise interaction sampling*, which aims to generate a preferably small sample that covers all possible interactions of features of degree t [6, 7]. Using t-wise interaction sampling, developers can ensure that all interactions of at most t features (e.g., all selected, none selected, only one selected, etc.) are indeed contained in at least one configuration in the generated sample. T-wise interaction sampling has shown to be a feasible trade-off between testing effectiveness and testing efficiency, as for small values of t (i.e., $t \in \{2, 3\}$) it usually returns a relatively small sample, while also achieving reliable test results [7–9].

A property of t-wise interaction sampling is that it works purely on the problem space of an SPL. Thus, it is a black-box approach that does not take into account the mapping between features and actual implementation artifacts, such as source code, models, and test cases. This can lead to a number of issues. First, a traditional t-wise sampling algorithm may create samples that contain configurations that generate similar products, which can decrease overall testing efficiency. Second, when choosing the parameter for t the degree of faulty interactions is not known to the developers. Thus, developers are incentivized to choose a low value for t in order to keep the testing effort feasible. However, some faults may require a feature interaction of a high degree to be included in a configuration, and thus may not be included at all in a sample, which potentially decreases testing effectiveness.

Due to the black-box nature of t-wise interaction sampling, it may not reveal certain faults resulting from feature interaction beyond t or requires too many configurations to even reach a certain code coverage or fault-detection rate. To overcome this limitation, Tartler et al. [10] propose *statement coverage*, a white-box approach that considers *presence conditions* (i.e., the selection of features for which an artifact is included in a product) to ensure that every artifact is present in at least one configuration in the sample and gets a chance of being tested. However, simply including an implementation artifact in a product does not grantee that is indeed being tested. Therefore, Ruland et al. [11] argue that, in addition to including each artifact, at least one test case must also cover each artifact in order to properly test it. While the approach of Tartler et al. [10] can generate a relatively small sample, the approach of Ruland et al. [11] produces a relatively large sample, but guarantees that each artifact will indeed be tested. Therefore, with this work, we aim to find a reasonable trade-off between both approaches by building on the approach of Tartler et al. [10] and try to increase testing effectiveness by combining it with t-wise interaction sampling.

We propose an extension of Tartler et al.'s coverage criterion *t-wise presence condition coverage*, which combines t-wise interaction coverage with presence condition coverage of implementation artifacts. Rather than counting only interactions between features, our new criterion considers interactions of presence conditions of implementation artifacts. A sample with a t-wise presence condition coverage of 100% ensures that every *t-wise interactions of all implementation artifacts* is contained in at least one product and can thus be tested. In addition, we present an algorithm for *t-wise presence condition sampling*, which generates samples with a 100% *t-wise presence condition coverage* for a given configurable system and a given t. This sampling algorithm works independently from the employed variability mechanisms such as preprocessors, build systems, feature-oriented programming, or aspect-oriented programming. We implemented our algorithm within a prototype called *PRESICE (PRESence condItion CoverageE)* to investigate its testing effectiveness, testing efficiency, and sampling efficiency compared to traditional t-wise interaction sampling algorithms. Our current prototype supports extraction of presence conditions from SPLs that use the C preprocessor and the Kbuild build tool (e.g., `BusyBox` and `Linux`). Additionally, we extensively evaluate both, the coverage criterion and the algorithm using 27 real-world systems from different domains. In summary, we contribute the following:

- We propose a novel coverage criterion for product-based testing: t-wise presence condition

coverage.

- We present a sampling algorithm for t-wise presence condition coverage.
- We provide an open-source implementation named named PRESICE as part of FeatureIDE[1].
- We evaluate our coverage criterion and algorithm using 27 real-world systems.
- We publish all data from our experiments[2].

2 Running Example and Problem Statement

In the following, introduce a running example, which we use throughout the paper. We use this example to describe the potential problems with the current approaches for t-wise interaction sampling and motivate our solution.

2.1 Running Example

To illustrate the challenge of finding effective and efficient samples, we show a slightly edited code snippet from the system `BusyBox` in Figure 1, which uses the C preprocessor [12] to implement its variability [13]. The example is taken from the file `tftp.c`, which handles client-server communication via tftp. The code contains five features, `TFTP` (T), `TFTP_GET` (G), `TFTP_PUT` (P), `TFTP_BLOCKSIZE` (B), and `TFTP_DEBUG` (D), which each can be set to either *true* or *false*. We display their dependencies in Figure 2 using an excerpt of the `BusyBox` variability model, represented as a feature diagram. The additional features `BusyBox_TFTP` (BB) and `TFTPD` (TD) do not appear in the code snippet, but are necessary to visualize the feature diagram hierarchy. In the remainder of the paper, we use the provided abbreviations of the feature names to ease the readability of all propositional formulas using these features.

In the example, we changed the statement in Line 671 such that a compilation error occurs for certain products. The variable `blksize` is declared in Line 626, which is dependent on feature B. Then, `blksize` is used in Line 671, which is dependent on feature D. Thus, if D is selected in a configuration, but B is not, the generated product will be syntactically incorrect.

2.2 Problem Statement

T-wise interaction coverage only considers interactions between single features, and thus is based purely within the problem space of an SPL. This can lead to two flaws. First, t-wise interaction coverage may consider some irrelevant feature interactions, which can yield a large sample, potentially leading to a low testing efficiency. Second, t-wise interaction coverage may not be sufficient to reliably detect a fault resulting from an interaction of a degree larger than t, potentially leading to a low testing effectiveness. In order to reliably find faults with a high interaction degree a high value for t is required. However, traditional t-wise interaction coverage does not scale well with higher values for t, as the number of possible feature interactions grows exponentially with the parameter t. Thus, often a low value, such as $t = 2$ or $t = 3$, is chosen, which keeps the testing effort manageable, but also decreases the fault detection rate. In the following, we explain both flaws in more detail. With the introduction of our new coverage criterion, we aim to address both of these flaws by taking the solution space into account.

[1] `https://featureide.github.io/`

[2] `https://github.com/skrieter/evaluation-pc-sampling`

```
 1 #if TFTP_GET || TFTP_PUT // G ∨ P
   // ...
621 # if TFTP // T
622 int tftp_main(int argc, char **argv) {
        // ...
625     #  if TFTP_BLOCKSIZE // B
626     const char *blksize_str=TFTP_BLKSIZE_DEFAULT;
        // ...
649     int blksize = tftp_blksize_check(blksize_str, 65564);
650     if (blksize < 0) return EXIT_FAILURE;
651     #  endif
        // ...
670     #  if TFTP_DEBUG // D
671     printf("blksize = %d\n", blksize); // changed
672     #  endif
        // ...
690 }
691 # endif
   // ...
827 #endif
```

Figure 1: Example adopted from `BusyBox`.

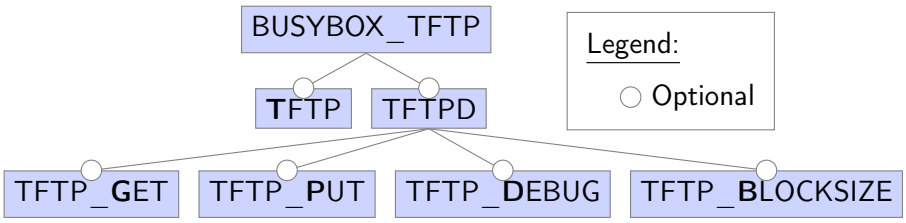

Figure 2: Excerpt of *BusyBox* feature diagram.

Efficiency of Sample-Based Testing In sample-based testing, we run the all test cases of a system once per sampled configuration. Although the execution time of the test cases may differ from configuration to configuration, in general, if the number of configurations in a sample (i.e., the sample size) increases the overall testing effort increases as well. Thus, analogous to other research, we consider the testing of smaller samples to be more efficient (i.e., *testing efficiency* [11]).

Applying pair-wise interaction sampling to our running example considers every interaction between two features. For instance, in Figure 1, for the features P and G, all four possible interactions are considered. However, the three interactions $(P, \neg G)$, $(\neg P, G)$, (P, G) all lead to the same product, as all of them satisfy the expression in Line 1. Thus, it is sufficient to consider only two interactions (e.g., $(\neg P, \neg G)$ and (P, G)) for this code snippet. In the table below, we show the sample generated from the t-wise interaction sampling algorithm ICPL [14, 15], which consists of six configurations:

Feature	**Configurations**					
	01	**02**	**03**	**04**	**05**	**06**
TFTP (T)	-	✓	-	✓	✓	-
TFTP_GET (G)	-	✓	✓	-	✓	-
TFTP_PUT (P)	-	✓	-	-	✓	✓
TFTP_DEBUG (D)	-	-	✓	-	✓	✓
TFTP_BLOCKSIZE (B)	-	✓	✓	-	-	✓

All four interactions of P and G are included. Including such unnecessary interaction can lead

to a larger sample, and thus to a lower testing efficiency.

Effectiveness of Sample-Based Testing We consider a sample to be more effective, the more faults could be detected using its derived products. Analogous to other research, we refer to this as *testing effectiveness* [11].

Although the fault in Figure 1 apparently involves only two features, it is in fact a feature interaction of degree four. To actually generate a product that contains the error, the corresponding configuration must have the feature B deselected and the features T, D, and G or P selected. Below, we show the sample generated by the pair-wise sampling algorithm IncLing [16]:

Feature	Configurations						
	01	**02**	**03**	**04**	**05**	**06**	**07**
`TFTP (T)`	✓	-	✓	-	-	-	✓
`TFTP_GET (G)`	✓	-	-	✓	-	-	✓
`TFTP_PUT (P)`	✓	-	-	✓	-	✓	-
`TFTP_DEBUG (D)`	✓	-	-	-	✓	✓	-
`TFTP_BLOCKSIZE (B)`	✓	-	✓	-	✓	-	-

Although the interaction $(D, \neg B)$ is covered in Configuration 06, the actual fault will not be included in the product, as the feature T is not selected and the preprocessor will remove the entire code block. Thus, the testing effectiveness of this sample is decreased.

3 Foundations of Configurable Systems

Before we can properly introduce the concept of t-wise presence condition coverage, we have to provide some essential definitions on feature modeling, configurations, and presence conditions. Our approach takes a feature model and a list of presence conditions as input and generates a list of configurations (i.e., a sample). Therefore, in the following, we revisit the basic notion of feature models, configurations, and presence conditions. We introduce all notions using propositional formulas and set notation.

3.1 Feature Models

A feature model specifies all features of an SPL and their interdependencies [17–19]. We define a feature model $\mathcal{M} = (\mathcal{F}, \mathcal{D})$ as a tuple, consisting of a set of features $\mathcal{F}$ and a set of dependencies $\mathcal{D}$ on $\mathcal{F}$. All features of a feature model are contained in the set $\mathcal{F} = \{F_1, ..., F_n\}$, where n is the total number of features. We represent the dependencies of a feature model as clauses of a propositional formula in conjunctive normal form (CNF). Each dependency in $\mathcal{D}$ represents one such clause over $\mathcal{F}$ (i.e., $\mathcal{D} = \{D_1, ..., D_m\}$, where m is the number of clauses). We denote a clause as a set of literals over $\mathcal{F}$. A literal is either a feature from $\mathcal{F}$ (i.e., a positive literal) or a negated feature from $\mathcal{F}$ (i.e., a negative literal). We define the function $\mathcal{L}$ that provides the set of literals for a feature model, $\mathcal{L}(\mathcal{M}) = \{false, true, \neg F_1, ..., \neg F_n, F_1, ..., F_n\}$. For example, consider the features `TFTP_PUT` (P), `TFTPD` (TD), and `BUSYBOX_TFTP` (BB) from Figure 2. Their dependencies can be written as the two clauses $D_1 = \neg P \vee TD$ and $D_2 = \neg TD \vee BB$.

