Continuous T-Wise Coverage

Tobias Pett
tobias.pett@kit.edu
Karlsruhe Institute for Technology (KIT)
Germany

Tobias Heß
tobias.hess@uni-ulm.de
University of Ulm
Germany

Sebastian Krieter
sebastian.krieter@uni-ulm.de
University of Ulm
Germany

Thomas Thüm
thomas.thuem@uni-ulm.de
University of Ulm
Germany

Ina Schaefer
ina.schaefer@kit.edu
Karlsruhe Institute for Technology (KIT); KASTEL
Germany

ABSTRACT

Quality assurance for highly configurable systems uses t-wise feature interaction coverage as a metric to measure the quality of selected samples for testing. Achieving t-wise feature interaction coverage requires testing many configurations, often exceeding the available testing time for frequently evolving systems. As testing time is a limiting factor, current testing procedures face the challenge of finding a reasonable trade-off between achieving t-wise feature interaction coverage and reducing the time required for testing. To address this challenge, we can consider t-wise feature interactions covered in previous test executions when calculating the achieved t-wise feature interaction coverage. However, the current definition of t-wise feature interaction coverage does not consider previously tested configurations. Therefore, we propose continuous t-wise coverage as a new customizable metric for tracking the ratio of achieved t-wise feature interaction coverage over time. Our metric allows customizing the tradeoff between test effort per system version and the time to achieve t-wise coverage. We evaluate various parameterizations for our metric on four real-world evolution histories and investigate how they impact the calculated t-wise feature interaction coverage. Our results show that a high t-wise feature interaction coverage can be achieved by testing significant (up to 50%) smaller samples per commit, when the evolution of the system is considered.

CCS CONCEPTS

• **Software and its engineering** → *Software product lines.*

KEYWORDS

t-wise coverage, software-product lines, spl testing, spl evolution, sampling

ACM Reference Format:
Tobias Pett, Tobias Heß, Sebastian Krieter, Thomas Thüm, and Ina Schaefer. 2023. Continuous T-Wise Coverage. In *27th ACM International Systems and Software Product Line Conference - Volume A (SPLC '23), August 28-September 1, 2023, Tokyo, Japan.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3579027.3608980

1 INTRODUCTION

Safety-critical software systems are highly configurable as well as frequently evolving. Many leading OEMs use the open-source platform Automotive Grade Linux (AGL), which is based on the Linux kernel, to develop in-vehicle software.[1] The Linux foundation measured more than 75,000 commits to the kernel repository in 2019, giving rise to about 250 commits per day on average.

Typically, the evolution of a large system, such as the Linux kernel, includes four phases (development, commit, integration, and testing) [29, 35]. The process of evolving a system starts with a developer changing the problem- or the solution space of the system (i.e., development) and committing the changes to a repository (i.e., commit). Integrating the developer's commit into the system creates a new system version (i.e., integration).

Testing is an important method to assure functional safety and is commonly executed after each commit [2, 27]. However, time and resources for testing are limited in practice, especially for rapidly evolving systems. For instance, with 250 daily commits, the test execution per commit may only take less than 6 minutes (i.e., $\frac{24 \cdot 60}{250} = 5.7$). This often mandates a trade-off between fast and resource-efficient test cycles (i.e., low testing time and testing costs) and sufficient system coverage [8, 9, 29, 35]. Configurable systems compound this conflict of interest, as the tests need to be executed on multiple configurations. As even tiny systems possess too many configurations to feasibly execute the tests on all configurations of even a single commit (e.g., JHipster with 48 features and 15 cross-tree constraints required 182 days of computing time to test all 26,256 valid configurations [10]), the aforementioned trade-off also needs to take the number of tested configurations into account.

Sample-based testing [18, 28, 36] mitigates the challenge posed by the combinatorial explosion by selecting a small and representative subset of product configurations (i.e., a sample) for testing. A product configuration consists of selected configuration options

[1]https://linuxfoundation.org/wp-content/uploads/2021_LF_Annual_Report_010222.pdf

(i.e., features) [3]. Covering all possible feature tuples of strength t (i.e., t-wise feature interaction coverage) is one prominent criterion to determine that a sample is representative [7, 26]. A sample must contain all possible combinations of t-tuples of features to achieve t-wise feature interaction coverage. For instance, to achieve pair-wise feature interaction coverage (i.e., $t = 2$), all valid combinations of two features must be present in at least one configuration in the sample. A number of t-wise sampling algorithms exist to calculate samples that achieve t-wise feature interaction coverage in a short amount of time [36]. However, for most real-world systems, state-of-the-art samplers only scale to $t \leq 3$ [15].

However, samples that achieve 100% t-wise feature interaction coverage for large configurable systems often contain hundreds of configurations, rendering sample-based testing of each commit infeasible [33]. One recently introduced sampling algorithm, YASA [15], calculates a pair-wise coverage sample consisting of 545 configurations for Linux version 2.6.28, which contains 6,889 features, in about 30 minutes. Still, spending half an hour calculating the sample and executing test cases on hundreds of configurations for each commit is not feasible for a system such as the Linux kernel. Thus, practitioners do not use the potential of systematic sample-based testing. Instead, they often choose a small number of configurations based on expert knowledge, random selection, or a naive heuristic as a test sample (S). In the case of the Linux Kernel, a set of predefined and a set random configurations are tested per commit. [2] This unsystematic sample-based testing process leads to an inefficient and unevenly distributed coverage of t-wise feature tuples, while they should be tested equally to reduce the risk of faulty and unforeseen feature interactions.

Tested configurations of previous commits contain knowledge about covered t-wise feature tuples. We can use this knowledge to incrementally guide configuration selection in highly time and resource-constrained test settings. Uncovering the number of already covered feature tuples is a prerequisite to using this knowledge for efficient configuration selection and prioritization. The currently established metric for calculating t-wise feature interaction coverage considers only the most recent commit when calculating the achieved coverage. In the remainder, we will therefore refer to the established metric as *one-shot coverage*.

The goal of this paper is to introduce the concept of accumulating covered t-wise feature tuples over time to increase the achieved t-wise feature interaction coverage throughout the evolution of a system. To quantify our approach, we define *continuous t-wise coverage* as a new metric to compute coverage over time. Our metric is not an approach used to compute samples, instead its purpose is to measure the achieved coverage of samples computed for successive system versions. We introduce the commit window size (n) as a parameter to adapt how many commits from the system's evolution are considered when calculating the achieved continuous t-wise feature interaction coverage. For instance, by setting the window size to five commits ($n = 5$), the tested configurations of the most recent five commits are considered when calculating t-wise feature interaction coverage. By setting $n = 1$, our metric emulates one-shot coverage. We evaluate our metric on four Kconfig-based systems[3], such as BusyBox, Fiasco, Soletta, and Uclibc, for which an extensive history is known. Our evaluation aims to show the benefits of considering previously tested configurations, under consideration of problem-space changes (i.e., feature model changes), to guide test strategies for a new system version. We also discuss the trade-off between the number of tested configurations and the size of the commit window (CW) to reach the required coverage, as this is of great relevance for the approach to be feasible in practice. In summary, we make the following contributions:

- We propose the metric *continuous t-wise coverage* to measure t-wise feature interaction coverage over time.
- We provide an implementation of our metric as open-source command-line tool.[4]
- We evaluate continuous t-wise coverage on evolution histories of real-world configurable systems[5]

2 BACKGROUND

This section provides background information about the sample-testing process of configurable systems. First, we elaborate on managing variability of configurable systems in Section 2.1. In Section 2.2, we provide essential information on how configurations for sample-based testing are typically generated.

2.1 Managing Variability

A configurable system consists of a set of possible products, which share a set of common and varying properties, captured in a feature model [3, 14]. The feature model of a configurable system manages variability by specifying all possible configuration options (features) and their dependencies [3, 6, 14, 34]. It is the representation of the problem space of a configurable system, and an essential part of each system version (i.e., commit of the system). A feature model $\mathcal{M} = (\mathcal{F}, \mathcal{D})$ is a tuple consisting of a set of features $\mathcal{F}$ and a set feature dependencies $\mathcal{D}$. We define the features of a feature model as $\mathcal{F} = \{f_0, f_1, \ldots, f_o\}$ where f_i represents a Boolean variable with a unique feature name and o is the total number of features in the feature model dependencies $d \in \mathcal{D}$ between features are typically represented by propositional formulas in conjunctive normal form (CNF) notation. We denote the set of all dependencies in the feature model as $\mathcal{D} = \{d_0, d_1, \ldots, d_p\}$ where p is the total number of dependencies contained in the feature model.

Feature models are often visualized as feature diagrams to increase human readability [4, 14]. A feature diagram orders features in a tree structure based on their dependencies. Figure 1 shows the feature diagram of a simplified car system, which we use as our running example. The feature model of our running example is inspired by the car example feature model[6] of FeatureIDE[7]. It consists of the following eleven features: `Car`, `Carbody`, `Radio`, `Gearbox`, `Ports`, `Navigation`, `Bluetooth`, `Manual`, `Automatic`, `USB`, `CD`. Throughout the paper, we use a shorthand notation which represents features

[2] https://www.linuxjournal.com/article/7445; https://people.netfilter.org/hawk/presentations/ifdef2016/ifdef_FOSD2016.pdf

[3] https://www.busybox.net/; https://github.com/kernkonzept/fiasco; https://github.com/solettaproject/soletta; https://github.com/wbx-github/uclibc-ng/

[4] https://doi.org/10.5281/zenodo.8043550

[5] https://doi.org/10.5281/zenodo.7885905

[6] https://github.com/FeatureIDE/FeatureIDE/tree/develop/plugins/de.ovgu.featureide.examples/featureide_examples/FeatureModels/Car

[7] https://featureide.github.io/

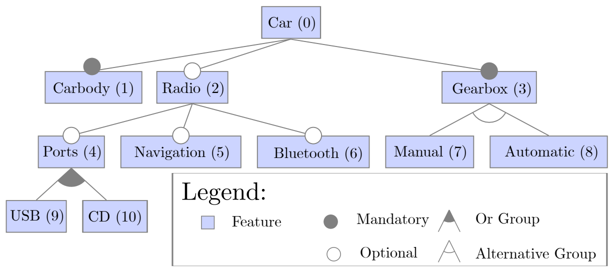


Figure 1: Feature diagram of a simplified car system. Inspired by FeatureIDE car example feature model.

by integer values. Feature names are directly mapped to integer values by enumerating the features in the feature model. Figure 1 indicates the mapping for our running example using the numbers in brackets. For instance, feature `Car` is mapped to 0, `Carbody` to 1, and so on until feature `CD` which is mapped to literal 10.

A feature diagram captures the dependencies between features through their edge types and propositional formulas below the diagram (e.g., $Navigation \implies USB$), so-called cross-tree constraints. Our running example contains four different edge types to model the dependencies, mandatory feature (i.e., filled circle), optional feature (i.e., empty circle), OR-group (i.e., filled arch), and alternative-group (i.e., empty arch). A mandatory feature, such as `Carbody` must always be true when its parent feature (e.g., `Car`) is true, while optional features, such as `Radio` can be true when its parent feature (e.g., `Car`) is true. The mandatory dependency between `Car` and `Carbody` can be written as $\neg Car \vee Carbody$ in CNF, or as $\{\neg 0, 1\}$ in a shorthand notation. The dependency between the features `Ports`, `USB`, and `CD` is called an Or-group. It describes that if the parent feature (i.e., `Ports`) is true at least one of the features `USB`, or `CD` must be true, and it can be written as $\{\neg 4, 9, 10\}$ in our shorthand notation. The features `Gearbox`, `Manual`, and `Automatic` are modeled as alternative-group, which means that if `Gearbox` is true either `Manual`, or `Automatic`, but not both simultaneously must be true. It can be expressed as $\{(\neg 3, 7, \neg 8); (\neg 3, \neg 7, 8)\}$ in shorthand notation.

2.2 Configuration Testing

Users of a configurable system can build configurations by selecting (i.e., setting a feature to true) and deselecting features (i.e., setting a feature to false). A configuration represents a variant of the configurable system, which can be derived using a variability realization mechanism [3, 6, 34]. We define a configuration C as the set of selected and de-selected features of a feature model. For instance, a users can create a configuration (C_1) of our running example by selecting `Car`, `Carbody`, `Gearbox`, `Manual` and deselecting all other features. In our shorthand notation we express the example configuration as $C_1 = \{ 1, \neg 2, 3, \neg 4, \neg 5, \neg 6, 7, \neg 8, \neg 9, \neg 10 \}$.

A configuration is called *valid*, if it conforms to the constraints defined by the corresponding feature model $\mathcal{M}(\mathcal{F}, \mathcal{D})$. Throughout our paper, we use the notation $C(\mathcal{M})$ to describe the set of valid configuration for feature model $\mathcal{M}(\mathcal{F}, \mathcal{D})$.

Real-world systems often provide many features, which results in an enormous number of possible valid configurations, due to the combinatorial explosion problem [18, 28, 36], and testing all of those is not feasible. Instead, sample-based testing is often used to verify the functional safety of a configurable system by testing a representative subset of all valid configurations (i.e., a sample). We define a sample as $S \subseteq C$. Selecting a sample to test a configurable system can be done by sampling algorithms, such as random sampling [30], or t-wise interaction sampling [5, 7, 12, 15, 24, 31]. One prominent criterion to rate the quality of a sample is the t-wise feature interaction coverage a sample achieves [36].

A t-wise feature interaction $(f_1 \dots f_t)$ is a tuple of size t, that contains selected or deselected features from the permutation set $2^{\mathcal{F}}$ of all features in the feature model. For instance, $\{ (\neg 1, 2); (1, 3); (\neg 2, 3) \}$ are examples for pair-wise ($t = 2$) feature tuples for our running example. We call a feature combination of size t valid if it can appear in at least one valid configuration $C(\mathcal{M})$ of the feature model. According to our definition of valid feature combinations, the tuple $(\neg 1, 2)$ is not a valid feature interaction for our running example because it violates the dependency that the feature `Carbody` must always be selected. We define the set of valid t-wise feature interactions $(\mathcal{I}(t, \mathcal{M}))$ for a feature model $(\mathcal{M})$ by $\mathcal{I}(t, \mathcal{M}) = \{ (f_1, \dots f_t) \mid (f_1, \dots f_t) \in C(\mathcal{M}) \}$

A sample achieves full (i.e., 100%) t-wise feature interaction coverage when all valid feature combinations are contained in at least one configuration of the sample. Not all sampling algorithms generate samples that achieve full t-wise feature interaction coverage. The set of valid t-wise feature interactions contained in a sample is defined by $\mathcal{I}(t, \mathcal{M}, S) = \{ (f_1, \dots f_t) \mid \text{there exists } C \in S \text{ such as } (f_1, \dots f_t) \subseteq C \}$. The ratio of achieved t-wise feature interaction coverage is calculated by dividing the number of t-wise feature interactions contained in a sample by the number of all valid t-wise feature interactions of the feature model.

$$\text{coverage}\ (t, \mathcal{M}, S) = \frac{|\mathcal{I}(t, \mathcal{M}, S)|}{|\mathcal{I}(t, \mathcal{M})|} \tag{1}$$

3 PROBLEM STATEMENT

Achieving t-wise feature interaction coverage over time by accumulating covered feature interactions for a number of previous commits (commit window) may reduce testing effort for a commit. We show this potential by a thought experiment.

Thought Experiment. Assume we have our running example from Figure 1, for which we want to ensure functional safety during the system's evolution. Typically, the system developers commit three times a day, but the feature model is not changed. The system needs to be tested with pair-wise feature interaction coverage for each commit. Using the YASA sampling algorithm [15], we calculate a pair-wise feature interaction sample containing seven configurations in less than 20 seconds. However, we only have time to test three configurations per commit. So, we cannot achieve full pairwise feature interaction coverage in one commit because we do not have the testing resources to fully test seven configurations. However, if we soften the requirement of achieving pair-wise coverage in every commit slightly to achieving coverage once per day, this becomes feasible, by splitting the seven configurations so that each is tested at least once after three commits.