3.2 Configurations

A configuration represents a selection of features from a feature model. From a configuration, we can derive the corresponding product using the variability mechanism of the SPL [17–19]. We define a configuration as a set of literals C, such that $C \subseteq \mathcal{L}(\mathcal{M})$ with $\forall l \in \mathcal{L}(\mathcal{M}) : l \notin$

$C \vee \neg l \notin C$. If a literal is contained in a configuration, the corresponding feature is defined as either selected (positive literal) or deselected (negative literal).

If all features are defined within a configuration, we call it *complete* and otherwise *partial*. We define this with the function:

$$complete(C, \mathcal{M}) = \begin{cases} true & |C| = |\mathcal{F}| \\ false & \text{otherwise} \end{cases}$$

A clause of a feature model represents a disjunction of literals. Thus, if a configuration contains at least one literal from a clause in $\mathcal{D}$, it satisfies this clause. Contrary, if a configuration contains all complementary literals of a clause, it contradicts this clause and, hence the entire feature model. Thus, if a configuration contradicts at least one clause, we call it *invalid*. We call a configuration *valid*, if it allows all clauses of a feature model to be satisfied:

$$valid(C, \mathcal{M}) = \begin{cases} true & \exists C' \supseteq C : \forall D \in \mathcal{D} : C' \cap D \neq \emptyset \\ false & \text{otherwise} \end{cases}$$

Note that, a partial configuration may also neither satisfy nor contradict a clause. In particular, a valid configuration may be partial and is not required to satisfy all clauses, as long as all clauses *can be satisfied* by adding more literals to the configuration.

3.3 Presence Conditions

A *presence condition* is a propositional formula over a feature model that describes whether a certain aspect is true (i.e., present) for a given configuration and/or its respective product. A presence condition is *not* part of a feature model and does *not* limit the valid configuration space. Rather it describes a subset of the valid configuration space (i.e., the set of all valid configurations that satisfy its propositional formula) for which a certain aspect is true. In this paper, we use presence conditions to describe whether an implementation artifacts is present in a product or not. In particular, we are interested in single lines of code (i.e., statements). Furthermore, in our evaluation, we also use presence conditions to describe whether a particular fault is present in a product or not. If and only if a configuration contains a combination of feature selections that satisfy a presence condition, the corresponding implementation artifact is included in the respective product of that configuration. If a presence condition is satisfied by a configuration we call it *active* and otherwise *inactive*.

For instance, Line 622 of Figure 1 has the presence condition $(G \vee P) \wedge T$, which can be derived from the nested C preprocessor annotations within the source code. This means that Line 622 is present in a product if the corresponding configuration has the feature G or P selected and the feature T selected, because the presence condition is active for these configurations. When we use presence conditions, in most cases, we are interested in which feature selections would make a presence condition active. In the example above these are the combinations $G \wedge T$ and $P \wedge T$, if any of these two is true, the presence condition is active. For this reason, it is beneficial to represent presence conditions in *disjunctive normal form* (DNF), which consists of a disjunction of conjunctive clauses. In a DNF, each clause represents a feature selection that satisfies the presence condition. In order to include an artifact in a product, at least one clause of the DNF must be satisfied, which means that all of its literals must be contained within a configuration. For example, we can write the presence condition of Line 622 as $\mathcal{P}^{+}_{622} = (G \wedge T) \vee (P \wedge T)$. Thus, in order to include this line in a product, the corresponding configuration must either contain features G and T or P and T.

To represent presence conditions within our formalism, we use a similar notation as for dependencies. We define a presence condition such that $\mathcal{P} = \{P_1, ..., P_k\}$ with $P_i \subseteq \mathcal{L}(\mathcal{M})$. In the context of t-wise presence condition coverage, we refer to a DNF clause simply as clause.

Further, we define the function *active* to check whether a presence condition is satisfied for a valid configuration:

$$active(\mathcal{P}, C) = \begin{cases} true & \exists P \in \mathcal{P} : P \subseteq C \\ false & \text{otherwise} \end{cases}$$

4 Presence Condition Coverage

In the following, we define our coverage criterion for t-wise presence condition coverage, based on presence conditions, the feature model, the parameter t, and a configuration sample. To this end, we describe interactions between presence conditions and how our coverage criterion is calculated for a given sample and t.

4.1 Presence Condition Interactions

An interaction between two presence conditions (i.e., between their implementation artifacts) is similar to interactions between features. However, the key difference is that a presence condition can be an arbitrary propositional formula. In a complete configuration, a feature can either be selected, if its corresponding positive literal is included in the configuration, or deselected, if its negative literal is included. In contrast, a presence condition may be active or inactive for several different literal combinations (cf. Section 3.3). Thus, an interaction between t presence conditions must include all possible union sets of these literal combinations.

Inactive Presence Conditions In order to account for the interactions between *present* and *absent* implementation artifacts, we must consider the interactions between *active* and *inactive* presence conditions. For this reason, we construct the complement of each presence condition, which is itself a presence condition and represents all literal combinations, for which the original presence condition is inactive. To this end, we negate the formula of a presence condition and convert it back into a DNF. As an example, consider the presence condition $\mathcal{P}^{+}_{626} = (G \wedge T \wedge B) \vee (P \wedge T \wedge B)$ for Line 626 of Figure 1. We process this formula as follows:

$$\begin{aligned} \mathcal{P}^{-}_{626} &= \neg\, \mathcal{P}^{+}_{626} \\ &= \neg((G \wedge T \wedge B) \vee (P \wedge T \wedge B)) \\ \textit{De Morgan's Law} &\equiv (\neg G \vee \neg T \vee \neg B) \wedge (\neg P \vee \neg T \vee \neg B) \\ \textit{Distributive Law} &\equiv (\neg G \wedge \neg P) \vee (\neg T) \vee (\neg B) \end{aligned}$$

After negating the formula, we apply De Morgan's law to get a CNF and then apply the distributive law to convert it back into a DNF. While the application of the distributive law could theoretically lead to an exponential growth in the number of clauses, in practice a presence condition consists of only a few clauses and literals, which makes the computational effort for this transformation neglectable. From the new DNF $\mathcal{P}^{-}_{626}$, we see that if a configuration either contains the literal $\neg T$, $\neg B$, or both, $\neg G$ and $\neg P$, the resulting product will not include Line 626.

For a given SPL, we construct one set $\mathcal{PC}$ that contains DNFs of all presence conditions from the SPL and their corresponding complements. To construct all t-wise interactions for a given SPL, we can then generate all t-wise combinations of $\mathcal{PC}$ by constructing the Cartesian product for the given t (i.e., $\mathcal{PC}^t$).

Combined Presence Condition For each interaction of t presence conditions, we can build a new combined presence condition $\mathcal{P}_I$ that is satisfied if and only if all individual presence conditions are active or inactive, respectively. To this end, we conjoin the DNFs of all presence conditions involved in a given interaction and converting the resulting expression back to a DNF. For each presence condition that is active in the given interaction we use its original DNF

and for each presence condition that is inactive we use its complementary DNF. For instance, consider the DNFs for *excluding* Line 626 and *including* Line 671 of Figure 1:

$$\begin{aligned} \text{Line 626:} \quad & \mathcal{P}^{-}_{626} = (\neg G \wedge \neg P) \vee (\neg T) \vee (\neg B) \\ \text{Line 671:} \quad & \mathcal{P}^{+}_{671} = (G \wedge T \wedge D) \vee (P \wedge T \wedge D) \end{aligned}$$

We build the presence condition for the interaction ($\mathcal{P}_I$) by conjoining the literals of each pair-wise clause combination:

$$\begin{aligned} \mathcal{P}_I &= \mathcal{P}^{-}_{626} \wedge \mathcal{P}^{+}_{671} \\ &\equiv_{\textit{Distributive Law}} ((\neg G \wedge \neg P) \wedge (G \wedge T \wedge D)) \vee ((\neg G \wedge \neg P) \wedge (P \wedge T \wedge D)) \vee \\ &\quad ((\neg T) \wedge (G \wedge T \wedge D)) \vee ((\neg T) \wedge (P \wedge T \wedge D)) \\ &\quad ((\neg B) \wedge (G \wedge T \wedge D)) \vee ((\neg B) \wedge (P \wedge T \wedge D)) \\ &\equiv_{simplify} (\neg B \wedge G \wedge T \wedge D) \vee (\neg B \wedge P \wedge T \wedge D) \end{aligned}$$

After merging, we further simplify the combined DNF by removing contradictions and redundant clauses. In case of our example, this results in the simplified DNF $(\neg B \wedge G \wedge T \wedge D) \vee (\neg B \wedge P \wedge T \wedge D)$. Thus, if a configuration either contains the literals $\neg B$, G, T, and D or $\neg B$, G, T, and P Line 671 will appear in the resulting product, but not Line 626.

4.2 Presence Condition Coverage Criterion

Given a set of presence conditions ($\mathcal{PC}$), feature model ($\mathcal{M}$), t, and configuration sample ($\mathcal{C}$) for an SPL, we can determine a value for our coverage criterion *t-wise presence condition coverage*. We define t-wise presence condition coverage as the ratio between the number of t-wise presence condition interactions that are covered by $\mathcal{C}$ and the number of valid t-wise presence condition interactions for $\mathcal{M}$. Naturally, with t-wise presence condition sampling we aim to compute a sample that achieves a coverage of 100%.

In our running example, there are five different presence conditions and their five complements.

$true$,	$false$,
$(G) \vee (P)$,	$(\neg G \wedge \neg P)$,
$(G \wedge T) \vee (P \wedge T)$,	$(\neg G \wedge \neg P) \vee (\neg T)$,
$(G \wedge T \wedge B) \vee (P \wedge T \wedge B)$,	$(\neg G \wedge \neg P) \vee (\neg T) \vee (\neg B)$,
$(G \wedge T \wedge D) \vee (P \wedge T \wedge D)$,	$(\neg G \wedge \neg P) \vee (\neg T) \vee (\neg D)$

In total, there are 11 valid combinations. Considering the sample from IncLing in Section 2.2, there are two interactions that are not covered by any configuration in the sample: $(G \wedge T \wedge B \wedge \neg D) \vee (P \wedge T \wedge B \wedge \neg D)$ and $(G \wedge T \wedge \neg B \wedge D) \vee (P \wedge T \wedge \neg B \wedge D)$. Thus, the number of interactions covered by the sample is 9, which results in a pair-wise presence condition coverage of approximately 82%.

5 T-Wise Presence Condition Sampling

In the following, we describe our algorithm to achieve t-wise presence condition coverage. The algorithm consists of three steps. First, we extract the presence conditions for each line of code from the given SPL. Second, we construct the set $\mathcal{PC}$ by *preprocessing the presence conditions* extracted from the SPL, removing tautologies, contradictions, and equivalent conditions and computing the complementary presence conditions (cf. Section 4.1). Third, we iteratively *construct a configuration sample* that includes all valid t-wise combinations of $\mathcal{PC}$.

5.1 Extracting Presence Conditions

As a first step, we have to extract a list of presence conditions from the target SPL. This step is highly dependent on the specifics of the given SPL, such as the programming language, the variability mechanism, and the configuration mechanism. Therefore, a general approach for this step is out of scope for this paper. In fact, both processes, extracting presence conditions and testing products, are independent from our approach and can be adapted to any suitable variability mechanism, programming language, and testing framework. We did however implement an algorithm for extracting presence conditions from SPLs that use the language C and the C preprocessor as a variability mechanism. The algorithm is based on the tool PCLocator [20], which is able to parse C files and analyze preprocessor annotations. The basic procedure from this algorithm would also work for other preprocessor implementations, such as Antenna for Java, but would of course require a different parser implementation. In the following, we describe the general logic behind the algorithm. We give some more details for our particular implementation in our prototype PRESICE in Section 6.1.1.

In general, the algorithm parses a C file and determines a presence condition for each line. As a result the algorithm returns a list of presence conditions for each file. The C preprocessor annotations that are used for variability management always form a block around variable code artifacts. Each block begins with an annotation, such as `#if` or `#ifdef` and ends with an annotation, such as `#else` or `#endif`. These block can be nested, as we display in our example in Figure 1. Consequently, for each line in a C file, the algorithm determines the C preprocessor blocks that surrounds the line. The presence condition of the line is then constructed by the conjunction of the conditionals of the surrounding blocks.

Due to the high expressiveness of the C preprocessor, the extraction algorithm has some limitations, which can lead to incorrect results in some rare cases. Most notably, the algorithm does not expand any preprocessor macro statements, which can lead parsing problems or missed annotations. In case of a parsing problem for a line, the impacted presence conditions are assumed to be true (i.e., the lines are always present). Further, the algorithm does only consider Boolean features. All features with numerical or string values are ignored. Due to these limitations, it is possible that some returned presence condition might be incorrect or incomplete.

5.2 Preprocessing Presence Conditions

Our core algorithm for t-wise presence condition sampling expects as input a list of presence conditions $\mathcal{PC}$, a feature model $\mathcal{M}$, and a value for t. From the previous step, we get a raw list of extracted presence conditions from an SPL, which can be arbitrary propositional formulas in any order. Therefore, the next step is to preprocess this initial list. In detail, we convert all presence conditions into DNF (cf. Section 3.3), compute their complements (cf. Section 4.1), and clean up the list by removing equivalent formulas, tautologies, and contradictions. In the end, we converted all raw presence conditions into a minimal list of DNFs $\mathcal{PC}$. Note that we do not need to convert a presence condition from CNF to DNF, but directly convert the propositional formulas extracted from the implementation artifacts, which typically contain only a small set of features and no complicated formulas.

Regarding our example in Figure 1, we extract the following raw presence conditions (for brevity, we already omit duplicates):

$$\mathit{true}, \quad G \vee P, \quad (G \vee P) \wedge T, \quad (G \vee P) \wedge T \wedge B, \quad (G \vee P) \wedge T \wedge D$$

The preprocessing results in the following list:

$$(G) \vee (P), \quad (G \wedge T) \vee (P \wedge T), \quad (G \wedge T \wedge B) \vee (P \wedge T \wedge B),$$
$$(G \wedge T \wedge D) \vee (P \wedge T \wedge D), (\neg G \wedge \neg P), \quad (\neg G \wedge \neg P) \vee (\neg T),$$
$$(\neg G \wedge \neg P) \vee (\neg T) \vee (\neg B), \quad (\neg G \wedge \neg P) \vee (\neg T) \vee (\neg D)$$

Splitting our algorithm in a preprocessing and a sampling part allows us to flexibly determine the input for the actual sampling algorithm. For instance, we can use the sampling algorithm also for regular t-wise interaction sampling, if we construct $\mathcal{PC}$ accordingly (i.e., $\mathcal{PC} = \{L \subseteq \mathcal{L}(\mathcal{M}) \mid |L| = 1)$. Furthermore, it also allows us to build subsets of $\mathcal{PC}$ and only consider interactions between these. To this end, we run the sampling algorithm once for each subset and iteratively add configurations to $\mathcal{C}_{Sample}$. Using this, we can, for example, group presence conditions for single files or folders, and thus limit the number of possible interactions and the resulting sample size.

5.3 Constructing a Configuration Sample

After computing the set $\mathcal{PC}$, we continue with the third step of our algorithm, constructing a sample of valid and complete configurations $\mathcal{C}_{Sample}$. For this, we extend the existing algorithm YASA [21], a deterministic greedy algorithm. We start with an empty sample $\mathcal{C}_{Sample}$ and then iterate over all t-wise interactions one at a time. For each, we either add a new partial configuration to the sample or the literals of one clause of the interaction's presence condition to an existing configuration. Due to the nature of this algorithm, we are guaranteed to check every possible combination of the given presence conditions exactly once. For each combination or presence conditions, we either include it in at least one configuration in the sample or determine that it cannot be covered by any valid configuration. Since there are finitely many possible combinations (i.e., $2^t \cdot \binom{k}{t}$ with k being the number of presence conditions), the algorithm always terminates and guarantees that a presence condition coverage of 100% is achieved by the computed sample.

We present pseudocode of the second step in Algorithm 1. It takes a parameter t, the feature model $\mathcal{M}$, and a list of DNFs $\mathcal{PC}$ and returns a sample $\mathcal{C}_{Sample}$ that covers every t-wise combination of DNFs in $\mathcal{PC}$. At the start, we initialize $\mathcal{C}_{Sample}$ with an empty set (Line 1). To list all presence condition interactions up to degree t, we then build the Cartesian product for $\mathcal{PC}$ with itself t times (i.e., $\mathcal{PC}^t$) (Line 2). For each interaction in $\mathcal{PC}^t$, we compute the combined presence condition $\mathcal{P}_I$ as outlined in Section 4.1 (Line 3). Using $\mathcal{P}_I$, we check whether this interaction is already covered by at least one configuration in the sample (Line 4). If not, we try to cover the interaction by iterating over all clauses in $\mathcal{P}_I$ (Line 6). For each clause P, we first check, whether it is satisfiable with regard to the feature model $\mathcal{M}$ (Line 7). If so, we add it to a temporary set of valid clauses $\mathcal{P}_{valid}$ (Line 12). Second, we iterate over all configurations in our current sample and check, whether we can add the literals of P to it without causing a contradiction (Line 9). If it does not cause a contradiction, we add all literals to the configuration (i.e., covering the interaction) and continue with the next interaction (Line 10 and 11). Otherwise, if we cannot find any suitable configuration for any clause in $\mathcal{P}_I$, we use the smallest clause in $\mathcal{P}_{valid}$, if any, and use it to build a new configuration that we add to our sample (Line 13 and 14).

As an example, we describe some iterations of the algorithm for Figure 1. In the first iteration we get the combination $I = ((G \vee P), (\neg G \wedge \neg P))$ (Line 2), which is converted into the combined presence condition $\mathcal{P}_I = G \wedge \neg G \vee P \wedge \neg P$ (Line 3). As there is no configuration yet in $\mathcal{C}_{Sample}$, we continue (Line 4). Both clauses in $\mathcal{P}_I$, $P_1 = G \wedge \neg G$ and $P_2 = P \wedge \neg P$ are invalid and are therefore not considered for inclusion into a configuration (Line 7). Thus, we continue with the next iteration. In the second iteration we get the combination $I = ((G \vee P), ((G \wedge T) \vee (P \wedge T)))$ (Line 2), which results in $\mathcal{P}_I = (G \wedge T) \vee (P \wedge T)$ (Line 3). There

Algorithm 1: Pseudocode for Constructing a Sample

```
    Data: Presence Conditions 𝒫𝒞, Feature Model ℳ, Parameter t
    Result: List of configurations 𝒞_Sample
1   𝒞_Sample ← ∅
2   for I ∈ 𝒫𝒞^t do
3       𝒫_I ← {⋃_{P∈T} P | T ∈ ∏_{𝒫∈I} 𝒫}
4       if ∄C ∈ 𝒞_Sample : active(𝒫_I, C) then
5           𝒫_valid ← ∅
6           for P ∈ 𝒫_I do
7               if valid(P, ℳ) then
8                   for C ∈ 𝒞_Sample do
9                       if valid(C ∪ P, ℳ) then
10                          C ← C ∪ P
11                          continue line 2
12                  𝒫_valid ← 𝒫_valid ∪ P
13          if 𝒫_valid ≠ ∅ then
14              𝒞_Sample ← 𝒞_Sample ∪ min(𝒫_valid)
15  return 𝒞_Sample
```

is still no configuration in $\mathcal{C}_{Sample}$, which may cover this interaction (Line 4). Both clauses $P_1 = G \wedge T$ and $P_2 = P \wedge T$ are valid (Line 7), but there is no configuration yet to which they could be added (Line 8). Thus, both of them are added to $\mathcal{P}_{valid}$ (Line 12). Next, the smallest clause in $\mathcal{P}_{valid}$ is added to $\mathcal{C}_{Sample}$ (Line 14). As both clauses have the same size, we use the first one, resulting in $\mathcal{C}_{Sample} = \{\{G, T\}\}$. In the third iteration we get combined presence condition $\mathcal{P}_I = (G \wedge T \wedge D) \vee (P \wedge T \wedge D)$. Both clauses are valid and can be added to the existing configuration in $\mathcal{C}_{Sample}$ (Line 9). We use the first clause, which results in $\mathcal{C}_{Sample} = \{\{G, T, D\}\}$ (Line 10). In the fourth iteration we get combined presence condition $\mathcal{P}_I = (G \wedge T \wedge \neg D) \vee (P \wedge T \wedge \neg D)$. Here, both clauses are valid, but conflict with the existing configuration in $\mathcal{C}_{Sample}$ (Line 9). Thus, a new configuration is added, It can be added to the existing configuration (Line 14), resulting in $\mathcal{C}_{Sample} = \{\{G, T, D\}, \{G, T, \neg D\}\}$.

The complete configuration sample after the algorithm terminated is shown in the following table:

Feature	**Configurations**				
	01	**02**	**03**	**04**	**05**
`TFTP (T)`	✓	✓	✓	-	-
`TFTP_GET (G)`	✓	✓	✓	✓	-
`TFTP_PUT (P)`	✓	✓	✓	✓	-
`TFTP_DEBUG (D)`	✓	-	✓	-	✓
`TFTP_BLOCKSIZE (B)`	✓	✓	-	-	✓

When comparing this sample to the sample generated by IncLing in Section 2.2, we see that, in contrast, it contains a configuration that covers the fault in the example (i.e., Configuration 03). This corresponds to an increase in testing effectiveness. Compared to the sample produced by ICPL in Section 2.2, we can see that it contains only the two real interactions of P and G, and thus requires only five configurations, which is an increase in testing efficiency.