Table 1: Configurations tested per commit of the evolution history of the running example.

commit	configuration	
commit 0	c1	$\{ 0, 1, \neg 2, 3, \neg 4, \neg 5, \neg 6, 7, \neg 8, \neg 9, \neg 10 \}$
	c2	$\{ 0, 1, 2, 3, 4, 5, 6, 7, \neg 8, 9, \neg 10 \}$
	c3	$\{ 0, 1, \neg 2, 3, \neg 4, \neg 5, \neg 6, \neg 7, 8, \neg 9, \neg 10 \}$
commit 1	c4	$\{ 0, 1, 2, 3, 4, 5, 6, 7, \neg 8, \neg 9, 10 \}$
	c5	$\{ 0, 1, 2, 3, 4, \neg 5, \neg 6, \neg 7, 8, 9, 10 \}$
	c6	$\{ 0, 1, 2, 3, \neg 4, \neg 5, 6, \neg 7, 8, \neg 9, \neg 10 \}$
commit 2	c7	$\{ 0, 1, 2, 3, \neg 4, 5, \neg 6, \neg 7, 8, \neg 9, \neg 10 \}$
	c1	$\{ 0, 1, \neg 2, 3, \neg 4, \neg 5, \neg 6, 7, \neg 8, \neg 9, \neg 10 \}$
	c2	$\{ 0, 1, 2, 3, 4, 5, 6, 7, \neg 8, 9, \neg 10 \}$

Table 1 shows the mapping between commits and tested configurations. The first column of the table shows the numbered commits from one to three, while the second column shows configurations with configuration names one to seven and the set of features contained by the configuration. According to our testing requirements we can test three configurations per commit. In commit 1 and 2, we test three configurations each. So that, we need to test only one of those seven configurations in commit 3, which leaves testing resources for retesting configurations (e.g., c1 and c2). Table 1 indicates that configurations c1 and c2 will be retested after the coverage threshold is reached. This is an example to illustrate how retesting of configurations can be done. In a real test scenario, the order of retesting configurations could be adjusted appropriately. Table 1 shows that it is possible to split a t-wise interaction sample over multiple commits to achieve t-wise feature interaction coverage after the third commit. In practice, however, calculating a sample in advance and distributing the configurations of that sample across multiple commits is not useful due to changes in the problem space and solution space. Instead, samples of the required size are created for each commit individually. Calculating how much t-wise feature interaction coverage is achieved by samples of previous system versions is not possible with the currently established one-shot metric, at least not without addressing the following questions: 1) For which feature model do we compute the t-wise feature interaction coverage? 2) Which feature tuples are still valid for the chosen system version? 3) How can we make sure that feature tuples are not considered multiple times, thus inflating the t-wise interaction coverage? With continuous t-wise coverage, we present a metric that addresses these questions and can that be applied to an evolution history of a configurable system.

However, simply accumulating t-wise feature interaction coverage over time is not yet sufficient. In a real-world evolution scenario, there are significant challenges that must be addressed by a well defined metric. For example, many of the current testing procedures focus on one-shot coverage, which should not be ignored by the new metric. Therefore, we require *equivalence (Req.1)* between one-shot coverage and continuous t-wise coverage to allow a seamless conversion of current procedures. Depending on the nature and structure of the system and development process, real-world testing scenarios differ strongly wrt. quality demands and testing time. For example, testing a volatile system with many small changes to the problem and solution space in a short period of time will invalidate previously covered feature tuples on each commit, and will likely not benefit from a large commit history. Instead, it will be more beneficial for these systems to test more new configurations, increasing the testing time per commit. On the other hand, for a stable system with few changes to the problem and solution space, covered feature tuples will remain valid for many commits in the commit history. Therefore, testing these systems will benefit from considering longer commit histories and testing fewer configurations in each commit to spread the testing effort over time. Therefore, we require that continuous t-wise coverage must be *adaptable (Req.2)* wrt. the number of configurations tested in one commit, and the number of commits that are used to achieve t-wise feature interaction coverage over time. During the evolution of configurable systems feature tuples may be covered by different configurations or become invalid because of changes to the problem or solution space of the system. We require *tuple uniqueness (Req.3)* and *robustness (Req.4)* when calculating t-wise feature interaction coverage, so that duplicate and invalid feature tuples do not distort the result of continuous t-wise coverage.

4 CONTINUOUS T-WISE COVERAGE METRIC

In this section, we define continuous t-wise coverage as an extension of t-wise coverage and argue, how our definition satisfies the requirements of Section 3.

4.1 Defining Commit and Sample Histories

We capture a systems' evolution by its commit history $\mathcal{H}$. A commit *comm* represents all changes done by a developer to create a new system version. We define the commit history as list of commits $\mathcal{H} = \left[\, comm_0, \ldots, comm_h \, \right]$, which describes all changes to the system throughout its history. Naturally, we define the length of the commit history $h = |\mathcal{H}|$ as the number of commits contained in the commit history. The first element in the list (i.e., $comm_0$) is the earliest commit of the system, while the last element in the list (i.e., $comm_h$) represents the most recent commit of the system. In our running example (see Section 3), we have a commit history containing three commits, starting with `commit 0` at index $comm_0$ and ending with `commit 2` at index $comm_2$.

In sample-based testing, tests are executed on a number of selected system configurations (i.e., a sample S). Over the systems' evolution, a history of tested samples is built, which we define as the sample history $SH = \left[\, S_0, \ldots, S_h \, \right]$ of the system. In case of our running example, SH contains three samples, i.e., $SH = \left[\, \{c1, c2, c3\}, \{c4, c5, c6\}, \{c7, c1, c2\} \, \right]$. A sample is tested for each commit of the system. We define the size of SH to be equal to the the length of $\mathcal{H}$. Further, we got a one-to-one mapping between the elements of $\mathcal{H}$ and SH. Using this mapping, we can query a sample $S_h \in SH$ from the sample history by a commit $comm_h \in \mathcal{H}$ from the commit history.

4.2 Defining Continuous T-Wise Coverage

As defined in Equation 1, t-wise interaction coverage is measured as the ratio between feature tuples contained in the configurations of a sample and all possible feature tuples of the respective commit. Our new metric, *continuous t-wise coverage*, extends this definition of by considering a sequence of samples (i.e., a sample window SW)

instead of a single sample. The size of this sequence (i.e., the size of the sample window) can be adapted to any number of available samples in the sample history SH. Equation 2 shows the resulting formula to calculate continuous t-wise coverage.

$$\begin{aligned} \text{contcoverage}\ (t, \mathcal{M}, SW) &= \frac{|\mathcal{I}(t, \mathcal{M}, \cup_{S \in SW} S)|}{|\mathcal{I}(t, \mathcal{M})|} \\ &= \text{coverage}\ (t, \mathcal{M}, \cup_{S \in SW} S) \end{aligned} \tag{2}$$

Recall that the function I only considers valid and unique interactions. Hence, per definition all invalid feature tuples caused by problem space changes are excluded from the calculation of continuous t-wise coverage so that continuous t-wise coverage satisfies the robustness requirement (Req.4). As the function I is defined using sets, computing the union of samples from the sample window yields itself a set of unique configurations, maintaining a correct number of covered interactions, even if they are covered by multiple sample configurations in different samples (i.e., continuous t-wise coverage satisfies Req.3) Furthermore, as the union of samples in the history constitutes a sample itself, it holds that the continuous coverage of a sample window SW of size one equals the classic coverage, which satisfies the first requirement Req.1.

4.3 Customization Options

An application of continuous t-wise coverage is to support the efficient usage of testing resources by determining how many configurations must be tested per commit for achieving t-wise feature interaction coverage in a fixed number of commits. We define the parameters window size n and sample size m to make continuous t-wise coverage adaptable for differing testing requirements.

We introduce the commit window CW to adjust the number of considered commits when calculating continuous t-wise coverage. The commit window CW is a sub-sequence of the commit history. We define the size of the commit window $n = |CW|$ as the number of commits the commit window contains. All commits in the commit window are ordered as a list of elements $CW = \left[comm_h, \ldots, comm_{h-n} \right]$, starting with the most recent commit of the commit history $comm_h$ and ending with the n-th most recent commit counting backwards. Similar we define the sample window $SW = \left[S_h \ldots S_{h-n} \right]$. In our running example, the commit window CW is equal to its commit history $\mathcal{H}$. However, we can imagine another example where only two commits (i.e., $n = 2$) in the running example are considered to calculate continuous t-wise coverage. In this case, the commit window contains the third and the second commit $CW = \left[\text{commit2, commit1} \right]$ and the respective sample window contains the configurations $SW = \left[\{c1, c2, c7\}, \{c4, c5, c6\} \right]$.