6 Evaluation

With t-wise presence condition coverage we aim to generate samples for a novel coverage criterion, which we expect to increase the chance of detecting faults in product-based testing. We are interested in the degree of testing effectiveness and testing efficiency of the t-wise presence condition coverage criterion and our algorithm for t-wise presence condition sampling. Therefore, we evaluate whether samples generated with t-wise presence condition sampling can detect more faults than samples generated with t-wise interaction sampling. We also evaluate what degree of t-wise presence condition coverage can be achieved by existing algorithms for t-wise interaction sampling. Further, we evaluate the sample size (i.e., testing efficiency) and sampling time (i.e., sampling efficiency) of our tool PRESICE (cf. Section 5). In summary, we aim to answer the following research questions:

RQ_1 Is t-wise presence condition coverage more effective in detecting faults than t-wise interaction coverage for the same value of t?

RQ_2 What degree of presence condition coverage can be achieved using traditional sampling algorithms?

RQ_3 Is the *testing* of samples more efficient with t-wise presence condition sampling than with t-wise interaction sampling?

RQ_4 Is the *generation* of samples more efficient with t-wise presence condition sampling than with t-wise interaction sampling?

Within our experiments, we compute several samples for different systems using our tool PRESICE and a selection of different state-of-the-art t-wise interaction sampling algorithms and compare the samples with respect to our evaluation criteria. In the following, we describe the setup for our experiments and our evaluation results. First, we introduce the algorithms that we compare against each other. Second, we present the subject systems, for which we generate samples. Third, we describe our measuring methods for our four evaluation criteria, fault detection, coverage, sample size, and sampling time. Fourth, we analyze and discuss our results. Finally, we discuss potential threats to the validity of our evaluation.

6.1 Algorithms

We use several state-of-the-art algorithms for t-wise interaction sampling as comparison for testing efficiency and effectiveness, which were also used in previous evaluations [15, 16, 22]. First, we employ Chvátal [23], ICPL [14, 15], and IncLing [16] as pure t-wise interaction sampling algorithms. Second, we use PRESICE as a pure t-wise interaction sampling algorithm (cf. Section 5.2), which only uses the feature model as input (PRESICE-FM). All of these algorithms compute complete t-wise samples for certain values of t using different methods. Third, we use a random sampling algorithm [24]. Instead of aiming for a certain coverage criteria it generates a fixed number of valid random configurations. Fourth, we include the algorithm PLEDGE [25], which does not try to achieve a certain t-wise interaction coverage, but is based on an evolutionary algorithm to optimize a sample of fixed size such that its contained configurations are as dissimilar as possible. By increasing dissimilarity, the sample's t-wise interaction coverage should also increase. Although this approach does not guarantee a complete t-wise interaction coverage, it aims to increase sampling and testing efficiency while maintaining a reasonably good testing effectiveness. Finally, we use PRESICE to compute samples based on presence conditions (PRESICE-PC).

6.1.1 Implementation Details

The implementation of these algorithm is provided by multiple open-source Java libraries, which we employ in our evalution. Chvátal and ICPL are implemented in the SPLCATool [15]. IncLing and Random are implemented in FeatureIDE [24, 26]. PLEDGE is implemented in a library of the same name [25]. For all other sampling algorithms we use PRESICE, for which we employ our own implementation[3].

We implemented our prototype PRESICE for t-wise presence condition sampling. It includes an algorithm for extracting presence conditions from systems that use the *C preprocessor* and the *kbuild* build tool. PRESICE is written in Java and employs several other Java libraries to implement its functionality.

For parsing C files and identifying preprocessor statements, we use the tool PCLocator [20], which combines several C parsers, such as SuperC, TypeChef, and FeatureCoPP to achieve more accurate results. This tool computes a presence condition for each line in a source file. To this end, we analyze every C file (i.e., files with the file extensions `.c`, `.h`, `.cxx`, and `.hxx`) in the source directories of the target SPL. For this, we exclude special directories that do not contribute to the actual implementation of the system, but contain examples, configuration logic, or header files of system libraries. As result, we get a list containing propositional formulas for each code block within a C project. We use this list as input for the t-wise presence condition sampling. Currently, we did not implement an extraction algorithm for any other variability mechanism. Thus, in our evaluation, we focus on C projects that use the C preprocessor and kbuild as build system to enable variability. During the extraction process, we warn the user, if we find presence conditions that contain features that are not on the feature model and vice versa. For our evaluation, we only consider features that we can find in both, the feature model and the source code.

Within our sampling algorithm, we use the satisfiability solver *Sat4J* [27] to check for validity of configurations and presence conditions. Furthermore, we use *KClause* [28] to extract a feature model for C projects that use Kconfig as configuration tool.

Regarding the random sampling, we use the default random sampling algorithm of FeatureIDE [24]. Their implementation is based on *Sat4J* as well and generates configurations by asking the satisfiability solver for a valid configuration using a randomized feature order. While this algorithm does not generate uniformly distributed random samples, as it is biased by the internal structure of the solver, it is an efficient way to generate a high number of valid configurations. Note that it is possible for this algorithm to generate a sample that contains duplicate configurations.

6.1.2 Parameter Details

As we employ a variety of sampling algorithms in our evaluation, the required parameters differ for most of them. The only common parameter for every algorithm is the feature model, which specifies the feature dependencies. Naturally, all algorithms always use the same feature model as input.

Regarding the parameter t, not all algorithms support the same values. IncLing is designed as a strict pair-wise interaction coverage algorithm, and thus only works for $t = 2$. ICPL supports values for t up to 3 and Chvátal up to 4. As described in Section 5, we can run our algorithm for t-wise presence condition sampling with any value for t. However, PRESICE currently has a technical limitation that allows to process only up to 2^{31} interactions. To enable a fair comparison, we set the value of t to $t = 2$ for all algorithms.

As Random and PLEDGE do not try to achieve a certain t-wise coverage, but just generate a set of valid configurations, it is not possible to set a value for t. Instead, they require to set

[3] `https://github.com/skrieter/evaluation-pc-sampling`

the size of the sample in advance. In order to ensure a fair comparison, for PLEDGE, we set the sample size equal to the size of the largest sample computed by any variant of PRESICE (i.e., either PRESICE-FM, PRESICE-PC, or PRESICE-Concrete, which ever returned the largest sample). For Random, we set several sample sizes for each system, ranging from the smallest to the largest sample size produced for every system by any algorithm.

PLEDGE also requires to set a time limit for the evolutionary algorithm. We decided to compare two different limits, the maximum and minimum time that PRESICE needs to compute a sample for a particular model (independent from its parameters settings).

For PRESICE we also have to specify additional parameters beside t. We are able to specify which expressions should be considered for interaction (cf. Section 5.2). Thus, we test the following setting: *FM* considers all t-wise interactions within a feature model, and thus behaves like other pure t-wise interaction sampling algorithms. *PC* considers all t-wise interactions between all presence conditions of a system. Finally, *Concrete* considers t-wise interactions between features, but only includes features that appear in at least one presence condition (i.e., concrete features).

6.1.3 Summary

We compare results from the following algorithms:

1. Chvátal [23]
2. ICPL [14, 15]
3. IncLing [16]
4. PLEDGE-Min (Minimum run time) [25]
5. PLEDGE-Max (Maximum run time) [25]
6. PRESICE-FM (All features within a model)
7. PRESICE-PC (All presence conditions)
8. PRESICE-Concrete (All concrete features)
9. Random [24]

6.2 Subject Systems

Currently, PRESICE can extract presence conditions from C preprocessor statements. Thus, we selected real-world open-source systems that use the C preprocessor as a variability mechanism. In particular, we reused 21 systems from the study of Medeiros, Kästner, Ribeiro, Gheyi, and Apel [29], which also compared different sampling algorithms in terms of testing effectiveness. However, most of these systems do not have a separate feature model, which prevents us from taking their feature dependencies into account. For this reason, we include six real-world open-source systems that use the C preprocessor and the Kconfig tool, namely, `fiasco` (latest), `axtls` (latest), `uclibc-ng` (latest), `toybox` (latest), `BusyBox` (version 1.29.2), and `Linux` (version 2.6.28). For `Linux`, we use a feature model for version 2.6.28 provided by She et al. [30]. For all other systems, we extracted the feature models from their Kconfig files using the tool *KClause* [28].

In Table 1, we provide an overview of the all systems. At the top we show the 6 systems for which we have a feature model and at the bottom the 21 systems from the study of Medeiros, Kästner, Ribeiro, Gheyi, and Apel [29]. For each respective feature model we show its number of features ($\#F$), concrete features ($\#CF$) (i.e., features that appear in at least one presence

Table 1: Subject systems — features ($\#F$), concrete features ($\#CF$), dependencies ($\#D$), presence conditions ($\#PC$), and the number of clauses ($\#C$) and literals ($\#L$) over all presence conditions.

System (Version)	Feature Model			Presence Conditions		
	$\#F$	$\#CF$	$\#D$	$\#PC$	$\#C$	$\#L$
fiasco (latest)	71	7	120	9	12	14
axtls (latest)	95	32	190	90	126	162
uclibc-ng (latest)	270	104	1,561	225	315	406
toybox (latest)	323	8	90	14	14	14
BusyBox (1.29.2)	1,018	507	997	1,020	1,475	1,975
Linux (2.6.28.6)	6,888	1,696	80,715	3,512	5,494	8,767
busybox (1.23.1)	–	–	–	3,278	5,046	7,281
bison (2.0)	–	–	–	695	1,161	1,871
cvs (1.11.17)	–	–	–	1,495	2,491	3,785
libssh (0.5.3)	–	–	–	393	663	962
dia (0.97.2)	–	–	–	606	708	810
libxml2 (2.9.0)	–	–	–	2,420	4,423	6,757
xterm (224)	–	–	–	796	1,302	1,859
lighttpd (1.4.30)	–	–	–	567	875	1,219
libpng (1.5.14)	–	–	–	1,752	3,937	7,421
fvwm (2.4.15)	–	–	–	777	1,482	4,075
irssi (0.8.15)	–	–	–	318	369	428
gnome-keyring (3.14.0)	–	–	–	453	539	631
vim (6.0)	–	–	–	3,888	8,714	16,613
xfig (3.2.4)	–	–	–	378	802	1,969
totem (2.17.5)	–	–	–	223	278	332
gnome-vfs (2.0.4)	–	–	–	253	313	373
cherokee (1.2.101)	–	–	–	1,128	1,589	2,077
bash (4.2)	–	–	–	3,659	6,577	10,262
lua (5.2.1)	–	–	–	324	496	714
gnuplot (4.6.1)	–	–	–	1,546	2,720	4,145
apache (2.4.3)	–	–	–	1,814	2,915	4,360

condition), and dependencies ($\#D$). Regarding the extracted presence conditions, we show the total number of conditions ($\#PC$), and the number of literals ($\#L$) and clauses ($\#C$) over all presence condition.

6.3 Evaluation Setup

6.3.1 Measuring Fault Detection

To answer our first research question, we reuse some artifacts from the study of Medeiros, Kästner, Ribeiro, Gheyi, and Apel [29]. In the study the authors report known faults in multiple systems and their respective presence conditions[4]. In this case, if the presence condition of a fault is active under a given configuration, it means that the fault will be present in the corresponding product. In total, the study presents a list of 75 unique presence conditions. However, 23 of these conditions contained features that do not occur in the actual source code. This can be due to abstract features, features that are only used during the build process

[4] `http://www.dsc.ufcg.edu.br/~spg/sampling/`

Table 2: Overview of presence conditions of faults used from Medeiros, Kästner, Ribeiro, Gheyi, and Apel [29].

Degree	Count	Example
1	34	`!ENABLE_FEATURE_SYSLOG`
2	11	`ENABLE_FEATURE_EDITING && !ENABLE_HUSH_INTERACTIVE`
3	4	`ENABLE_FEATURE_GETOPT_LONG && !ENABLE_FEATURE_SEAMLESS_LZMA && !ENABLE_FEATURE_TAR_LONG_OPTIONS`
4	2	`!FEAT_GUI_W32 && !PROTO && !FEAT_GUI_MOTIF && !FEAT_GUI_GTK`
5	1	`!FEAT_GUI_W32 && !FEAT_GUI_GTK && !FEAT_GUI_MOTIF && !FEAT_GUI_ATHENA && !FEAT_GUI_MAC && FEAT_GUI`
$\sum$	52	

(e.g., in Makefiles), or due to features that have a different name in the configuration tool than in the source code. Therefore, we only used the remaining 52 presence conditions in our evaluation. Most of these presence conditions represent interaction of degree one, which means that the selection or deselection of a single feature is enough to make them active. However, the list also contains presence conditions that represent interactions of degree two, three, four, and five. We show the distribution of presence conditions in Table 2 together with an example for each degree of interaction. Note that, five of the 52 presence conditions also have multiple clauses in their DNF. For such a case, we consider the number of literals in the smallest clause as degree of interaction for that presence condition. For instance, the presence condition `(SHUTDOWN_SERVER && NO_SOCKET_TO_FD && START_RSH_WITH_POPEN_RW) || (NO_SOCKET_TO_FD && !SHUTDOWN_SERVER && START_RSH_WITH_POPEN_RW)` also represent an interaction of degree three, because it can be activated by the (de)selection of three features. Thus a complete three-wise interaction coverage would be guaranteed to find this fault. All in all, the study includes a wide variety of interaction faults with varying degrees of complexity.

We use the list of presence conditions to check whether samples generated for theses systems do cover each fault in at least one configuration. To this end, we generate samples for each of these systems with PRESICE-PC (i.e., presence condition coverage) and PRESICE-Concrete (i.e., interaction coverage) for $t = 1$ and $t = 2$. We then count how many reported faults are covered by each sample. To determine whether a fault is covered, we check if there exists at least one configuration in the sample that satisfies the corresponding presence condition of the fault.

Both algorithms are susceptible to the order of features or order of presence conditions that are provided as input, meaning that they will produce different results for different feature orders. Thus, we evaluate both algorithms using multiple iterations with a randomized feature order. In detail, we execute all algorithms 100 times, each time shuffling the feature order. To enable a fair comparison we use the same 100 randomized feature orders for each algorithm. A number of 100 iterations is an empirical value for our evaluation that provides a good trade-off between effort and accuracy.

Note, that we do not use any of the other algorithms in this experiment, as we do not have feature models for these systems. The lack of a feature model for a system also means that the configurations within a sample may be invalid according to the feature dependency of the system. However, without a feature model we are not able to test this.

6.3.2 Measuring Coverage

We compute the coverage achieved by every sample with regard to two different coverage criteria, pair-wise interaction coverage (FM) and pair-wise presence condition coverage (PC). We consider a sample and, consequently, its sampling algorithm to be more effective the higher its coverage, as it potentially exposes more faults in the code.

Similar to the previous experiment, all used sampling algorithms are susceptible to the feature order in a feature model. Thus, again, we execute all algorithms 100 times, each time shuffling the feature order. In addition, we execute Random 10 times for each feature order, which results in 1,000 iterations for each system. For `Linux`, we only use 5 iterations of the experiment, as most algorithms take several hours to compute just one sample.

6.3.3 Measuring Sample Size

Regarding testing efficiency, we count the number of configurations in each sample computed by each algorithm. We do not consider the time required to run any actual test cases of a particular system. We do not consider the time required to run any test cases of a particular system, as this time is depended on the actual test cases for each product and the general testing approach. Nevertheless, we can assume that the testing time increases with the number of configuration in a sample, and thus, in general, a smaller sample will lead to a smaller testing time. Analogous to measuring coverage, we execute each algorithm, except Random, 100 times and randomize the feature order. Random is again run 1,000 times for each sample size.

6.3.4 Measuring Sampling Time

For measuring sampling efficiency, we take the time that is needed for generating a sample with each algorithm. Each experiment runs on an own JVM, in order to mitigate any side effects (e.g., just-in-time compilation). As our algorithm requires additional information from the source code (i.e., the presence conditions), we differentiate between the time needed to extract the presence conditions from the source code and the time to actually generate the sample. This is relevant, as the extraction process only needs to be run once for each system. Though it takes some time to analyze the source code, the resulting presence conditions for each file can be saved for later reuse. For instance, if we compute samples for different values of t, we only need to run the extraction process once.

6.3.5 Computing Environment

We run all algorithms on the same computing environment, with the following specifications: *CPU*: Intel Core i5-8350U, *Memory*: 16 GB, *OS*: Manjaro (Arch Linux), *Java*: OpenJDK 15.0.2, *JVM Memory*: Xmx: 14 GB, Xms: 2 GB.

6.4 Evaluation Results

For brevity, we primarily present figures showing aggregated data over our measurement results. All data and a tabular overview can be found online.[5] We structure our findings according to our four research questions, that is fault detection, coverage, testing efficiency, and sampling efficiency. Afterwards, we analyze and discuss our results.

6.4.1 Faults Covered

We present the results of our first experiment in Table 3. For each algorithm and value for t, we show the number of faults that are covered or not covered by the produced samples across all

[5] `https://github.com/skrieter/evaluation-pc-sampling/tree/master/results`

Table 3: Faults covered across all 21 systems from Medeiros, Kästner, Ribeiro, Gheyi, and Apel [29], including aggregated sample size and sampling time over all systems.

Algorithm	**t**	**Size**			**Time (s)**			**Faults Covered**	
		∅	*Min*	*Max*	∅	*Min*	*Max*	*Yes*	*No*
PRESICE-PC	1	7.5	4	14	0.3	0.2	0.6	41	11
PRESICE-Concrete	1	2.0	2	2	0.3	0.2	0.4	36	16
PRESICE-PC	2	65.7	22	167	5.6	0.3	38.8	51	1
PRESICE-Concrete	2	16.7	12	24	0.9	0.2	3.7	47	5

Table 4: Relative mean sample size, mean sampling time, and mean coverage aggregated over all 6 systems with a feature model.

Algorithm	**∅Time (%)**	**∅Size (%)**	**∅Coverage (%)**	
			FM	*PC*
PRESICE-FM	100.0	100.0	100.0	98.6
PRESICE-PC	59.3	73.9	79.1	100.0
PRESICE-Concrete	36.1	16.6	61.7	62.9
ICPL	319.5	132.7	100.0	97.9
Chvátal	1,046.7	131.3	100.0	98.0
IncLing	53.6	153.7	100.0	99.3
PLEDGE-Min	51.5	122.0	98.8	97.1
PLEDGE-Max	118.2	122.0	98.8	97.1

systems. The number of covered faults is the *minimum* number over all 100 iterations, meaning that if any of the 100 samples for a system was not able to cover a particular fault it is *not* counted as covered. Analogous, the number of not covered faults is the *maximum* number over all 100 iterations. In addition, we report the aggregated sample size and sampling time over all systems. For both values, we report its minimum, maximum and average over all 21 systems and 100 iterations.

Of the 52 faults, which we investigated, we see that for both values of t PRESICE-PC is able to detect more faults than PRESICE-Concrete (i.e., 31 vs. 36 for $t = 1$ and 51 vs. 47 for $t = 2$). The presence condition that belongs to the fault that could not always be covered by PRESICE-PC with $t = 2$ is the following:

```
ENABLE_HUSH_CASE && ENABLE_FEATURE_EDITING_SAVE_ON_EXIT
  && ENABLE_HUSH_INTERACTIVE && !ENABLE_FEATURE_EDITING
```

This presence condition is of degree four, and thus, when using t-wise interaction sampling, is only guaranteed to be found with $t \geq 4$.

On the other hand, we can also see that on average PRESICE-PC produced larger samples than PRESICE-Concrete (i.e., 7.5 vs. 2.0 for $t = 1$ and 65.7 vs. 16.7 for $t = 2$). Thus, the higher fault detection may also be a result of the larger sample sizes. However, as we pointed out before, we do not use a feature model for this experiment. Therefore, there are no restrictions on the configuration space, which can lead to a lower sample size. Furthermore, as we see in later experiments, a feature model, which may also include abstract features, leads to a larger sample size than considering only concrete features.

6.4.2 Achieved Coverage

In Table 4, we show a comparison of the coverage for different criteria for all algorithms. These values are aggregated over all systems and all experiments using the arithmetic mean. We

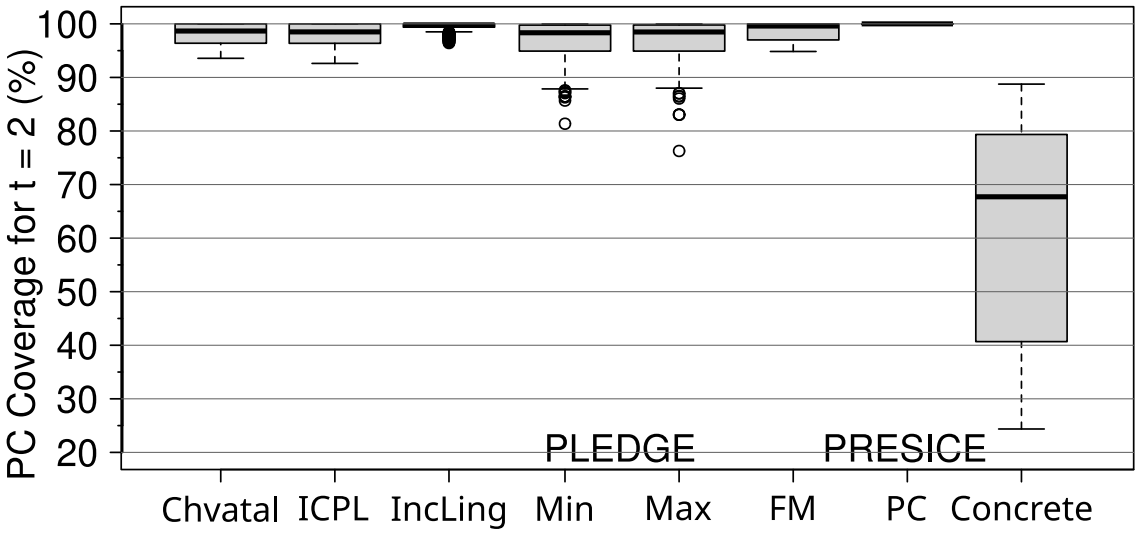

Figure 3: Pair-wise presence condition coverage aggregated over all systems.