We define the number of configurations of a sample S, by m. With this sample size parameter, we can choose how many configurations of a sample are tested in each commit. Consequently, $m = 3$ for our running example. The commit window size n and the sample size m, provide the freedom to experiment with various strategies to achieve t-wise feature interaction coverage over time. Hence, our metric can be applied flexibly to various testing scenarios, which satisfies the adaptability requirement Req.2.

5 EVALUATION

RQ1: How does continuous coverage compare to one-shot coverage over time? In RQ1, we measure how the accumulation of t-wise feature tuples influences the t-wise feature interaction coverage throughout the system's evolution. In particular, we are interested in analyzing the trend of continuous t-wise coverage compared to one-shot coverage. As part of this analysis, we answer the following sub-research questions: RQ1.1) Does continuous t-wise coverage reach coverage values comparable to one-shot coverage values, even if a significantly smaller sample size is used? Our expectation for this sub-research question is that continuous t-wise coverage achieves at least the same t-wise feature interaction coverage as one-shot sampling, when the same sample size is used. If a smaller sample size compared to one-shot sampling is used, we expect to see a ramp-up phase for the first n commits, where n is the size of the chosen commit window. We expect that continuous t-wise coverage outperforms one-shot coverage after this ramp-up phase even for significantly smaller sample sizes. RQ1.2) After how many commits does continuous t-wise coverage achieve 98% of covered feature interactions? After the ramp-up phase, we expect that the achieved coverage will further increase until a threshold is reached. Ideally this threshold will be at 100% t-wise feature interaction coverage, but depends on the parameter configuration of the sample size m and the commit window size n. We will analyze various parameter combinations for different subject systems, to evaluate which parameter combinations achieve 98% covered feature interactions after how many commits. RQ1.3) How strongly do changes in the feature model influence continuous t-wise coverage? One factor that influences which t-wise feature interaction coverage can be achieved by continuous t-wise coverage are the changes in the problem space (i.e., feature model changes) between commits. Changing the feature model will influence the number of t-wise feature tuples which need to be covered, by either increasing their number (i.e., adding features, or removing restricting constraints) or decreasing their number (i.e., removing features, or adding restrictive constraints). Our expectation is that in general continuous t-wise coverage will be resilient against feature model changes. A strong increase of feature tuples to be covered (i.e., adding new feature, removing restrictive constraints) will result in a temporal decrease of the calculated continuous t-wise coverage, because we expect that the sample size m is not large enough to cover the newly added tuples in one commit. A strong decrease of feature tuples to be covered (i.e., removing features, or adding restrictive constraints) will not affect the continuous t-wise coverage metric, because previously covered feature tuples will be invalidated, but there will also be less feature tuples to be covered.

RQ2: What is the trade-off between test effort per commit and time to achieve continuous t-wise interaction coverage? With this research question, we analyze how the parameters sample size (m) and commit window size (n) influence the increase of t-wise feature interaction coverage over time. Based on this analysis, we achieve an estimation of trade-offs for different testing scenarios. Theoretically, the ideal trade-off between sample size (m) and commit window size (n) is to achieve high t-wise feature interaction coverage with low values for m and n. However, the actual best trade-off between both parameters may depend on the testing scenario. We

Table 2: Subject Systems with the number of commits, features, dependencies, changes of the variability model throughout their commit history.

System	#Commits	#Feat.	#Dep.	#Changes
Fiasco	33	253	1795	2
Soletta	173	457	2319	153
Uclibc	177	235	1905	142
BusyBox	3713	631	1312	249

measure continuous t-wise coverage for various permutations of sample size and commit window size to analyze the trade-off between them. As a result, we expect to find that a larger commit window is necessary to achieve higher t-wise feature interaction coverage when the sample size is reduced and vice versa.

5.1 Experiment Setup

In the following, we describe the setup of our experiments, starting with selecting the subject systems and their commit histories.

Subject Systems. We use the commit histories of BusyBox, Fiasco, Soletta, and Uclibc as subject systems in our evaluation. Those are highly configurable and frequently evolving systems from real-world applications, which use the kconfig language to model the system's variability. They are frequently used to evaluate newly developed concepts in the domain of product-line research [15, 32] because they represent small and middle-sized configurable systems. Table 2 shows details of the most recent commits from the subject systems used in our experiment. Even though the most recent commit does not reflect the whole system evolution, it is still an indicator of the system's size and complexity. The system details for all commits can be found in our data package [8] The first column of Table 2 shows the system's name. The second column shows how many system commits were used in our experiments. Column three shows the number of features for the most recent commit of each subject system, while column four shows the number of dependencies. The fifth column shows the number of commits that change the variability model of the respective subject system. To identify which commit changes the variability model, we analyze the uniqueness of each system version based on a hash value as indicator. We use a hashing algorithm to build a checksum value for each version of the variability model in the evolution history of our subject systems. We then compare the checksum values of consecutive versions against each other. If the checksum differs, we mark the newer variability model version as changed.

Acquiring Samples. Continuous t-wise coverage can be calculated as long as the underlying sampling approach generates a set of configurations for the commits of a system. It is therefore irrelevant which underlying sampling approach is used to generate the samples. However, continuous t-wise coverage is affected by the underlying sampling strategy. For example, using a t-wise sampling algorithm to create samples for each commit of the system will result in higher continuous t-wise coverage values because each configuration in the commit window *n* contains specifically selected t-wise feature interaction tuples.

In our evaluation, we aim to reduce the influence of the underlying sampling approach in order to obtain meaningful results. Therefore, we use uniform random sampling to generate the samples for the evaluation. Uniform random sampling considers each configuration from the configuration space equally likely. Therefore, the distribution of the configuration space is reflected in the sample, and no biased algorithm optimizations are introduced. However, uniform random sampling does not guarantee that a generated sample achieves 100% feature interaction coverage and may introduce a bias based on the randomly selected configurations. We repeat our experiments ten times to counter the bias of randomly selected configurations. Not achieving 100% t-wise feature interaction coverage using a single sample is irrelevant in our experiments because we are interested in investigating how continuous t-wise coverage behaves over time and compares to one-shot coverage.

Configuration Selection. For each subject system, we select a sample of size *m* for each commit of its evolution history. The sample size *m* is an essential parameter for our experiments because it strongly influences how fast continuous t-wise coverage may achieve a certain ratio of t-wise feature interaction coverage. Therefore, we perform multiple experiments with different sample sizes. The chosen sample sizes allow us to analyze the behavior of continuous t-wise sampling on a broad scale, starting with a detailed analysis of small differences in the sample size and the possibility of analyzing the effects of large differences in the sample. We use uniform random sampling to generate the samples with different sample sizes. The random selection of configurations by uniform random sampling may influence our results because each configuration selected after the initial configuration may cover a different ratio of t-wise feature interaction tuples. As mentioned before, we mitigate the influence of randomness in our results by executing the experiments ten times. Each of those executions generates an intermediate result. We build the arithmetic mean value of the intermediate results for our discussion.

Evaluation Hardware. We perform our experiments on a virtual server running an Ubuntu 20.04 operating system. The server has a virtual processor with eight cores running at 2400MHz. The server's physical memory is 16 GB, from which we use 12 GB as virtual memory to execute our experiments. The experiments are executed as Java command-line tools integrating functionality from the FeatureIDE library. For running our command-line tool, we use OpenJDK version 1.8.0_292. We provide our command tooling online as open source package [9]. *We are unable to provide our open source tooling for the double blind review, because details about the authors maybe leaked in the source code.*

5.2 Experiment Execution

We perform multiple experiments to investigate continuous coverage for various permutations of the parameters sample size *m* and window size *n*. We start our experiments by generating samples with the following sample sizes $m: \{1, 2, 4, 8, 16, 32, 64, 128, 256,$

[8]https://doi.org/10.5281/zenodo.7885905

[9]Anonymous

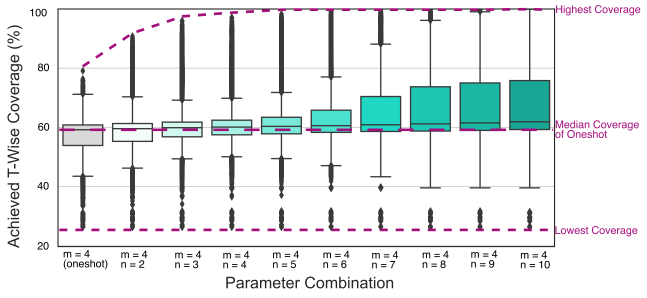


Figure 2: Comparison between one-shot coverage and continuous t-wise coverage for $n = 4$. Each box accumulates the results for all of our subject systems.