Table 5: Results of the paired t-tests for difference in *FM* and PC coverage between PRESICE and other algorithms.

Algorithm	**ICPL**	**Chvatal**	**IncLing**	**Pledge**	
				Min	*Max*
FM-Coverage					
PRESICE-FM	=	=	=	+	+
PRESICE-PC	−	−	−	−	−
PRESICE-Concrete	−	−	−	−	−
PC-Coverage					
PRESICE-FM	+	+	−	+	+
PRESICE-PC	+	+	+	+	+
PRESICE-Concrete	−	−	−	−	−

show a more detailed plot of the coverage criterion *PC* over all systems and experiments in Figure 3 using boxplots. In addition, we performed paired t-tests to test whether the difference in achieved coverage by the different algorithms is significant. We present the results of the statistical tests in Table 5. In this table, we compare the coverage of all three variants of PRESICE with the coverage of all other algorithms for the two coverage criteria *FM* and *PC* with $t = 2$. The symbol = indicates that there is no significant difference (i.e., $p > 0.05$) in the achieved coverage between both algorithms. The symbol − indicates that the coverage achieved by the variant of PRESICE is significantly lower than the coverage of the other algorithm (i.e., $p < 0.001$). Analogous, the symbol + indicates the coverage of PRESICE is significantly higher (i.e., $p < 0.001$).

Regarding the coverage criterion *FM*, we can see that only the t-wise interaction sampling algorithms (Chvátal, ICPL, IncLing, PRESICE-FM) are able to achieve a 100% coverage. Both, PRESICE-PC and PRESICE-Concrete achieve a significant lower *PC* coverage. On the other hand, only PRESICE-PC is able to achieve a 100% coverage criterion *PC*. All other algorithms produce samples with a significant lower *PC* coverage on average. Still, many algorithms (Chvátal, ICPL, IncLing, PLEDGE, PRESICE-FM) achieve a rather high average *PC* coverage of over 97%.

Table 6: Results of the paired t-tests for sample size difference between PRESICE and other algorithms.

Algorithm	**ICPL**	**Chvatal**	**IncLing**	**Pledge**	
				Min	*Max*
PRESICE-FM	–	–	–	–	–
PRESICE-PC	–	–	–	–	–
PRESICE-Concrete	–	–	–	–	–

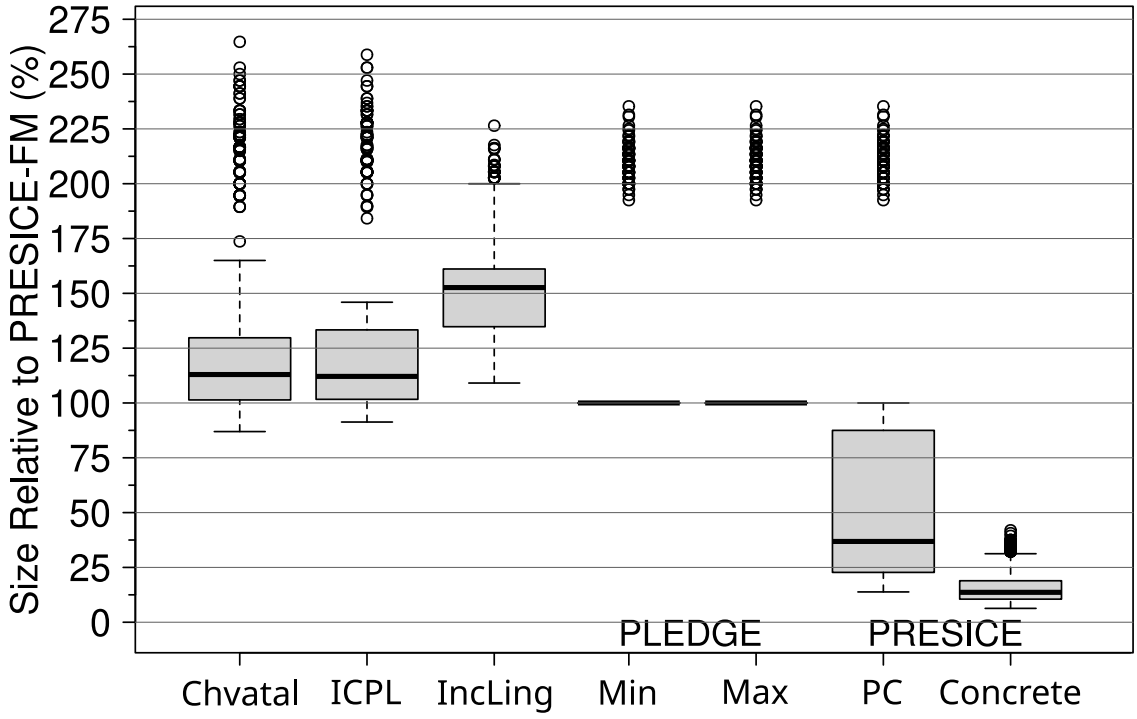

Figure 4: Sample size relative to PRESICE-FM for all 6 systems with a feature model.

6.4.3 Sample Size

In Table 4, we show a comparison of the sample sizes for all algorithms. Again, the values are aggregated over all systems and all experiments using the arithmetic mean. As the actual sample size is dependent on the subject system, we normalized the sample size for every experiment using the sample size of PRESICE-FM as 100%. In Figure 4, we depict the sample size for all algorithms in more detail using boxplots. Additionally, we show the absolute values of the mean sample size per system for PRESICE-FM and PRESICE-PC in Table 7. Furthermore, we performed paired t-tests to test whether the difference in sample size computed by the different algorithms is significant and show the results in Table 6. In this table, we compare the sample size of all three variants of PRESICE with the sample size of all other algorithms. The symbol – indicates that the sample size computed by the variant of PRESICE is significantly smaller than the sample size of the other algorithm (i.e., $p < 0.001$). Analogous, the symbol + indicates the sample size of PRESICE is significantly greater (i.e., $p < 0.001$).

We can observe that on average all algorithms that use presence conditions as input produce significantly smaller samples than algorithm that only use the feature model. An exception is the system `BusyBox`, for which PRESICE-PC produces samples that are about two times larger than the sample of PRESICE-FM.

Table 7: Absolute mean sample size and mean sampling time for PRESICE-FM and PRESICE-PC for all 6 systems with a feature model.

System	∅Size		∅Time (s)		∅Extract (s)
	FM	PC	FM	PC	
fiasco	21.5	5.4	1.0	0.6	0.7
axtls	32.3	27.3	1.4	1.3	1.4
uclibc-ng	362.4	54.5	6.8	3.3	0.8
toybox	18.4	6.5	3.6	0.7	4.5
BusyBox	37.6	79.1	28.6	20.7	2.1
Linux	493.4	189.4	8,938.4	1,248.6	64.5

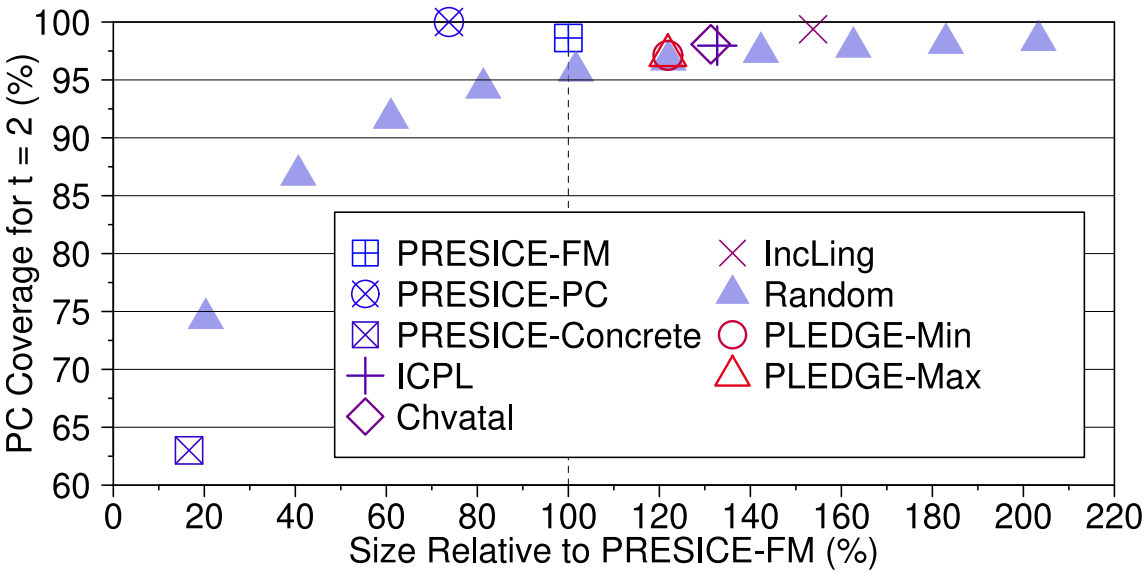

Figure 5: Presence condition coverage compared to sample size for all algorithms aggregated over all 6 systems with a feature model.

6.4.4 Correlation Between Coverage and Sample Size

To illustrate the correlation between sample size and testing effectiveness, we show, in Figure 5, a comparison of the coverage criterion PC with $t = 2$ for all algorithms for different configuration sizes. On the x-axis, we show the sample size relative to the sample size of PRESICE-FM (i.e., being 100%). On the y-axis, we show coverage in % for PC with $t = 2$. Each data point represents the average for all samples per algorithm and system. Random acts as a base line in this diagram, as it does not aim for a certain coverage criterion. We can see a clear correlation between sample size and the coverage criterion PC (i.e., increasing the sample size leads to higher coverage on average). Further, we can see that PRESICE can reach a 100% coverage with a substantially smaller sample than all other tested algorithms for most cases.

In addition, we calculate the Spearman's rank correlation coefficient between the degree of coverage and the sample size for all algorithms. For the coverage criterion PC, we get a significant positive correlation of ≈ 0.157 ($p < 0.001$). Similarly, for the coverage criterion FM, we also get a significant positive correlation of ≈ 0.2 ($p < 0.001$).

6.4.5 Sampling Time

In Table 4, we show the average sampling time for all algorithms, aggregated over all systems using the arithmetic mean and relative to the sampling time of PRESICE-FM. In addition, we

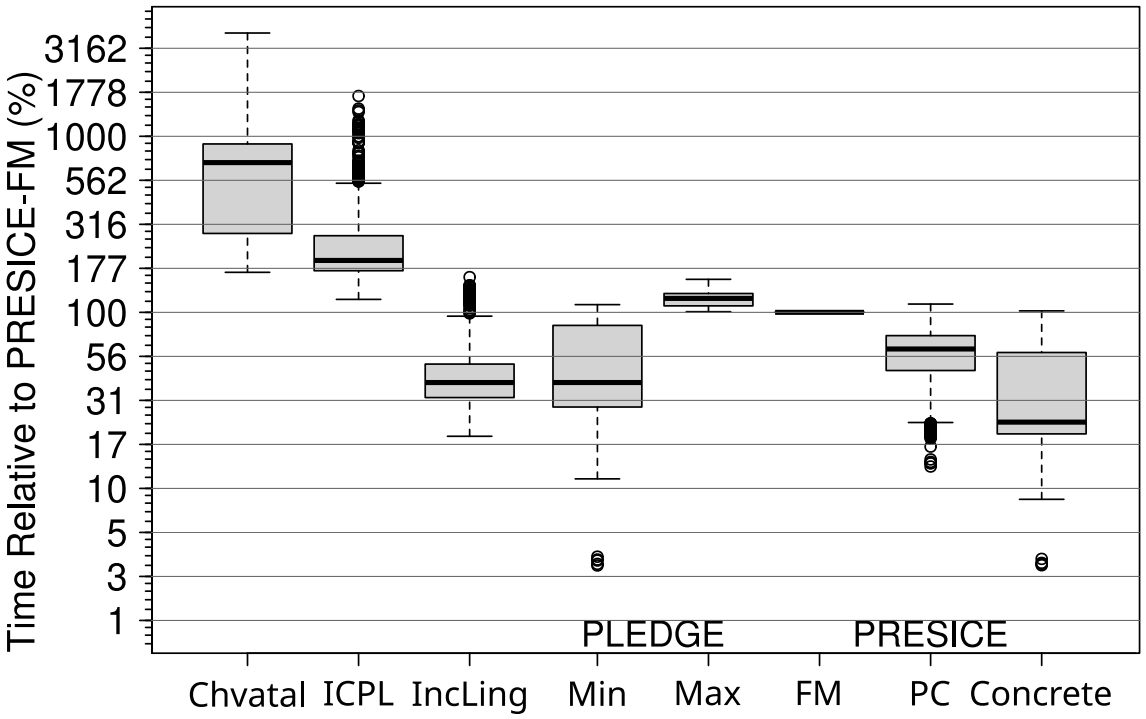

Figure 6: Sampling times of all algorithms relative to *PRESICE-FM* on the same system.

depict the sampling time for all algorithms in more detail in Figure 6 relative to the sampling time of PRESICE-FM. Here, we only show the pure sampling time, excluding the time needed for extracting presence conditions. As we described in Section 5, the extraction process is dependent on the employed variability mechanism, but only needs to be executed once, if the implementation artifacts do not change. In Table 7, we show for each system the absolute time in seconds required for PRESICE to extract the presence conditions. Additionally, in Table 7, we show the absolute values for the mean sampling time of PRESICE-FM and PRESICE-PC. We performed paired t-tests to test whether the difference in sampling time required by the different algorithms is significant. In Table 8, we show the results of these statistical tests by comparing the sampling time of all three variants of PRESICE with the sample size of all other algorithms. The symbol − indicates that the sampling time computed by the variant of PRESICE is significantly smaller than the sampling time of the other algorithm (i.e., $p < 0.001$). Analogous, the symbol + indicates the sampling time of PRESICE is significantly greater (i.e., $p < 0.001$).

We can see that the algorithms that consider presence conditions are significantly faster on average than some algorithms using only the feature model (ICPL, and Chvátal, and PLEDGE-Max). However, IncLing and PLEDGE-Min significantly outperform PRESICE-FM and are also on average significantly faster than PRESICE-PC. Regarding presence condition extraction time, we can see that in most cases it is similar to the sampling time. In case of Linux and BusyBox the extraction time even is substantially lower than the sampling time.

6.5 Discussion

Regarding RQ_1, we found in our experiments that samples with a 100% t-wise presence condition coverage were able to cover more faults than samples with 100% t-wise interaction coverage. These results indicate that t-wise presence condition coverage is indeed able to detect more faults than t-wise interaction coverage for the same value of t.

For RQ_2, we can see that only PRESICE-PC is able to guarantee a 100% t-wise presence condition coverage. Nevertheless, most of the other sampling algorithms, such as Chvátal, ICPL,

Table 8: Results of the paired t-tests for difference in sampling time between PRESICE and other algorithms.

Algorithm	**ICPL**	**Chvatal**	**IncLing**	**Pledge**	
				Min	*Max*
PRESICE-FM	−	−	+	+	−
PRESICE-PC	−	−	+	+	−
PRESICE-Concrete	−	−	−	−	−

IncLing, and PLEDGE achieve a high pair-wise presence conditions coverage (i.e., over 97% on average). While these results indicate an already good t-wise presence condition coverage with traditional sampling algorithms, it also shows that samples from these algorithms most likely miss some interactions between code blocks, which then cannot be tested. Furthermore, the high coverage of these algorithms might be due to the relatively large sample size compared to PRESICE-PC. Notably, random sampling with similar sample sizes also yields a similar coverage as the traditional sampling algorithms. In summary, it is possible for other algorithms to achieve a high t-wise presence condition coverage, though it cannot be guaranteed. However, the high t-wise presence condition coverage of the traditional sampling algorithms seems to be caused by their larger sample sizes.

Concerning RQ_3, we observe that for all systems, except `BusyBox`, PRESICE-PC generates smaller samples, than Chvátal, ICPL, IncLing, and PRESICE-FM. This may be caused by the relatively low number of concrete features for some systems, which facilitates the coverage of presence conditions within a configuration. The larger sample size of `BusyBox` may be caused due to a high number of mutually exclusive presence conditions. When considering presence conditions, we see that even for systems with more presence conditions than features (e.g., `axtls`, `uclibc-ng`, and `Linux`) PRESICE-PC is able to generate smaller samples. Moreover, the results of PRESICE-Concrete show that it is crucial to not just consider concrete feature, which yields smaller samples using t-wise interaction algorithms, but only reaches a PC coverage of about 67% on average. In summary, for most cases t-wise presence condition sampling produce relatively small samples, which may increase its testing efficiency.

Regarding RQ_4, we see that the sampling time of PRESICE-PC is relatively small and even outperforms some t-wise interaction sampling algorithms. The additional time for extracting presence conditions is similar to the sampling time for the smaller systems and even neglectable for larger systems such as `Linux`. In summary, the initial generation of samples with PRESICE-PC is only slightly less efficient than with traditional sampling algorithms due to the necessary extraction of presence conditions.

To conclude, with PRESICE we are able to efficiently generate relatively small samples for t-wise presence condition coverage. In addition, our results indicate that t-wise presence condition coverage is able to increase the fault-detection rate for product-based testing. Thus, we may be able to increase the testing efficiency and testing effectiveness by using t-wise presence condition sampling.

6.6 Threats to Validity

We are aware that our evaluation might suffer from some biases that may threaten its validity. First, we are using a rather small set of systems with feature models in our evaluation. Thus, the results might not scale to other systems with more features or presence conditions. However, we reused systems from other research and try to get a wide range of feature model sizes.

Second, we do not evaluate the actual testing effectiveness of our approach, but only compare it to other algorithms with respect to our own coverage criterion. As we proposed the coverage

criterion of t-wise presence condition coverage ourself, this may create an unfair comparison with other algorithms that aim for different coverage criteria. We do evaluate whether samples with 100% t-wise presence condition coverage are able to detect some known faults in several systems. However, as this is only a small set of faults, it may hamper the generalizability of our results. Nevertheless, we tried to include faults with varying degrees of interaction and complex presence conditions to mitigate this bias.

Third, we aim to evaluate the concept of t-wise presence condition sampling. However, we only use one particular sampling algorithm to cover interactions between presence conditions. It may be the case that the concept works better or worse when employing other sampling algorithms or different heuristics. Likewise, there is the chance of implementation bugs that may bias the result. To this end, we use automated tests to ensure that the samples we receive from PRESICE are valid and indeed achieve a t-wise presence condition coverage of 100%. Furthermore, we used one particular algorithm for extracting presence conditions from C preprocessor SPLs, which has limitations that may affect the correctness of the extracted presence conditions. However, the tool on which our algorithm is based employs three established tools for analyzing C preprocessor annotations in order to achieve more reliable results.

Finally, as we are using randomized features orders and random sampling, our results may be affected by a random bias. We tried to mitigate this by using multiple iterations of all experiments.

7 Related Work

There exist many different approaches to sampling an SPL [5, 31]. Like many other sampling algorithms, our sampling algorithm uses a greedy strategy to compute a minimal sample [8, 10, 14–16, 32–41]. In fact, our sampling algorithm is based on YASA [21], which is similar to the algorithm IPOG [42], as it also starts with an empty sample and iteratively adds literals. However, our sampling algorithm can cope with more general inputs, and thus is able to process arbitrary presence conditions. Furthermore, there also exist many algorithms that employ meta-heuristic strategies [7, 43–52] or other strategies [53] to cope with the variability explosion problem. These approaches can be seen as complementary to our concept, as we may also adapt meta-heuristic sampling to support presence conditions.

There already exist sampling strategies that consider other inputs in addition to a feature model [31]. Analogous to our approach, there are sampling algorithms that consider implementation artifacts [10, 38, 41, 54] and test artifacts [35, 36] to compute a sample. While these algorithms also consider the underlying SPL we are the first to combine presence conditions for implementation artifacts with regular t-wise interaction sampling to achieve higher effectiveness.

8 Conclusion & Future Work

With t-wise presence condition coverage, we present a new coverage criterion for SPLs that considers the actual implementation artifacts by looking at their presence conditions. Further, we describe a t-wise sampling algorithm that covers presence conditions instead of features and implement it for systems that use the C preprocessor. We test the fault-detection rate of t-wise presence condition coverage in comparison to t-wise interaction coverage. We also test our implementation by comparing it to existing sampling algorithms with regard to achieved coverage, sampling size, and sampling time. We find that t-wise presence condition coverage is able to detect more faults for a given t and that t-wise presence condition sampling produces mostly smaller samples compared to t-wise interaction sampling, while guaranteeing a 100% t-wise presence condition coverage. Therefore, we suspect that t-wise presence condition sampling

has the potential to increase both, testing effectiveness and testing efficiency.

Regarding future work, there are several research aspects, we would like to investigate further. To begin with, we want to investigate whether the results we achieved in our evaluation for testing effectiveness and efficiencies also scale to large systems with more or more complex presence conditions. In addition, we also want to evaluate the impact on testing effectiveness and efficiency, when grouping presence conditions as outlined in Section 5.2. Furthermore, there are many sampling strategies beside t-wise interaction sampling, for instance all-yes, all-no, 1-enabled, and 1-disabled. Each strategy uses a heuristic to trade-off testing efficiency against overall testing effectiveness. We can easily adapt our coverage criterion and sampling algorithm to consider these strategies for presence condition coverage instead of t-wise. Thus, we want to investigate, whether there is a benefit in using one of these strategies in combination with presence conditions.

Acknowledgments

We thank Tobias Heß for his valuable feedback.

References

[1] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

[2] John McGregor. "Testing a Software Product Line". In: *Testing Techniques in Software Engineering*. Springer, 2010, pp. 104–140.

[3] Emelie Engström and Per Runeson. "Software Product Line Testing - A Systematic Mapping Study". In: *J. Information and Software Technology (IST)* 53.1 (2011), pp. 2–13.

[4] Jihyun Lee, Sungwon Kang, and Danhyung Lee. "A Survey on Software Product Line Testing". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2012, pp. 31–40.

[5] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. "A Classification and Survey of Analysis Strategies for Software Product Lines". In: *ACM Computing Surveys* 47.1 (2014), 6:1–6:45.