$512, 1024\}$ for each commit in the commit history of each subject system under consideration. The samples are generated by using an implementation of uniform random sampling contained in FeatureIDE [10]. For each calculated sample, we calculate the ratio of t-wise feature interaction coverage for multiple values of t (e.g., $t = 1$, $t = 2$) using our continuous t-wise coverage metric with a window size of $n = 1$. Calculating continuous t-wise coverage with a window size of 1 is equivalent to calculating one-shot coverage, as discussed in Section 4.3. Therefore, we use the calculated coverage values as a baseline for our evaluation. In addition to calculating the baseline coverage for each generated sample, we compute continuous t-wise coverage values for the following window sizes: $n : \{2, 3, 4, 5, 6, 7, 8, 9, 10\}$. In practice, this means that for each commit of our subject systems, we calculate a continuous t-wise coverage for the parameter combinations $\{(m = 1, n = 1); (m = 2, n = 1); \ldots (m = 1024, n = 10);\}$ resulting in 110 coverage values per commit. We order the calculated coverage values for each subject system chronologically so that we can correlate the coverage values to changes in the variability model of the subject system. We use the calculated t-wise feature interaction coverage ratios to answer RQ1 by analyzing the behavior of continuous t-wise coverage over time compared to one-shot coverage. Furthermore, we use our results to investigate the trade-off between sample size m and window size n, to answer RQ2.

5.3 Results

For the analysis required for RQ1, we visualize the calculated values as a box plot (i.e.,Figure 2) and as scatter plot (i.e., Figure 3). Answering RQ2 requires a direct comparison between continuous t-wise coverage values of various parameter combinations, which we show as a heat map (i.e., Figure 4).

RQ1. Figure 2 shows accumulated results for one-shot coverage (grey box, far left) and continuous t-wise coverage (boxes in green gradient) measured across all subject systems for a sample size of $m = 4$. Each box in the plot represents a corresponding parameter combination of sample size (m) and window size (n). For continuous t-wise coverage, the results are shown for all window sizes used in the experiment. The y-axis of Figure 2 shows the measured feature interaction coverage in percent. It ranges from 20% to 100%, since for no parameter combination t-wise feature coverage was measured below 20%. We use three trend lines (shown in purple) to highlight the difference between the measurements. The bottom trend line indicates the smallest measured coverage. For each parameter combination, this value is 25%. The middle trend line shows the average over the results from all subject systems for the corresponding parameter combination. It increases continuously from a value of 60% (one-shot) to a value of 62% (continuous coverage with $n = 10$). The upper trend line shows the highest coverage measured for the corresponding parameter combination. There is a continuous increase from 80% one-shot coverage to 100% for all parameter combinations of continuous t-wise coverage with window size $n \geq 6$.

Figure 3 depicts the results for the subject system Fiasco as a surrogate. We visualize the calculated continuous coverage values for the parameter combination $\{(m = 16, n = 1); (m = 1024, n = 1); (m = 8, n = 10); (m = 128, n = 10)\}$ as representatives for our results. Our supplementary material[11] contains the results computed for the different subject systems and parameter combinations. The parameter combinations $(m = 16, n = 1)$ (i.e., lower bound) and $(m = 1024, n = 1)$ (i.e., upper bound) represent the behavior of one-shot coverage overtime on the Fiasco subject system. The parameter combination $(m = 8, n = 10)$, and $(m = 128, n = 10)$ represent the behavior of continuous coverage over time. On the x-axis of Figure 3, we show the version numbers of commits in our analyzed commit window in chronological order. On the y-axis, we show 2-wise feature interaction coverage in percent (%). The y-axis starts with 40% as the lowest value measured in this experimental setup. The highest value of the y-axis is 100%, indicating that all possible t-wise feature tuples are covered. A data point in the scatter plot represents the 2-wise feature interaction coverage achieved by a certain parameter combination for the respective commit. As representative for our over all results for Fiasco, we visualize data points for $m = 16$ (gray circles), and $m = 1024$ (gray diamonds) as examples for one-shot sampling (i.e., $n = 1$). Additionally we visualize the parameter combinations $(m = 8, n = 10)$ (green stars), $(m = 128, n = 10)$ (green triangles) for continuous coverage. We use a purple background color for commits that do not change the variability model. For the Fiasco, we observe that the variability model changes frequently throughout the system's evolution.

As depicted in Figure 3, one-shot coverage with sample size $m = 16$, varies between 46% to 66% (i.e., a variation of about 20%). For one-shot coverage with sample size $m = 1024$, we observe the highest coverage ratio at 94% and lowest at 64%, which shows a variation in coverage of 30%. For continuous coverage (parameter combination $(m = 8, n = 10)$), a general upwards trend can be observed between versions one and ten, which is exactly the window size used in this experimental setup. During this ramp-up phase, the coverage ratio increases from 49% to 81%. After that, small variations between 83% coverage at peaks and 78% as lowest values can be observed until a downward trend from version 24 to version 33 sets in. During this downward trend, the coverage ratio decreases from 78% to 67%.

[10]https://featureide.github.io/

[11]https://doi.org/10.5281/zenodo.7885905

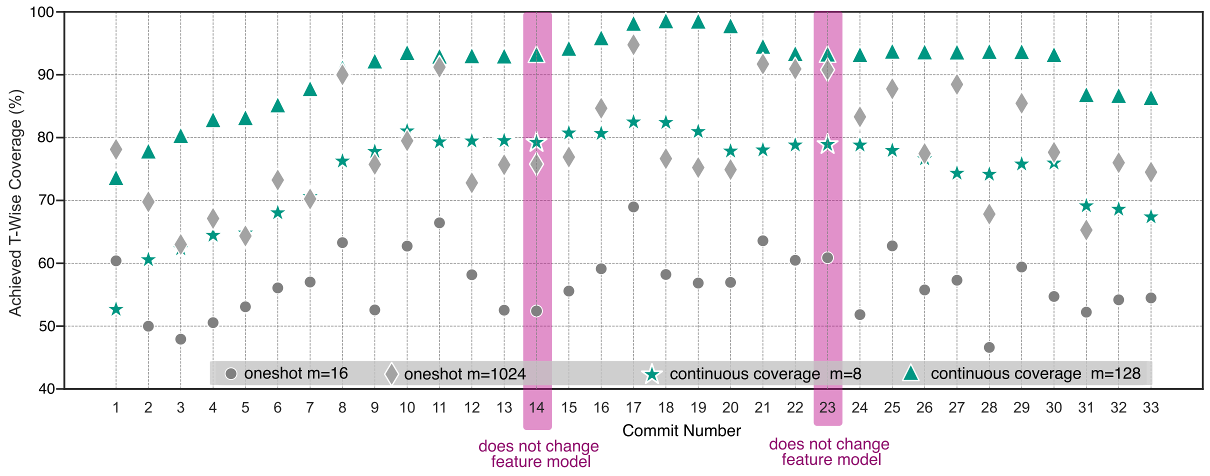


Figure 3: Results of measuring continuous t-wise coverage for (m = 8, n=10), and (m = 128, n=10), as well as one-shot coverage (m = 16, m = 512, and m = 1024) over time, for Fiasco.

RQ2. In RQ2, we analyze the influence of sample size and commit window size to continuous t-wise coverage. We calculate continuous t-wise coverage for various combinations of sample size m and the commit window size n. We use a heat map to visualize the differences and the distribution of t-wise feature interaction coverage between various parameter combinations. Figure 4, shows the calculated results for our subject systems BusyBox, Fiasco, Soletta, and Uclibc in four different heat maps. We represent and discuss our results using the Fiasco subject system as a surrogate.

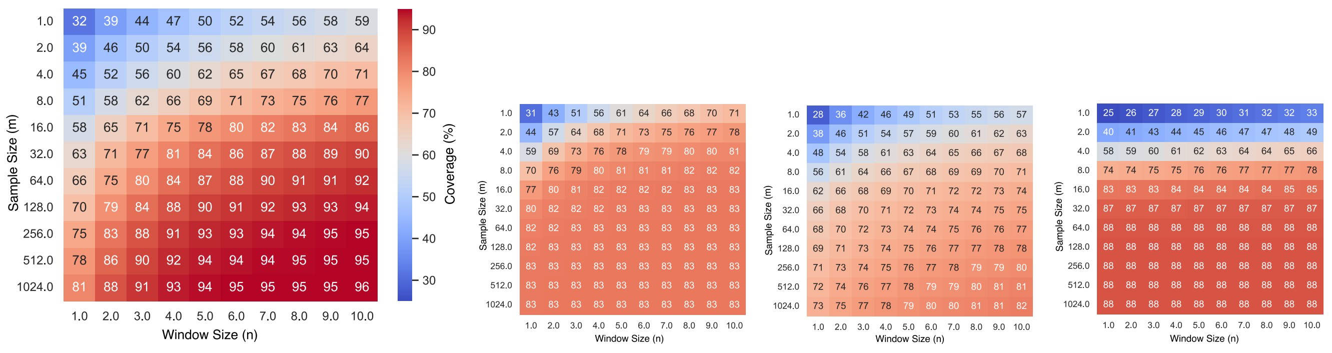
The x-axis of the heat map represents the window sizes used for calculating continuous t-wise coverage, while the y-axis represents the sample sizes. An entry in the heat map represents the average pair-wise feature interaction coverage ratio calculated with continuous t-wise coverage for the combination of commit window size (n) and sample size m. For instance, the t-wise feature interaction coverage ratio of the entry in the top left corner of the heat map is calculated for the parameter combination ($n = 1, m = 1$). Each rectangle is shaded using a color scale that reaches from blue (low coverage ratio) to red (high coverage ratio). The color scale is shown on the right side of Figure 4. A dark blue shade represents the lowest possible t-wise feature interaction ratio, while a dark red shade represents the highest ratio.

While the concrete coverage ratios differ for all depicted subject systems, we can observe a trend in all of them. This trend indicates that the lowest coverage ratio (e.g., 32% for Fiasco) is always achieved for a parameter combination of $m = 1$ and $n = 1$ (i.e., the upper-left corner of the heat map). From there, increasing one of the parameters increases the calculated coverage value. The highest coverage values are always reached for the parameter combination $m = 1024$ and $n = 10$. As for our surrogate Fiasco, the highest coverage value is 96%.

5.4 Discussion

RQ1: As a basis for the discussion of this research question, we use the results computed for Fiasco as an example. For the other subject systems, we observe slight deviations in details such as the concrete ratio of computed feature interaction coverage. However, the same general trend for continuous t-wise coverage can be observed for all subject systems, as seen in Figure 2. It shows that continuous t-wise coverage achieves at least the minimum t-wise feature interaction coverage for all window sizes across all of our subject systems. It also shows a slight increase in the achieved t-wise feature interaction coverage on average, while a significant increase in the highest value for t-wise feature interaction coverage is shown. The detailed analysis for the Fiasco subject system underlines these results. Our results for Fiasco show that continuous t-wise coverage increases over time until the maximum window size is reached. Using a sample size of $m = 8$ and a window size of $n = 10$, increases the achieved coverage ratio by about 32% over the first ten system versions. During this ramp-up phase, continuous t-wise coverage considers feature tuples from each previously used sample, which explains the strong increase in the coverage ratio. Between the 10th and 23rd system versions, the measured t-wise coverage ratio no longer increases. Instead, it remains stable with a small fluctuation of about 5%. This behaviour is expected because continuous t-wise coverage replaces feature tuples covered by older system versions with newer feature tuples. This limits the number of feature interaction tuples that can be covered, meaning the maximum t-wise interaction coverage ratio is also limited. From the 24th to the 33rd system version, we discovered a downward trend in the achieved coverage ratio of continuous t-wise coverage (i.e., a reduction of 8%). This trend is also visible for the calculated baseline configurations of one-shot coverage, which indicates that achieving a high t-wise feature interaction ratio for the last system versions of a Fiasco is generally hard.

In comparison to the baseline results (i.e., upper and lower limit) from Figure 3, we observe that after the ramp-up phase, continuous t-wise coverage is higher than the lower baseline (i.e., $(m = 16, n = 1)$), but remains mostly lower than the upper baseline (i.e., $(m = 1024, n = 1)$). This behaviour is expected for the shown parameter combinations since after the ramp-up phase, continuous t-wise coverage accumulates feature-interaction tuples from



(a) Fiasco **(b) Soletta** **(c) Uclibc** **(d) Busybox**

Figure 4: Achieved t-wise feature interaction coverage using various parameter combinations (m and n) for continuous t-wise coverage. The heatmaps show for all subject systems that increasing either the sample size or the window size increases pair-wise feature interaction coverage.

800 configurations per system version. Compared to 16 configurations, feature tuples from 800 configurations should achieve a higher t-wise feature interaction coverage, while compared to 1024 configurations, the achieved t-wise feature interaction coverage should be lower. We observe similar results for the other subject systems in our experimental setup.

RQ1.1: In our experiments, continuous t-wise coverage is comparable to one-shot coverage within the limitations of certain parameter combinations. For instance, we cannot expect that continuous t-wise coverage using the parameter combination $m = 2$ and $n = 10$ achieves higher t-wise interaction coverage than one-shot coverage using a sample size of $m = 1024$. However, Figure 3 shows that after the second commit, continuous t-wise coverage with the parameter combination $m = 8$ and $n = 10$ reaches a higher t-wise feature interaction coverage ration than one-shot coverage with double the sample size (i.e. $m = 16$).

RQ1.2: Our results show that reaching a certain amount of pair-wise feature interaction coverage with continuous t-wise coverage depends on the parameter combination of sample size (m) and window size (n). The visualized results for the parameters $m = 8$ and $n = 10$ never reach 98% of pair-wise coverage, while increasing the sample size to 128 is enough to reach this threshold for Fiasco One-shot sampling with a sample size of 1024 does not achieve pair-wise feature interaction coverage either. Therefore, t is particularly difficult to reach this threshold using uniform random sampling. We observe similar results for Busybox, Soletta, and Uclibc.

RQ1.3: The variation in coverage mostly depends on the nature of the commits. If a feature model is changed often, continuous coverage will be more varied. If a feature model is changed drastically, the variation will be larger. Our results for Fiasco, where the feature model changes frequently, show that the calculated coverage ratios remain decently stable with at most 6% of fluctuation between consecutive system versions. We observe similar results for the other subject systems, with slightly higher fluctuation rates. Considering that all samples are generated with uniform random sampling, we argue that those results indicate that continuous coverage is resilient to most changes in the feature model.

RQ1 Summary: Continuous coverage has a great advantage compared to one-shot coverage. For one, if the same sample size is used, the achieved coverage outperforms one-shot coverage overtime. Consequently, smaller sample sizes can achieve the same coverage over time. Another benefit of continuous coverage is its resilience towards feature model changes. First, changes to the feature model do not affect one-shot coverage, as the sample configurations are always computed wrt. the current feature model version. However, continuous coverage achieves higher coverage nevertheless and suffers smaller coverage losses over time, as the knowledge of previous samples mitigates the impact of feature model changes.