[6] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. "Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints: A Greedy Approach". In: *IEEE Trans. Software Engineering (TSE)* 34.5 (2008), pp. 633–650.

[7] Dusica Marijan, Arnaud Gotlieb, Sagar Sen, and Aymeric Hervieu. "Practical Pairwise Testing for Software Product Lines". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2013, pp. 227–235.

[8] Iago Abal, Jean Melo, Stefan Stănciulescu, Claus Brabrand, Márcio Ribeiro, and Andrzej Wąsowski. "Variability Bugs in Highly Configurable Systems: A Qualitative Analysis". In: *Trans. Software Engineering and Methodology (TOSEM)* 26.3 (2018), 10:1–10:34.

[9] Christian Kästner, Sven Apel, Syed Saif ur Rahman, Marko Rosenmüller, Don Batory, and Gunter Saake. "On the Impact of the Optional Feature Problem: Analysis and Case Studies". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Software Engineering Institute, 2009, pp. 181–190.

[10] Reinhard Tartler, Christian Dietrich, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. "Static Analysis of Variability in System Software: The 90,000 #Ifdefs Issue". In: *Proc. USENIX Annual Technical Conference (ATC)*. USENIX Association, 2014, pp. 421–432.

[11] Sebastian Ruland, Lars Luthmann, Johannes Bürdek, Sascha Lity, Thomas Thüm, Malte Lochau, and Márcio Ribeiro. "Measuring Effectiveness of Sample-Based Product-Line Testing". In: *Proc. Int'l Conf. on Generative Programming and Component Engineering (GPCE)*. ACM, 2018, pp. 119–133.

[12] Richard M. Stallman and Zachary Weinberg. "The C preprocessor". In: *Free Software Foundation* (1987).

[13] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. "An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines". In: *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 2010, pp. 105–114.

[14] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. "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, 2011, pp. 638–652.

[15] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. "An Algorithm for Generating T-Wise Covering Arrays from Large Feature Models". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2012, pp. 46–55.

[16] Mustafa Al-Hajjaji, Sebastian Krieter, Thomas Thüm, Malte Lochau, and Gunter Saake. "IncLing: Efficient Product-line Testing Using Incremental Pairwise Sampling". In: *Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 2016, pp. 144–155.

[17] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[18] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[19] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines*. Springer, 2013.

[20] Elias Kuiter, Sebastian Krieter, Jacob Krüger, Kai Ludwig, Thomas Leich, and Gunter Saake. "PClocator: a tool suite to automatically identify configurations for code locations". In: *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. 2018, pp. 284–288.

[21] Sebastian Krieter, Thomas Thüm, Sandro Schulze, Gunter Saake, and Thomas Leich. "YASA: yet another sampling algorithm". In: *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 2020, 4:1–4:10.

[22] Mustafa Al-Hajjaji, Thomas Thüm, Malte Lochau, Jens Meinicke, and Gunter Saake. "Effective Product-Line Testing Using Similarity-Based Product Prioritization". In: *Software and System Modeling (SoSyM)* 18.1 (2019), pp. 499–521.

[23] Vasek Chvatal. "A Greedy Heuristic for the Set-Covering Problem". In: *Mathematics of operations research* 4.3 (1979), pp. 233–235.

[24] Mustafa Al-Hajjaji, Jens Meinicke, Sebastian Krieter, Reimar Schröter, Thomas Thüm, Thomas Leich, and Gunter Saake. "Tool Demo: Testing Configurable Systems with FeatureIDE". In: *Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 2016, pp. 173–177.

[25] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, Patrick Heymans, and Yves Le Traon. "Bypassing the Combinatorial Explosion: Using Similarity to Generate and Prioritize T-Wise Test Configurations for Software Product Lines". In: *IEEE Trans. Software Engineering (TSE)* 40.7 (2014), pp. 650–670.

[26] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. *Mastering Software Variability with FeatureIDE*. Springer, 2017.

[27] Daniel Le Berre and Anne Parrain. “The sat4j library, release 2.2, system description”. In: *Journal on Satisfiability, Boolean Modeling and Computation* 7 (2010), pp. 59–64.

[28] Jeho Oh, Paul Gazzillo, Don Batory, Marijn Heule, and Maggie Myers. *Uniform sampling from kconfig feature models*. Tech. rep. TR-19-02. The University of Texas at Austin, Department of Computer Science, 2019.

[29] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. “A Comparison of 10 Sampling Algorithms for Configurable Systems”. In: *Proc. Int’l Conf. on Software Engineering (ICSE)*. ACM, 2016, pp. 643–654.

[30] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wąsowski, and Krzysztof Czarnecki. “Reverse Engineering Feature Models”. In: *Proc. Int’l Conf. on Software Engineering (ICSE)*. ACM, 2011, pp. 461–470.

[31] Mahsa Varshosaz, Mustafa Al-Hajjaji, Thomas Thüm, Tobias Runge, Mohammad Reza Mousavi, and Ina Schaefer. “A Classification of Product Sampling for Software Product Lines”. In: *Proc. Int’l Systems and Software Product Line Conf. (SPLC)*. ACM, 2018, pp. 1–13.

[32] Alireza Ensan, Ebrahim Bagheri, Mohsen Asadi, Dragan Gasevic, and Yevgen Biletskiy. “Goal-Oriented Test Case Selection and Prioritization for Product Line Feature Models”. In: *Proc. Inte’l Conf. on Information Technology: New Generations (ITNG)*. IEEE, 2011, pp. 291–298.

[33] Evelyn Nicole Haslinger, Roberto E. Lopez-Herrejon, and Alexander Egyed. “Using Feature Model Knowledge to Speed Up the Generation of Covering Arrays”. In: *Proc. Int’l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 2013, 16:1–16:6.

[34] Martin Fagereng Johansen, Øystein Haugen, Franck Fleurey, Anne Grete Eldegard, and Torbjørn Syversen. “Generating Better Partial Covering Arrays by Modeling Weights on Sub-Product Lines”. In: *Proc. Int’l Conf. on Model Driven Engineering Languages and Systems (MODELS)*. Springer, 2012, pp. 269–284.

[35] Chang Hwan Peter Kim, Don Batory, and Sarfraz Khurshid. “Reducing Combinatorics in Testing Product Lines”. In: *Proc. Int’l Conf. on Aspect-Oriented Software Development (AOSD)*. ACM, 2011, pp. 57–68.

[36] Chang Hwan Peter Kim, Eric Bodden, Don Batory, and Sarfraz Khurshid. “Reducing Configurations to Monitor in a Software Product Line”. In: *Proc. Int’l Conf. on Runtime Verification (RV)*. Springer, 2010, pp. 285–299.

[37] Matthias Kowal, Sandro Schulze, and Ina Schaefer. “Towards Efficient SPL Testing by Variant Reduction”. In: *Proc. Int’l Workshop on Variability and Composition (VariComp)*. ACM, 2013, pp. 1–6.

[38] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. “Scalable Analysis of Variable Software”. In: *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 2013, pp. 81–91.

[39] Sebastian Oster, Florian Markert, and Philipp Ritter. “Automated Incremental Pairwise Testing of Software Product Lines”. In: *Proc. Int’l Systems and Software Product Line Conf. (SPLC)*. Springer, 2010, pp. 196–210.

[40] Dennis Reuling, Johannes Bürdek, Serge Rotärmel, Malte Lochau, and Udo Kelter. “Fault-Based Product-Line Testing: Effective Sample Generation Based on Feature-Diagram Mutation”. In: *Proc. Int’l Systems and Software Product Line Conf. (SPLC)*. ACM, 2015, pp. 131–140.

[41] Jiangfan Shi, Myra B. Cohen, and Matthew B. Dwyer. “Integration Testing of Software Product Lines Using Compositional Symbolic Execution”. In: *Proc. Int’l Conf. on Fundamental Approaches to Software Engineering (FASE)*. Springer, 2012, pp. 270–284.

[42] Yu Lei, Raghu N. Kacker, D. Richard Kuhn, Vadim Okun, and James Lawrence. “IPOG: A General Strategy for T-Way Software Testing”. In: *Proc. Int’l Conf. on Engineering of Computer-Based Systems (ECBS)*. IEEE, 2007, pp. 549–556.

[43] Anastasia Cmyrev and Ralf Reissing. “Efficient and Effective Testing of Automotive Software Product Lines”. In: *Int’l J. Applied Science and Technology (IJAST)* 7.2 (2014).

[44] Thiago N. Ferreira, Josiel Neumann Kuk, Aurora Pozo, and Silvia Regina Vergilio. “Product Selection Based on Upper Confidence Bound MOEA/D-DRA for Testing Software Product Lines”. In: *Proc. Congress Evolutionary Computation (CEC)*. IEEE, 2016, pp. 4135–4142.

[45] Faezeh Ensan, Ebrahim Bagheri, and Dragan Gasevic. “Evolutionary Search-Based Test Generation for Software Product Line Feature Models”. In: *Proc. Int’l Conf. on Advanced Information Systems Engineering (CAiSE)*. Vol. 7328. Springer, 2012, pp. 613–628.

[46] Thiago N. Ferreira, Jackson A. Prado Lima, Andrei Strickler, Josiel N. Kuk, Silvia R. Vergilio, and Aurora Pozo. “Hyper-Heuristic Based Product Selection for Software Product Line Testing”. In: *Comp. Intell. Mag. (CIM)* 12.2 (2017), pp. 34–45.

[47] Helson L. Jakubovski Filho, Jackson A. Prado Lima, and Silvia R. Vergilio. “Automatic Generation of Search-Based Algorithms Applied to the Feature Testing of Software Product Lines”. In: *Proc. Brazilian Symposium on Software Engineering (SBES)*. ACM, 2017, pp. 114–123.

[48] Christopher Henard, Mike Papadakis, and Yves Le Traon. “Mutation-Based Generation of Software Product Line Test Configurations”. In: *Search-Based Software Engineering*. Ed. by Claire Le Goues and Shin Yoo. Vol. 8636. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 92–106.

[49] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. “Multi-Objective Test Generation for Software Product Lines”. In: *Proc. Int’l Systems and Software Product Line Conf. (SPLC)*. ACM, 2013, pp. 62–71.

[50] Roberto Erick Lopez-Herrejon, Javier Ferrer, Francisco Chicano, Alexander Egyed, and Enrique Alba. “Comparative Analysis of Classical Multi-Objective Evolutionary Algorithms and Seeding Strategies for Pairwise Testing of Software Product Lines”. In: *Proc. Congress Evolutionary Computation (CEC)*. IEEE, 2014, pp. 387–396.

[51] Rui Angelo Matnei Filho and Silvia Regina Vergilio. “A Multi-Objective Test Data Generation Approach for Mutation Testing of Feature Models”. In: 4.1 (2016).

[52] Xavier Devroey, Gilles Perrouin, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. “Covering SPL Behaviour with Sampled Configurations: An Initial Assessment”. In: *Proc. Int’l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 2015, 59:59–59:66.

[53] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. “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, 2010, pp. 459–468.

[54] Reinhard Tartler, Daniel Lohmann, Christian Dietrich, Christoph Egger, and Julio Sincero. “Configuration Coverage in the Analysis of Large-Scale System Software”. In: *ACM SIGOPS Operating Systems Review* 45.3 (2012), pp. 10–14.