RQ2: With RQ2, we aim to identify a beneficial trade-off between sample size (m) and commit window size (n). Therefore, we use continuous t-wise coverage to calculate the pair-wise feature interaction coverage ratio for various combinations of m and n. We visualize the results of those calculations as a heat map, shown in Figure 4. The lowest pair-wise feature interaction coverage ratio for all systems is computed for the combination ($m = 1, n = 1$). We observe that increasing m and n simultaneously increases the ratio of computed t-wise feature interaction coverage the fastest. For instance, 81% pair-wise feature interaction coverage for Fiasco is achieved for a combination of $m = 32$ and $n = 4$ when using continuous t-wise coverage for the calculation. Achieving the same coverage in one commit ($n = 1$), requires 1024 configurations ($m = 1024$), which results in high testing effort for the commit. Minimizing the testing effort by considering one configuration per commit ($m = 1$) increases the risk of missing a fault because in general only about 25% of pair-wise coverage is achieved per commit, and it takes a long time to achieve 100% of pair-wise coverage. Testing 16 configurations ($m = 16$) per commit over eight commits ($n = 7$) achieves a better pair-wise coverage for Fiasco as testing 1024 configurations in one commit. This averages the testing effort and the risk of missing faults per commit, which is a beneficial trade-off for testing a configurable system.

The results achieved in our experiments indicate that the ratio of pair-wise feature interaction coverage calculated with continuous t-wise coverage depends on the total number of configurations considered over time. To achieve equal coverage ratios, we can reduce the number of considered configurations per commit, but then we must increase the size of our commit window. This discovery means that splitting coverage between multiple commits can reduce testing effort per commit while still achieving t-wise feature interaction coverage. However, the more commits it takes to achieve t-wise feature interaction coverage, the higher is the risk of losing the traceability of faults.

Based on our results, we can answer RQ2 that there is a direct trade-off between testing effort (i.e., sample size *m*) and testing uncertainty (i.e., commit window size *n*). By investigating different subject systems, we identified that the optimal trade-off depends on the system itself and the testing strategy applied to the system.

5.5 Threats to Validity

Internal Validity. A threat to internal validity is that we use pre-processed variability models as input for our evaluation. Means of acquiring those models (e.g., using Tseitin transformation [16]) may influence the validity of our results. However, we argue that the variability models used in our experiments must be of high quality because they were already used in peer-reviewed publications. A second threat is that we may have introduced faults while implementing our evaluation tool kit. We mitigate this threat by using the well-validated FeatureIDE library as a resource for utility implementation wrt. processing and analyzing variability models.

External Validity. A threat to external validity is that we only use four subject systems with their variability modeled in KConfig for our evaluation. Using subject systems that use different variability languages may cause differing evaluation results. However, we argue that KCofig is a well-known language to model variability used in many configurable software systems. To mitigate this threat even further, we selected small and middle-sized real-world systems, which are often used to evaluate new concepts in the software-product line domain [15, 32]. The next threat to external validity is choosing uniform random sampling as underlying sampling approach over a t-wise sampling approach. Using a t-wise sampling approach would have resulted in generally higher and less fluctuating values for continuous t-wise coverage, making it easier to show the capabilities of our new metric. However, those results would always be influenced by the concrete implementation of the sampling algorithm. Choosing random configurations as input for calculating continuous t-wise coverage threatens the external validity because random fluctuations may appear. We mitigate the influence of random selection by performing our experiments ten times, each with different randomly selected configurations. The results of those experiment executions are accumulated using the arithmetic mean value. The last threat to external validity is that our results only measure pair-wise feature interaction coverage. Results for high t-wise coverage strengths will differ from our results, wrt. calculated coverage ratios. However, we expect that the general behaviour of continuous t-wise coverage and the comparison to one-shot coverage will remain the same.

Influence of Solution Space Changes. Our evaluation focuses on the influence of problem space changes (i.e., feature model changes) on continuous t-wise coverage. An empirical evaluation of solution space changes (i.e., feature code changes) was beyond the scope of this paper and is part of our ongoing research. Nevertheless, solution space changes are an essential part of the evolution of a configurable system, and must be considered when arguing about continuous t-wise coverage calculation.

In practice, the solution space realization of a feature may change dramatically between two commits. For instance, the source code of a feature may be replaced completely by a refactoring action. In context of a continuous testing approach, this means that the changed features and all feature tuples which they are included in must be retested. Therefore, considering solution space changes when accumulating t-wise feature tuples over time requires the identification of changed features. Feature changes can be found by applying a change impact and dependency analysis. All previously covered feature tuples that contain a changed feature cannot be considered as covered anymore. Therefore, all feature tuples that contain at least one changed feature (i.e., impacted feature tuples) must be removed from the set of already covered feature tuples, when accumulating t-wise feature interaction coverage over time.

We estimate that the described approach for considering solution space changes will result in a large number of impacted feature tuples. Since those feature tuples will be removed from the set of covered tuples, the continuous t-wise coverage for the subject system under consideration is also be reduced. Especially changing the solution space of multiple features will reduce continuous t-wise coverage drastically in between commits. Accumulating t-wise coverage for development phases with solution space changes in many features does not result in more benefits than one-shot sampling. However, in development phases where only a few features are changed between commits accumulating t-wise coverage over time, will be beneficial in reducing the test effort per commit.

6 RELATED WORK

This section elaborates on previous research related to our concept of continuous t-wise coverage. We start by discussing various types of state-of-the-art one-shot sampling algorithms [1, 5, 11, 13, 15]. After that, we elaborate on the existing regression testing procedures for configurable systems [17, 19, 20].

6.1 One-Shot Sampling Algorithms

Over the past decades, many different t-wise feature interaction sampling algorithms were introduced. Varshosaz et al. [36] present an extensive overview of current research of sampling algorithms. Currently, YASA [15] shows the best performance values between the established t-wise feature interaction sampling algorithms, including Chvatal [5], ICPL [13], and IncLing [1]. Those algorithms are primarily used to generate a sample for the most recent commit of the system. Applying them in a continuous sampling approach is challenging, because they use a metric for t-wise feature interaction coverage that cannot consider previously calculated configurations (i.e., one-shot coverage). One-shot coverage does not allow to make an informed trade-off between testing effort and the risk of missing faults while testing a commit, which makes it impossible to know which feature tuples are already covered by previous samples. With continuous t-wise coverage, we introduce a coverage metric that considers the evolution of the system to make this informed trade-off possible and enable the use of continuous sampling.

6.2 Regression Testing of Configurable Systems

Over the past decades regression testing for configurable systems was mostly applied for the selection of test cases [17, 20–23]. The selection and prioritization of product configurations that consider the systems evolution, is not as well researched [19, 25]. The following paragraphs elaborate on how the systems evolution was previously used for test case and configuration selection strategies.

Test Case Selection and Prioritization. Lochau et al. [21–23] proposed a workflow for incremental testing of configurable systems, where test results are reused for consecutively tested product configurations of the same system version. Lity et al. [17, 20] extend this workflow so that test artifacts and test results are also usable for testing versions of variants. The presented approaches work on the level of test case selection, while our concept of continuous t-wise coverage addresses the configuration selection over the evolution of a configurable system. However, a combination between approach on configuration level and existing approaches on test case level is possible, by using information about previously covered feature tuples in the selection of test-cases that address them specifically.

Product Configuration Selection and Prioritization. Lima et al. [19] propose strategies to prioritize all configurations under test to reduce the testing effort for evolving configurable systems in a regression-based test environment. Marijan et al. [25] introduce an algorithm to calculate configurations that achieve single feature interaction coverage for evolving configurable systems. Their algorithm is intended to be explicitly used for regression testing in continuous integration. None of the existing approaches uses the ratio of achieved t-wise feature interaction coverage over time as a prioritization criterion because the current metric is not designed for this use case. With continuous t-wise coverage, we propose a metric exactly for this use case. Therefore, we may be able to guide the configuration selection and prioritization in sample-based regression testing to be more efficient.

7 CONCLUSION AND FUTURE WORK

In this paper, we present continuous t-wise coverage as a new concept to calculate the ratio of t-wise feature interaction coverage. Our concept accumulates feature tuples tested for commits contained in a commit window and uses the accumulated set of feature tuples for calculating the t-wise feature interaction coverage. Continuous t-wise coverage allows distributing the testing effort for covering all t-wise feature interactions between multiple commits (i.e., reducing the testing effort per commit). However, distributing the testing effort increases the risk of missing faults per commit because it takes more commits to cover all t-wise feature interactions. We evaluated our concept on four real-world systems (BusyBox, Fiasco, Soletta, and Uclibc). Our evaluation shows a direct trade-off between the configurations tested per commit and the number of commits it takes to achieve t-wise interaction coverage. We discover that continuous t-wise coverage makes it possible to demonstrate trade-offs of existing testing settings, and analyze their efficiency for frequently evolving configurable systems.

In the future, we aim to expand our evaluation to more real-world systems with richer evolution histories to show that our results are valid for a broader set of systems. Further, we want to analyze how continuous t-wise coverage behaves for higher strength values of t-wise feature interaction coverage (i.e., $t = 3$, $t = 4$, $t = 5$). We also plan to measure continuous t-wise coverage based on established t-wise sampling algorithms, such as YASA, and compare those results with the results presented in this paper. Since, in the current evaluation, we only investigate the influence of problem space changes (i.e., changed feature models), it is part of our future research to investigate solution space changes (i.e., changed source code) as well. Besides strengthening our existing evaluation results, we aim to develop an incremental sampling algorithm that uses continuous t-wise coverage to select configurations. We plan to analyze the actual trade-off between the risk of missing faults and reducing test effort per commit by using concepts of mutation testing to introduce faults into the subject systems and execute test cases on the mutated systems. Comparing the fault finding rate between continuous t-wise testing and one-shot testing will then reveal its effectiveness.

ACKNOWLEDGMENTS

This work was supported by funding from the topic Engineering Secure Systems of the Helmholtz Association (HGF) and by KASTEL Security Research Labs (46.23.03). This publication is also partially based on the research project SofDCar, which is funded by the German Federal Ministry for Economic Affairs and Climate Action under the grant number 19S21002. The authors would like to thank the Ministry of Science, Research and Arts of the Federal State of Baden-Württemberg for the financial support of the projects within the Innovations Campus Future Mobility (ICM). We thank Niklas Mund for developing first concepts and prototype implementation during his master thesis. We also thank Tim Witschel and Elias Kuiter for providing the framework implementations to analyze KConfig-based variability models.

REFERENCES

[1] Mustafa Al-Hajjaji, Sebastian Krieter, Thomas Thüm, Malte Lochau, and Gunter Saake. 2016. IncLing: Efficient Product-line Testing Using Incremental Pairwise Sampling. 144–155.
[2] Paul Ammann and Jeff Offutt. 2016. *Introduction to software testing.* Cambridge University Press.
[3] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines.*
[4] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. 2013. A Survey of Variability Modeling in Industrial Practice. 7:1–7:8.
[5] Vasek Chvatal. 1979. A Greedy Heuristic for the Set-Covering Problem. *Mathematics of operations research* 4, 3 (1979), 233–235. https://pubsonline.informs.org/doi/abs/10.1287/moor.4.3.233
[6] Paul Clements and Linda Northrop. 2001. *Software Product Lines: Practices and Patterns.*
[7] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. 2008. Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints: A Greedy Approach. 34, 5 (2008), 633–650.
[8] Paul M Duvall, Steve Matyas, and Andrew Glover. 2007. *Continuous integration: improving software quality and reducing risk.* Pearson Education.
[9] Martin Fowler and Matthew Foemmel. 2006. Continuous integration.
[10] Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, and Benoit Baudry. 2019. Test them all, is it worth it? Assessing configuration sampling on the JHipster Web development stack. 24, 2 (2019), 674–717. https://doi.org/10.1007/s10664-018-9635-4

[11] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. 2011. Properties of Realistic Feature Models Make Combinatorial Testing of Product Lines Feasible. 638–652. https://link.springer.com/chapter/10.1007/978-3-642-24485-8_47

[12] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. 2012. An Algorithm for Generating T-Wise Covering Arrays from Large Feature Models. 46–55.

[13] Martin Fagereng Johansen, Øystein Haugen, Franck Fleurey, Anne Grete Eldegard, and Torbjørn Syversen. 2012. Generating Better Partial Covering Arrays by Modeling Weights on Sub-Product Lines. 269–284.

[14] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-21.

[15] Sebastian Krieter, Thomas Thüm, Sandro Schulze, Gunter Saake, and Thomas Leich. 2020. YASA: Yet Another Sampling Algorithm. Article 4, 10 pages.

[16] Elias Kuiter, Sebastian Krieter, Chico Sundermann, Thomas Thüm, and Gunter Saake. 2023. Tseitin or Not Tseitin? The Impact of CNF Transformations on Feature-Model Analyses. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 110, 13 pages. https://doi.org/10.1145/3551349.3556938

[17] Remo Lachmann, Sascha Lity, Sabrina Lischke, Simon Beddig, Sandro Schulze, and Ina Schaefer. 2015. Delta-oriented test case prioritization for integration testing of software product lines. In *Proceedings of the 19th International Conference on Software Product Line*. 81–90.

[18] Jihyun Lee, Sungwon Kang, and Danhyung Lee. 2012. A Survey on Software Product Line Testing. 31–40.

[19] Jackson A. Prado Lima, Willian D. F. Mendonça, Silvia R. Vergilio, and Wesley K. G. Assunção. 2020. Learning-based prioritization of test cases in continuous integration of highly-configurable software *(SPLC '20)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/3382025.3414967

[20] Sascha Lity, Manuel Nieke, Thomas Thüm, and Ina Schaefer. 2019. Retest test selection for product-line regression testing of variants and versions of variants. *JSS* 147 (Jan. 2019), 46–63. https://doi.org/10.1016/j.jss.2018.09.090

[21] Malte Lochau. 2012. *Model-Based Conformance Testing of Software Product Lines*. Ph. D. Dissertation. Verlag Dr. Hut.

[22] Malte Lochau, Sascha Lity, Remo Lachmann, Ina Schaefer, and Ursula Goltz. 2014. Delta-oriented model-based integration testing of large-scale systems. *Journal of Systems and Software* 91 (2014), 63–84.

[23] Malte Lochau, Ina Schaefer, Jochen Kamischke, and Sascha Lity. 2012. Incremental Model-Based Testing of Delta-oriented Software Product Lines. In *ACM Transactions on Applied Perception*. 67–82.

[24] Roberto E. Lopez-Herrejon, Stefan Fischer, Rudolf Ramler, and Aalexander Egyed. 2015. A First Systematic Mapping Study on Combinatorial Interaction Testing for Software Product Lines. 1–10.

[25] Dusica Marijan, Arnaud Gotlieb, and Marius Liaaen. 2019. A learning algorithm for optimizing continuous integration development and testing practice. 49, 2 (2019), 192–213. https://doi.org/10.1002/spe.2661

[26] Dusica Marijan, Arnaud Gotlieb, Sagar Sen, and Aymeric Hervieu. 2013. Practical Pairwise Testing for Software Product Lines. 227–235.

[27] John McGregor. 2010. Testing a Software Product Line. In *Testing Techniques in Software Engineering*. 104–140.

[28] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Sarah Nadi, and Rohit Gheyi. 2015. The Love/Hate Relationship with the C Preprocessor: An Interview Study. In *29th European Conference on Object-Oriented Programming (ECOOP 2015)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 495–518.

[29] Mathias Meyer. 2014. Continuous integration and its tools. *IEEE Software* 31, 3 (2014), 14–16.

[30] Daniel-Jesus Munoz, Jeho Oh, Mónica Pinto, Lidia Fuentes, and Don Batory. 2019. Uniform Random Sampling Product Configurations of Feature Models That Have Numerical Features. 289–301.

[31] Changhai Nie and Hareton Leung. 2011. A Survey of Combinatorial Testing. *Comput. Surveys* 43, 2, Article 11 (2011), 11:1–11:29 pages.

[32] Tobias Pett, Sebastian Krieter, Tobias Runge, Thomas Thüm, Malte Lochau, and Ina Schaefer. 2021. Stability of Product-Line Samplingin Continuous Integration. In *15th International Working Conference on Variability Modelling of Software-Intensive Systems*. 1–9.

[33] Tobias Pett, Thomas Thüm, Tobias Runge, Sebastian Krieter, Malte Lochau, and Ina Schaefer. 2019. Product Sampling for Product Lines: The Scalability Challenge. 78–83.

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

[35] Sean Stolberg. 2009. Enabling agile testing through continuous integration. In *2009 agile conference*. IEEE, 369–374.

[36] Mahsa Varshosaz, Mustafa Al-Hajjaji, Thomas Thüm, Tobias Runge, Mohammad Reza Mousavi, and Ina Schaefer. 2018. A Classification of Product Sampling for Software Product Lines. 1–13.